DFKI Oz Documentation Series

The Oz Notation

Martin Henz

Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany

January 26, 1995

DFKI Oz Documentation Series

Gert Smolka and Ralf Treinen, editors

- An Oz Primer
- Constraint Programming in Oz
- Programming in Oz
- The Definition of Kernel Oz
- The Oz Notation
- The Oz Standard Modules
- DFKI Oz User's Manual
- Window Programming in DFKI Oz
- Open Programming in DFKI Oz
- DFKI Oz Demos

More information about DFKI Oz, the documentation, and the system can be obtained via WWW from

http://ps-www.dfki.uni-sb.de/oz/

The documentation and the system are also available by anonymous ftp from ps-ftp.dfki.uni-sb.de, directory pub/oz.

For specific questions and bug reports, you may mail to oz@dfki.uni-sb.de. To join the Oz users mailing list, contact oz-users-request@dfki.uni-sb.de.

The development of Oz is supported by the Bundesminister für Forschung und Technologie (Hydra, ITW 9105), the Esprit Basic Research Project ACCLAIM (contract EP 7195) and the Esprit Working Group CCL (contract EP 6028).

Cover Illustration by Andreas Schoch.

The Oz Notation

Martin Henz

Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
email: henz@dfki.uni-sb.de

January 26, 1995

Abstract

Oz is a concurrent language providing for functional, object-oriented, and constraint programming. This report defines how Oz program text is transformed into a Kernel Oz program. Kernel Oz has been designed to be the base for a formal definition of Oz, and therefore to be minimal in the number of syntactic constructs. Technically, Kernel Oz is able to express a wide range of programming styles, including functional and object-oriented programming. Being a purely relational language, however, Kernel Oz does not provide easy notational access to programming methods from functional and object-oriented programming, making it hard to fully exploit the capacities of the language.

It is such ergonomic considerations that lead to the development of the Oz Notation, where syntactic extensions provide convenient constructs for functional and object-oriented programming. The semantics of these extensions is defined in this document by their stepwise translation to Kernel Oz.

Contents

1 Introduction

This report defines how Oz program text, which is a sequence of characters, is transformed into a Kernel Oz program. This transformation is performed in five steps.

Lexical syntax. First, a given program text is transformed into a sequence of words. Each word represents a sequence of tokens.

Context-free syntax. The resulting sequence of tokens is transformed into a parse tree.

Programs. The parse tree is transformed into an abstract syntax tree, called program.

Standardized Programs. The program is standardized, eliminating a number of abbreviations that allow for more concise programs.

Kernel Programs. The standardized program is translated to a Kernel program eliminating nesting and state threading, extensions that support functional and object-oriented programming.

At each step, errors may occur. A text represents an Oz program, if it can be tokenized and parsed into a program which can be translated without error into a Kernel program.

Kernel programs [3], denoted by $\mathcal{E}$, are kernel expressions with a "hole" $\bullet$. This hole may be filled later with another program $\mathcal{E}'$, which may use local variables of $\mathcal{E}$ whose scope included the hole. We write $\mathcal{E}[\mathcal{E}']$ for the program obtained by replacing $\bullet$ in $\mathcal{E}$ by $\mathcal{E}'$.

This concept provides for an interactive programming environment by allowing to incrementally extend the scope of variables over program boundaries. A stream of incoming program texts $T_1, T_2, T_3, \cdots$ is transformed into a stream of programs $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \cdots$. Each of the programs replaces the hole in the previous one, resulting in $\mathcal{E}_1[\mathcal{E}_2[\mathcal{E}_3[\cdots]]]$. Note that we use the symbol $\cdots$ for ellipsis; the character sequence ... is an Oz keyword.

During the transformation, new variables are introduced that do not occur in the given program text. New variables that are not bound locally are either bound by the Kernel prelude $\mathcal{E}_k$ given in [3] or by the notational prelude $\mathcal{E}_n$, which is summarized in Section 7.

The notational prelude in turn uses some Standard procedures, defined in the Standard prelude $\mathcal{E}_s$ described in [1]. Therefore, entering user programs $\mathcal{E}_1, \mathcal{E}_2, \cdots$ generates the following stream of programs.

$$\mathcal{E}_k[\mathcal{E}_n[\mathcal{E}_s[\mathcal{E}_1[\mathcal{E}_2[\cdots]]]]]$$

2 Lexical Syntax

A program text is a sequence of characters. In this section, we describe how such a sequence is split into a sequence of words. Each word represents zero or more tokens such that the result is a sequence of tokens. There are 9 classes of words, described by regular expressions. We use the following notation to compose regular expressions e: **[**e**]** for an optional expression e, e^+ for one or more expressions e, e^* for zero or more expressions e, e_1 **|** e_2 for either expression e_1 or expression e_2, and **(**e**)** for grouping of expressions.

The following definitions represent the classes of characters used in the remainder. For characters, we use ISO 8859-1. For example, the character `a` has code 97. A convenient way to see the character codes on UNIX-based systems is by using the UNIX command `man ascii`.

⟨*lower*⟩	::=	`a` **|** ··· **|** `z`	*lower-case letter*
⟨*upper*⟩	::=	`A` **|** ··· **|** `Z`	*upper-case letter*
⟨*digit*⟩	::=	`0` **|** ··· **|** `9`	*decimal digit*
⟨*alphaNum*⟩	::=	⟨*lower*⟩ **|** ⟨*upper*⟩ **|** ⟨*digit*⟩ **|** _	*alpha-numerical character*
⟨*atomChar*⟩	::=	any, except ´, \, and code 0	*atom character*
⟨*stringChar*⟩	::=	any, except ", \, and code 0	*string character*
⟨*variableChar*⟩	::=	any, except `, \, and code 0	*variable character*
⟨*escape*⟩	::=	`a` **|** `b` **|** `f` **|** `n` **|** `r` **|** `t` **|** `v` **|** `\` **|** ´ **|** `"` **|** `	*escape character*
⟨*oct*⟩	::=	`0` **|** ··· **|** `7`	*octal digit*
⟨*hex*⟩	::=	`0` **|** ··· **|** `9` **|** `a` **|** ··· **|** `f` **|** `A` **|** ··· **|** `F`	*hexadecimal digit*

In the classes of words ⟨*variable*⟩, ⟨*atom*⟩, and ⟨*string*⟩, we use pseudo-characters, which represent single characters in different notations.

⟨*pseudoChar*⟩ ::= `\` **(**⟨*oct*⟩⟨*oct*⟩⟨*oct*⟩ **|** `x`⟨*hex*⟩⟨*hex*⟩ **|** ⟨*escape*⟩**)**

Pseudo-characters allow to enter any 8 bit character using octal or hexadecimal notation. Octal notation is restricted to numbers less than 256. The pseudo-characters `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v` denote special purpose characters, and `\\`, `\´`, `\"`, `\`` denote their second component character.

2.1 Keywords

⟨*keyword*⟩ ::= **andthen** | **attr** | **case** | **class** | **create** | **declare** | **div**
| **else** | **elsecase** | **elseif** | **elseof** | **end** | **extern** | **false**
| **feat** | **fi** | **from** | **fun** | **if** | **in** | **local** | **meth** | **mod**
| **not** | **of** | **or** | **OR** | **orelse** | **proc** | **ro** | **RO** | **self** | **then**
| **thread** | **true** | **with**
| `{` | `}` | `(` | `)` | `[` | `]` | `|` | `#` | `:` | `...` | `<=` | `=` | `.`
| `[]` | `@` | `<-` | `<<` | `>>` | `$` | `!` | `~` | `+` | `-` | `*` | `/`
| `==` | `\=` | `<` | `=<` | `>` | `>=`
| `=:` | `\=:` | `<:` | `=<:` | `>:` | `>=:` | `::` | `\::` | `:::` | `\:::`

Each keyword represents itself as token.

2.2 Variables

⟨*variable*⟩ ::= ⟨*upper*⟩⟨*alphaNum*⟩* | `` `(⟨*variableChar*⟩ | ⟨*pseudoChar*⟩)*` ``

A word of the form ⟨*variable*⟩ represents a variable token of the form $(\!|variable, n^+|\!)$ where n^+ is the sequence of characters that make up the word, including the possibly surrounding ` characters. For example, the word `Xs` represents the token $(\!|variable, 88\ 115|\!)$ and the word `` `\n` `` represents the token $(\!|variable, 96\ 10\ 96|\!)$. The set of all variable tokens is denoted by $(\!|variable|\!)$.

2.3 Atoms

⟨*atom*⟩ ::= ⟨*lower*⟩⟨*alphaNum*⟩* | ´(⟨*atomChar*⟩ | ⟨*pseudoChar*⟩)*´

A word of the form ⟨*atom*⟩ represents an atom token of the form $(\!|atom, n^*|\!)$ where n^* is the sequence of characters that make up the word, excluding the possibly surrounding ´ characters. For example, the word `atom` represents the token $(\!|atom, 97\ 116\ 111\ 109|\!)$ and the word ´\n´ represents the token $(\!|atom, 10|\!)$. The set of all atom tokens is denoted by $(\!|atom|\!)$.

2.4 Constructors

⟨*label*⟩ ::= (⟨*variable*⟩ | ⟨*atom*⟩)(

A word of the form ⟨*label*⟩ represents a sequence of two tokens. The first one is a label token either of the form $(\!|atomlabel, n^*|\!)$ or of the form $(\!|variablelabel, n^*|\!)$ (similar to the corresponding tokens for words of the form ⟨*variable*⟩ and ⟨*atom*⟩). The second one is the keyword token `(`. For example, the word `Xs(` represents the tokens $(\!|variablelabel, 88\ 115|\!)$ and `(`. The set of all variable label tokens is denoted by $(\!|variablelabel|\!)$. The set of all atom label tokens is denoted by $(\!|atomlabel|\!)$.

2.5 Integers

$$\langle int \rangle \quad ::= \quad \mathbf{[}\texttt{\~{}}\mathbf{]}\ \langle digit \rangle^{+}$$

A word of the form $\langle int \rangle$ represents an integer token of the form $(\!|int, n|\!)$ where n represents the integer for which the word is the decimal representation. For example, the word `~159` represents the token $(\!|int, -159|\!)$. The set of all integer tokens is denoted by $(\!|int|\!)$.

2.6 Floats

$$\langle float \rangle \quad ::= \quad \langle int \rangle\ .\ \langle digit \rangle^{*}\ \mathbf{[(}e \mid E\mathbf{)}\ \langle int \rangle\mathbf{]}$$

A word of the form $\langle float \rangle$ represents a float token of the form $(\!|float, f|\!)$ where f represents the floating point number for which the word is the decimal representation. The letters `e` and `E` both indicate the exponent to 10. For example, the word `~1.5e2` represents the token $(\!|float, -1.5e2|\!)$. The set of all float tokens is denoted by $(\!|float|\!)$. The syntax of floats is implementation dependent in that syntactically correct floats may be refused by the compiler, if they cannot be represented by the implementation.

2.7 Strings

$$\langle string \rangle \quad ::= \quad \texttt{"}\mathbf{(}\langle stringChar \rangle \mid \langle pseudoChar \rangle\mathbf{)}^{*}\ \texttt{"}$$

A word of the form `"`$c_1 \cdots c_m$`"` represents a sequence of $m+2$ tokens of the form `[` $n_1 \ \cdots \ n_m$ `]`, where the n_i represent integer tokens according to the code of c_i. For example, the word `"ab"` represents the sequence of tokens `[` $(\!|int, 97|\!)$ $(\!|int, 98|\!)$ `]`.

2.8 Spaces

Spaces are tab (code 9), newline (code 10), vertical tab (code 11), form feed (code 12), carriage return (code 13), and blank (code 32). Spaces produce no tokens. This means that they are ignored, except that they separate words form each other.

2.9 Comment

A comment is:

- a sequence of characters from `%` until the next newline,

- a sequence of characters within comment brackets `/*` and `*/`, in which `/*` and `*/` are properly nested,
- the character `?`.

Comments produce no tokens. This means that they are ignored, except that they separate words form each other.

2.10 Lexical Analysis

The splitting of a sequence of characters using these regular expressions is not unique. We use the following rules to perform left to right longest match tokenization obtaining either error or a unique sequence of tokens from a given sequence of characters. We start with the given sequence of characters and an empty sequence of tokens.

1. If the sequence of characters is empty, we append the program marker token • to the end of the sequence of tokens obtained so far, and return the result.
2. Find the set of all regular expressions that recognize a prefix of the sequence of characters of maximal length (longest match).
3. This set is either empty or has one or two elements.
   - If the set is empty, the given sequence of characters does not represent an Oz program, a lexical error is reported and we stop the process.
   - If the set has one element, the corresponding generated tokens are added to the back of the sequence of tokens obtained so far and we proceed with 1.
   - If the set has two elements, then one of the corresponding tokens must be a keyword token. This one is added to the back of the sequence of tokens obtained so far and we proceed with 1.

2.11 Lexical Errors

Lexical errors result from using the characters `& , \ ; ^` outside of variables, atoms or strings, from using the variable, atom, or string delimiters `` ` ´ " `` not paired, from using comment brackets `/*` and `*/` not properly nested and from using `\` within variables, atoms or strings, but outside of ⟨*pseudoChar*⟩.

3 Context-free Syntax

In this section, the language of token sequences that are considered is described using a context-free grammar.

We use the following variation on BNF: **[**e**]** for an optional expression e, e^+ for one or more expressions e, e^* for zero or more expressions e, e_1 **|** e_2 for either expression e_1 or expression e_2, and **(**e**)** for grouping of expressions.

As terminal symbols, we use the keywords from the previous section and the tokens ⟨*variable*⟩, ⟨*atom*⟩, ⟨*variablelabel*⟩, ⟨*atomlabel*⟩, ⟨*int*⟩, and ⟨*float*⟩.

3.1 Programs

$\mathcal{E}$	::=	•	*program marker*
	|	$E\ \mathcal{E}$	*composition*
	|	**local** E **in** $\mathcal{E}$ **end**	*declaration*
	|	**declare** E **(**• **|** **in** $\mathcal{E}$**)**	*program declaration*

3.2 Expressions

E	::=	**true** | **false**	*success and failure*
	|	t **=** t	*equation*
	|	t (**==** | **\=** | **<** | **=<** | **>** | **>=**) t	*comparison*
	|	t (**=:** | **\=:** | **<:** | **=<:** | **>:** | **>=:**) t	*finite domain comparison*
	|	t (**::** | **\::** | **:::** | **\:::**) t	*finite domain constraint*
	|	**proc** {t t^*} B **end**	*procedure definition*
	|	**fun** {t t^*} b **end**	*function definition*
	|	E E	*composition*
	|	**local** E **in** E **end**	*declaration*
	|	{t t^*}	*application*
	|	**if** CL (**elseif** CL)* [**else** B] **fi**	*if conditional*
	|	**case** EL (**elsecase** EL)* [**else** B] **end**	*case conditional*
	|	**or** D (**[]** D)$^+$ **ro**	*disjunction*
	|	**OR** D (**[]** D)$^+$ **RO**	*non-distributing disjunction*
	|	**thread** B **end**	*thread creation*
	|	**not** B **end**	*negation*
	|	(**create** | **class**) t	*object definition*
		[**from** t^+]	(**from**, **attr**, **feat**, **with**
		[**attr** (t [:t])$^+$]	*parts may occur in*
		[**feat** (t [:t])$^+$]	*arbitrary order*)
		[**with** t]	
		(**meth** t B **end**)*	
		end	
	|	<< [t] t >>	*method application*
	|	t **<-** t	*assignment*
	|	**extern** B **end**	*external expression*
	|	v	*variable*
B	::=	[E **in**] E	*block*

We allow variables as expressions in order to provide for the declaration of variables without initialization.

3.3 Terms

r, s, t	::=	v	*variable*
	|	⟨*atom*⟩	*atom*
	|	l(*t** (*t* <= *t*)* [...])	*tuple constr. or match.*
	|	l((t:*t* [<= *t*])* [...])	*record constr. or match.*
	|	⟨*int*⟩ | ⟨*float*⟩	*integer or float*
	|	*t* | *t*	*cons list*
	|	[*t*+]	*fixed length list*
	|	*t* (# *t*)+	*mixfix tuple construction*
	|	*t* = *t*	*nested equation*
	|	*t* (== | \= | < | =< | > | >=) *t*	*nested comparison*
	|	**proc** {$ *t**} *B* **end**	*nested procedure def.*
	|	**fun** {$ *t**} *b* **end**	*nested function def.*
	|	**local** *E* **in** [*E*] *t* **end**	*nested declaration*
	|	{*t t**}	*nested application*
	|	_	*wildcard*
	|	$	*nesting marker*
	|	*t* . *t*	*field selection*
	|	~ *t* | *t* (+ | - | * | / | **div** | **mod**) *t*	*arithmetic*
	|	*t* (:: | \:: | ::: | \:::) *t*	*nested f.d. constraint*
	|	*t* **andthen** *t* | *t* **orelse** *t*	*boolean operators*
	|	**if** *CTL* (**elseif** *CTL*)* [**else** *b*] **fi**	*nested if conditional*
	|	**case** *ETL* (**elsecase** *ETL*)* [**else** *b*] **end**	*nested case conditional*
	|	**or** *DT* ([] *DT*)+ **ro**	*nested disjunction*
	|	**OR** *DT* ([] *DT*)+ **RO**	*nested non-distr. disjunct.*
	|	**thread** *b* **end**	*nested thread creation*
	|	(**create** | **class**) [$]	*nested object definition*
		[**from** *t*+]	(**from**, **attr**, **feat**, **with**
		[**attr** (*t* [:*t*])+]	*parts may occur in*
		[**feat** (*t* [:*t*])+]	*arbitrary order*)
		[**with** *t*]	
		(**meth** *t B* **end**)*	
		end	
	|	@ *t*	*attribute access*
	|	<< [*s*] *t* >>	*nested method appl.*
	|	**self**	*self for objects*
	|	(*b*)	*grouping*
b	::=	[*E* **in**] [*E*] *t*	*nested block*
l	::=	⟨*atomlabel*⟩ | [!] ⟨*variablelabel*⟩	*label*
v	::=	[!] ⟨*variable*⟩	*variable*

3.4 Clauses

CL	::=	C `[]` C)*	*list of if clauses*
C	::=	[E **in**] E **then** B	*if clause*
D	::=	[E **in**] E [**then** B]	*disjunctive clause*
EL	::=	t **of** t **then** B (`[]` F)* (**elseof** FL)*	*list of elseof clauses*
	|	t **then** B	*boolean case clause*
FL	::=	F (`[]` F)*	*list of case clauses*
F	::=	t **then** B	*case clause*

3.5 Nested Clauses

CTL	::=	CT (`[]` CT)*	*list of nested if clauses*
CT	::=	[E **in**] E **then** b	*nested if clause*
DT	::=	[E **in**] E **then** b	*nested disjunctive clause*
ETL	::=	t **of** t **then** b (`[]` FT)* (**elseof** FTL)*	*list of nested elseof clauses*
	|	t **then** b	*nested boolean case clause*
FTL	::=	FT (`[]` FT)*	*list of nested case clauses*
FT	::=	t **then** b	*nested case clause*

3.6 Operators

Precedences of infix and mixfix term operators in increasing order:

`=`	*nested equation*
orelse	*boolean disjunction*
andthen	*boolean conjunction*
`== \= < =< > >=`	*nested comparisons*
`:: \:: ::: \:::`	*nested finite domain constraints*
`|`	*cons list*
`#`	*mixfix tuple construction*
`+ -`	*additive operators*
`* /` **div mod**	*multiplicative operators*
`.`	*field selection*

"Having higher precedence" means "binding tighter"; e.g. the term `c # X . g = Y` is parsed like `(c # (X . g)) = Y`. The prefix operators `@` and `~` have higher precedence than any of the above. The infix operators `=` **orelse andthen** `|` are right-associative (e.g. `X|Y|Z` is parsed like `X|(Y|Z)`). The infix operators `+ - * /` **div mod** `.`

`::` `\::` `:::` `\:::` are left-associative (e.g. `X+Y+Z` is parsed like `(X+Y)+Z`). The mixfix tuple constructor `#` is not associative.

3.7 Grammatical Errors

Any sequence of tokens that is not member of the language described by the above grammar, is considered erroneous. For any member of the language there exists a unique parse tree.

4 Programs

In this section, we describe how a sequence of tokens that is an element of the language defined in Section 3 is transformed to a program.

At this point, we leave the realm of syntax, containing entities like characters, words and tokens. From now on, we deal with semantic constructs that we call programs. A program is obtained from a sequence of tokens by taking its parse tree according to the grammar in Section 3, and

- replacing atom tokens of the form $(\!|atom, s|\!)$ by atoms that correspond to the character sequence s (see [3]),
- replacing variable tokens by variables, such that if two variable tokens $(\!|variable, s_1|\!)$ and $(\!|variable, s_2|\!)$ have different character sequences s_1 and s_2, then the corresponding variables are different,
- replacing label tokens of the form $(\!|atomlabel, s|\!)$ by atoms that correspond to the character sequence s,
- replacing label tokens of the form $(\!|variablelabel, s|\!)$ by variables analogous to variable tokens,
- replacing integer tokens of the form $(\!|int, i|\!)$ by the integer i,
- replacing float tokens of the form $(\!|float, f|\!)$ by the float f,
- dropping parentheses `()`.

Variables whose syntactic representation begins and ends with ` are called *system variables*. Programs that use system variables are called *system programs*. An implementation may restrict the use of system programs.

To describe sets of programs, we use a grammar obtained from the one we used for sets of token sequences, except that

- the symbol $(\!|atom|\!)$ is replaced by a, denoting atoms of the Oz Universe,
- the symbol $(\!|variable|\!)$ is replaced by x, denoting variables,
- the symbol $(\!|atomlabel|\!)$ is replaced by a, denoting atoms,
- the symbol $(\!|variablelabel|\!)$ is replaced by x, denoting variables,
- the symbol $(\!|int|\!)$ is replaced by i, denoting integers of the Oz Universe,
- the symbol $(\!|float|\!)$ is replaced by f, denoting floating point numbers of the Oz Universe,
- parentheses `()` do not occur,

- a new form of expression is added:

$$E \quad ::= \quad \wr E \int_x^x \quad \textit{state threading}$$

5 Standardized Programs

In this section, we describe how the resulting programs are rewritten to standardized programs. Standardized programs are syntactically much simpler than programs. The following constructs for programs are not allowed in standardized programs:

- program declaration
- (nested) comparison
- (nested) finite domain comparison and constraint
- (nested) function definition
- case conditional
- (nested) thread, object and class definition
- external expression
- cons and fixed length list
- field selection and arithmetic

Furthermore, only a restricted form of declaration, if conditionals and disjunctions is allowed.

More formally, standardized programs are members of the language described by the following context-free grammar:

Programs

$\mathcal{E}$	::=	$\bullet$	*program marker*
	|	$E\ \mathcal{E}$	*composition*
	|	**local** x^* **in** $\mathcal{E}$ **end**	*declaration*

Expressions

E	::=	**true** | **false**	*success and failure*
	|	t = t	*equation*
	|	**proc** {t x^*} E **end**	*procedure definition*
	|	E E	*composition*
	|	**local** x^* **in** E **end**	*declaration*
	|	{t t^*}	*application*
	|	**if** C ([] C)* **else** E **fi**	*conditional*
	|	**or** D ([] D)$^+$ **ro**	*disjunction*
	|	**OR** D ([] D)$^+$ **RO**	*non-distributing disjunction*
	|	<< t t >>	*method application*
	|	t <- t	*assignment*
	|	$\wr E \int_x^x$	*state threading*

Terms

r, s, t	::=	v	*variable*
	|	a	*atom*
	|	l(t^* (t <= t)* [...])	*tuple construction or matching*
	|	l((t:t [<= t])* [...])	*record construction or matching*
	|	i | f	*integer or float*
	|	t = t	*nested equation*
	|	**proc** {$ x^*} E **end**	*nested procedure definition*
	|	**local** x^* **in** b **end**	*nested declaration*
	|	{t t^*}	*nested application*
	|	_	*wildcard*
	|	$	*nesting marker*
	|	**if** CT ([] CT)* **else** b **fi**	*nested conditional*
	|	**or** DT ([] DT)$^+$ **ro**	*nested disjunction*
	|	**OR** DT ([] DT)$^+$ **RO**	*nested non-distr. disjunction*
	|	@ t	*attribute access*
	|	<< t t >>	*nested method application*
b	::=	[E] t	*term with optional expression*
l	::=	a | v	*label*
v	::=	[!] x	*variable*

Clauses

$$C \quad ::= \quad x^* \ \textbf{in}\ E\ \textbf{then}\ E \quad \textit{conditional clause}$$
$$D \quad ::= \quad x^* \ \textbf{in}\ E\ \textbf{then}\ E \quad \textit{disjunctive clause}$$

Nested Clauses

$$CT \quad ::= \quad x^* \ \textbf{in}\ E\ \textbf{then}\ b \quad \textit{conditional clause}$$
$$DT \quad ::= \quad x^* \ \textbf{in}\ E\ \textbf{then}\ b \quad \textit{disjunctive clause}$$

We proceed in three steps, each step containing rewrite rules that must be applied to subtrees of the given program until none is applicable or an error is found.

Finite Domains (Section 5.1). Finite domain comparisons and constraints are eliminated.

Abbreviations (Section 5.2). Common abbreviations like function definition, case conditionals, and threads are eliminated.

Extended Declaration (Section 5.3). Extended declaration of variables is reduced to Kernel Oz.

Some rewrite rules introduce fresh variables for which no capturing can happen. These variables are indicated in this chapter by using typewriter font like `X`. Since some syntactic constructs can be generated from different non-terminals of the context-free grammar, we give the non-terminals at the beginning of the rule, when we want to limit the rule's applicability.

5.1 Finite Domains

t ::= t_1 $\langle FDcompare \rangle$ t_2 $\Longrightarrow$ {`$\langle FDcompare \rangle$` $[\![t_1]\!]$ $[\![t_2]\!]$}

where $\langle FDcompare \rangle \in \{$`<:`, `=<:`, `>:`, `>=:`, `=:`, `\=:`$\}$.

$$\begin{aligned}
[\![s\ \langle FDinfix \rangle\ t]\!] &= \texttt{`FD}\langle FDinfix \rangle\texttt{`(}[\![s]\!]\ [\![t]\!]\texttt{)} \\
[\![\texttt{~}t]\!] &= \texttt{`FD~`(}[\![t]\!]\texttt{)} \\
[\![i]\!] &= \texttt{`FDint`(}i\texttt{)} \\
&\text{otherwise:} \\
[\![t]\!] &= \texttt{`FDother`(}t\texttt{)}
\end{aligned}$$

where $\langle FDinfix \rangle \in \{$`+`, `-`, `*`$\}$.

Program 5.1 Comparison

```
proc {`==` X Y} if X =  Y then true   else False fi end
proc {`=<` X Y} /* as defined in Kernel Oz [3] */   end
proc {`\=` X Y} if X =  Y then false  else true  fi end
proc {`<`  X Y} if X =< Y then X \= Y else false fi end
proc {`>`  X Y} Y <  X end
proc {`>=` X Y} Y =< X end

fun {`==B` X Y} if X == Y then True else False fi end
fun {`\=B` X Y} if X \= Y then True else False fi end
fun {`<B`  X Y} if X <  Y then True else False fi end
fun {`=<B` X Y} if X =< Y then True else False fi end
fun {`>B`  X Y} if X >  Y then True else False fi end
fun {`>=B` X Y} if X >= Y then True else False fi end
```

For example, the expression `X + {P Y} * 10 =<: Z` rewrites to

```
{`FD=<:`
 `FD+`(`FDother`(X) `FD*`(`FDother`({P Y}) `FDint`(10)))
 `FDother`(Z)}
```

The variables `` `FD+` ``, `` `FD-` ``, `` `FD*` ``, `` `FD~` ``, `` `FDint` ``, and `` `FDother` `` are declared in the notational prelude (see Section 7) and bound to names. The variables `` `<:` ``, `` `=<:` ``, `` `>:` ``, `` `>=:` ``, `` `=:` ``, and `` `\\=:` `` are declared in the notational prelude. Their definition is given in [2].

E, t ::= t_1 ⟨*fd*⟩ t_2 $\Longrightarrow$ {`` `⟨*fd*⟩` `` t_2 t_1}

where ⟨*fd*⟩ $\in$ {`::`, `:::`, `\::`, `\:::`}.

The variables `` `::` ``, `` `\\::` ``, `` `:::` ``, `` `\\:::` `` are declared in the notational prelude. Their definition is given in [2].

5.2 Abbreviations

Comparison

In the following rules, ⟨*compare*⟩ stands for one of the symbols `==`,`\=`,`<`,`=<`,`>` and `>=`.

E ::= t_1 ⟨*compare*⟩ t_2 $\Longrightarrow$ {`` `⟨*compare*⟩` `` t_1 t_2}

The procedure `` `=<` `` is defined in [3]. The procedures `` `==` ``,`` `\\=` ``,`` `<` ``,`` `>` ``,`` `>=` `` are given in Program 5.1.

t ::= t_1 ⟨*compare*⟩ t_2 $\Longrightarrow$ {`` `⟨*compare*⟩B` `` t_1 t_2}

The procedures `` `==B` ``, `` `\\=B` ``,`` `<B` ``,`` `=<B` ``,`` `>B` ``,`` `>=B` `` are defined in Program 5.1.

Procedure Definition

E, t ::= **proc** {t $t_1 \cdots t_n$}
  E
end

$\Longrightarrow$

proc {t X$_1 \cdots$X$_n$}
  if !X$_1$ = t_1 $\cdots$!X$_n$ = t_n **in true**
  then E
  fi
end

if $t_1 \cdots t_n$ are not distinct variables.

Function Definition

E, t ::= **fun** {t^+} b **end**

$\Longrightarrow$

proc {t^+ Y} Y = b **end**

Declaration

$\mathcal{E}$ ::= **declare** E •

$\Longrightarrow$

local E **in** • **end**

$\mathcal{E}$ ::= **declare** E **in** $\mathcal{E}$

$\Longrightarrow$

local E **in** $\mathcal{E}$ **end**

B ::= E_1 **in** E_2

$\Longrightarrow$

local E_1 **in** E_2 **end**

b ::= E_1 **in** [E_2] t

$\Longrightarrow$

local E_1 **in** [E_2] t **end**

If Conditional and Disjunction

E ::= **if** CL_1
elseif CL_2
$\vdots$
fi

$\Longrightarrow$

if CL_1
else
  if CL_2
  $\vdots$
  fi
fi

t ::= **if** CTL_1
elseif CTL_2
$\vdots$
fi

$\Longrightarrow$

if CTL_1
else
  if CTL_2
  $\vdots$
  fi
fi

E ::= **if** C^* **fi**

$\Longrightarrow$

if C^* **else false fi**

t ::= **if** CT^* **fi**

$\Longrightarrow$

if CT^* **else false** _ **fi**

C, D ::= E **[then** E**]** $\Longrightarrow$ **in** E **[then** E**]**

CT, DT::= E **then [**E**]** t $\Longrightarrow$ **in** E **then [**E**]** t

D ::= E_1 **in** E_2 $\Longrightarrow$ E_1 **in** E_2 **then true**

Case Conditional

E, t ::= **case** t $\cdots$ **end** $\Longrightarrow$ **local X in X** $= t$ **case X** $\cdots$ **end**

if t is no variable.

E ::= **case** EL_1 $\Longrightarrow$ **case** EL_1
elsecase EL_2 **else**
$\vdots$ **case** EL_2
end $\vdots$
end
end

t ::= **case** ETL_1 $\Longrightarrow$ **case** ETL_1
elsecase ETL_2 **else**
$\vdots$ **case** ETL_2
end $\vdots$
end
end

E ::= **case** x **then** E **end** $\Longrightarrow$ **case** x **of** !`true` **then** E **end**

t ::= **case** x **then** t **end** $\Longrightarrow$ **case** x **of** !`true` **then** t **end**

E ::= **case** x **then** E_1 $\Longrightarrow$ **case** x **of** !`true` **then** E_1
else E_2 **end** **elseof** !`false` **then** E_2 **end**

t ::= **case** x **then** t_1 $\Longrightarrow$ **case** x **of** !`true` **then** t_1
else t_2 **end** **elseof** !`false` **then** t_2 **end**

The variable `true` is declared and bound in the notational prelude (see Section 7).

E ::= **case** x $\Longrightarrow$ **case** x
of FL_1 **of** FL_1
elseof FL_2 **else**
$\vdots$ **case** x **of** FL_2
end $\vdots$
end
end

```
t ::=   case x                    ⟹   case x
        of FTL1                        of FTL1
        elseof FTL2                    else
        ⋮                                 case x of FTL2
        end                               ⋮
                                          end
                                       end

E ::=   case x                    ⟹   if !x = t1 in true then E1
        of t1 then E1                  [] ⋮
        [] ⋮                           [] !x = tn in true then En
        [] tn then En                  [else E]
        [else E]                       fi
        end

t ::=   case x                    ⟹   if !x = t1 in true then b1
        of t1 then b1                  [] ⋮
        [] ⋮                           [] !x = tn in true then bn
        [] tn then bn                  [else b]
        [else b]                       fi
        end
```

Thread Creation

```
t ::=   thread [E1 in] [E2] t     ⟹   local X in
        end                            thread [E1 in] [E2] X = t end
                                       X
                                    end

E ::=   thread E end              ⟹   local X in
                                       if X = 'go'
                                       then {`adjustPriority`} E
                                       fi
                                       X = 'go'
                                    end
```

The variable `` `adjustPriority` `` is defined in the notational prelude (see Section 7) as shown in Program 5.2.

Negation

```
E ::=   not E end                 ⟹   if E then false else true fi

E ::=   not E1 in E2 end          ⟹   if local E1 in E2 end then false
                                       else true fi
```

Program 5.2 Thread Creation

```
proc {`adjustPriority`}
   case {Thread.getPriority} > Thread.default
   then {Thread.setPriority Thread.default}
   else true
   end
end
```

Class and Object Creation

The first rule for class and object creation rewrites class and objects creation expressions into object creation terms:

t ::= **(create | class)** t $\cdots$ **end** $\Longrightarrow$ t = **(create | class)** $ $\cdots$ **end**

In the following rule, the letters o, a, b, f, g, w, m and p all denote terms.

t ::= **(create | class)** [$]
from $o_1 \cdots o_h$
attr $a_1[:b_1] \cdots a_k[:b_k]$
feat $f_1[:g_1] \cdots f_r[:g_r]$
with w
meth m_1 F_1 **end**
$\vdots$
meth m_n F_n **end**
end

$\Longrightarrow$

```
local x_1 = {`ooPrivate`}
      ⋮
      x_q = {`ooPrivate`}
in
   {`(create | class)`
    [o_1 ··· o_h]
    attributes(a_i1:b_i1 ··· a_is:b_is)
    [a_c1 ··· a_ct]
    features(f_j1:g_j1 ··· f_ju:g_ju)
    [f_d1 ··· f_dt]
    [w]
    methods(l_1:p_1 ··· l_n:p_n)}
end
```

if all m_i are of the form **(**v **|** $l(\cdots)$**)**[= t]

The terms p_i are defined as follows:

```
proc {$ I m'_i `ooSelf` O}
   ⟅F_i⟆_0^I
end
```

The term m'_i is `!`x if m_i is of the form x, `!`x=t if m_i is of the form x=t, and m_i otherwise.

The variable or atom l_i is v if m_i is of the form v or v=t, and l if m_i is of the form $l(\cdots)$ or $l(\cdots)$=t.

If **from** $o_1 \cdots o_h$ is omitted, the empty list `nil` is used instead of [$o_1 \cdots o_h$].

The indices i_z are the ones for which $: b_{i_z}$ is given, and the indices c_z are the ones for which no $: b_{i_z}$ is given. The indices j_z are the ones for which $: g_{j_z}$ is given, and the indices d_z are the ones for which no $: g_{j_z}$ is given.

In class creation, the keyword **`with`** is not allowed. If **`with`** w is omitted in object creation, the atom `nil` is used instead of w.

The variables $x_1, \cdots, x_q$ are extracted as follows:

$$\{x_1, \cdots, x_q\} = \{x \in \{a_1, \cdots, a_k, f_1, \cdots, f_r, l_1, \cdots, l_n\} \mid x \text{ is non-escaped variable}\}$$

The rewriting of class and object creation uses the system procedures `` `class` `` and `` `create` `` defined in the notational prelude as shown in Programs 5.3 and 5.4. The function `Inherit` in Program 5.3 uses the procedure `Class.precedence` described in [1] that defines the mechanism for multiple inheritance. The procedure `` `create` `` calls the procedure `Inheritable.new`, given in [1] along with the standard object protocol of Oz.

Method Application

E, t ::= `<<` t `>>` $\Longrightarrow$ `<<` **`self`** t `>>`

External Expression

E ::= **`extern`** E **`end`** $\Longrightarrow$ **`{self`** `` `ooExtern` `` **`(proc {$ I O}`** $\{E\}_{\mathtt{O}}^{\mathtt{I}}$ **`end)}`**

The variable `` `ooExtern` `` is declared in the notational prelude (see Section 7) and bound to a name.

Cons List

t ::= s `|` t $\Longrightarrow$ $'|'(s\ t)$

Fixed Length List

t ::= `[`$t_1 \cdots t_n$`]` $\Longrightarrow$ t_1 `|` $\cdots$ `|` t_n `|` `nil`

Mixfix Tuple Construction

t ::= t_1 `#` $\cdots$ `#` t_n $\Longrightarrow$ $'\#'(t_1 \cdots t_n)$

Selection

t ::= t_1 `.` t_2 $\Longrightarrow$ `{`` `.` `` t_1 t_2`}`

Program 5.3 Class Creation

```
local
   fun {Inherit C}
      {FoldL
       {Class.precedence C}
       fun {$ Attr#FreeAttr#Feat#FreeFeat#Methods Canc}
          CancAttr=Canc.`ooNewAttr`
          CancFeat=Canc.`ooNewFeat`
       in
           {Adjoin Attr CancAttr}
          #{Append FreeAttr
            {FilterB Canc.`ooNewFreeAttr`
             fun {$ X} {Not {HasSubtreeAtB CancAttr X}} end}}
          #{Adjoin Feat Canc.`ooNewFeat`}
          #{Append FreeFeat
            {FilterB Canc.`ooNewFreeFeat`
             fun {$ X} {Not {HasSubtreeAtB CancFeat X}} end}}
          #{Adjoin Methods Canc.`ooNewMethods`}
       end
       attributes#nil#features#nil#methods
      }
   end
in
   proc {`class` FromList NewAttr NewFreeAttr NewFeat NewFreeFeat
                 NewMethods ?C}
      AllAttr AllFreeAttr AllFeat AllFreeFeat AllMethods
   in
      C = ChunkLabel(`ooAllAttr`       : AllAttr
                     `ooNewAttr`       : NewAttr
                     `ooAllFreeAttr`   : AllFreeAttr
                     `ooNewFreeAttr`   : NewFreeAttr
                     `ooAllFeat`       : AllFeat
                     `ooNewFeat`       : NewFeat
                     `ooAllFreeFeat`   : AllFreeFeat
                     `ooNewFreeFeat`   : NewFreeFeat
                     `ooAllMethods`    : AllMethods
                     `ooNewMethods`    : NewMethods
                     `ooParents`       : {Map FromList Inheritable.´class´}
                     `ooClassId`       : {NewName})
      AllAttr#AllFreeAttr#AllFeat#AllFreeFeat#AllMethods = {Inherit C}
   end
end
```

Program 5.4 Object Creation

```
proc {`create` FromList NewAttr NewFreeAttr NewFeat NewFreeFeat
               WithMessage NewMethods ?Self}
   Class={`class` FromList NewAttr NewFreeAttr NewFeat NewFreeFeat
                  NewMethods}
in
   {Inheritable.new Class WithMessage ?Self}
end
```

Arithmetic

t ::= `~` t $\Longrightarrow$ {`` `~` `` t}

t ::= t_1 ⟨*infix*⟩ t_2 $\Longrightarrow$ {`` `⟨infix⟩` `` t_1 t_2}

where ⟨*infix*⟩ ∈ {`+`, `-`, `*`, `/`, **`div`**, **`mod`**}.

The procedures `` `+` ``,`` `-` ``,`` `*` ``,`` `/` `` and defined in [3]. The procedures `` `~` ``, `` `div` ``,`` `mod` `` are declared in the notational prelude (see Section 7) and defined in [1].

Boolean Operators

t ::= t_1 **`andthen`** t_2 $\Longrightarrow$ **`case`** t_1 **`then`** t_2 **`else`** `` `false` `` **`end`**

The variable `` `false` `` is declared and bound in the notational prelude (see Section 7).

t ::= t_1 **`orelse`** t_2 $\Longrightarrow$ **`case`** t_1 **`then`** `` `true` `` **`else`** t_2 **`end`**

The variable `` `true` `` is declared and bound in the notational prelude (see Section 7).

Self

t ::= **`self`** $\Longrightarrow$ `!` `` `ooSelf` ``

5.3 Extended Variable Declarations

The pattern variables of an expression E, denoted $\mathcal{PV}(E)$, and of a term t, denoted $\mathcal{PV}(t)$ are defined as follows:

E	$\mathcal{PV}(E)$
$E_1\ E_2$	$\mathcal{PV}(E_1) \cup \mathcal{PV}(E_2)$
s `=` t	$\mathcal{PV}(s) \cup \mathcal{PV}(t)$
proc `{`$s\ t^*$`}` E **end**	$\mathcal{PV}(s)$
local E_1 **in** E_2 **end**	$\mathcal{PV}(E_2) - \mathcal{PV}(E_1)$
x	$\{x\}$
otherwise	$\emptyset$

t	$\mathcal{PV}(t)$
x	$\{x\}$
l`(`s_1`[<=`t_1`]`$\cdots s_n$`[<=`t_n`][...])`	$\bigcup_i \mathcal{PV}(s_i)$
l`(`l_1`:`s_1`[<=`t_1`]`$\cdots l_n$`:`s_n`[<=`t_n`][...)]`	$\bigcup_i \mathcal{PV}(s_i)$
local E_1 **in** `[`E_2`]` t **end**	$(\mathcal{PV}(E_2) \cup \mathcal{PV}(t)) - \mathcal{PV}(E_1)$
otherwise	$\emptyset$

Declaration

$\mathcal{E}, E, t$ ::= **local** E **in** $\vdots$ **end** $\Longrightarrow$ **local** $\mathcal{PV}(E)$ **in** E $\vdots$ **end**

Variables as expressions in E are discarded between **in** and **end**.

Clauses

C,D, CT,DT ::= E_1 **in** E_2 **then** $\cdots$ $\Longrightarrow$ $\mathcal{PV}(E_1)$ **in** $E_1\ E_2$ **then** $\cdots$

Variables as expressions in E_1 are discarded between **in** and **then**.

5.4 Rewrite Error

The following errors may occur:

- **Methods.** In object and class creation, one of the terms m after **meth** is not of the form `(`v `|` l`(`$\cdots$`)`**`[=`** t**`]`**.
- **Variables.** A variable occurs as expression and is not a pattern variable.

6 Kernel Programs

In this section, we describe how the resulting standardized programs are rewritten to Kernel programs. Kernel programs are standardized programs in which

- no nesting occurs, i.e. terms are either variables, atoms, integers or floats, and
- no state threading occurs.

More formally, kernel programs are members of the language described by the following context-free grammar:

Programs

$\mathcal{E}$	::=	$\bullet$	*program marker*
	|	$E\ \mathcal{E}$	*composition*
	|	**local** x^* **in** $\mathcal{E}$ **end**	*declaration*

Expressions

E	::=	**true** | **false** | x **=** t	constraints
	|	**proc** {$x\ x^*$} E **end**	procedure definition
	|	{$x\ x^*$}	procedure application
	|	$E\ E$	composition
	|	**local** x^* **in** E **end**	variable declaration
	|	**if** C **(**[] C**)*** **else** E **fi**	conditional
	|	**or** C **(**[] C**)**$^+$ **ro**	disjunction
	|	**OR** C **(**[] C**)**$^+$ **RO**	
C	::=	x^* **in** E **then** E	clause
t	::=	x | a | i | f	term

We proceed in two steps, each step containing rewrite rules on trees that must be applied to subtrees of the given program until none is applicable or an error is found.

Nesting (Section 6.1). Nested constructs are eliminated.

State threading (Section 6.2). State threadings as introduced by object and class creation are eliminated.

6.1 Nesting

Nesting is eliminated by reducing nested constructs to equations (unnesting) and by further rewriting these equations.

Unnesting

Terms in equations, applications, procedure definitions, assignment, attribute access, and method applications are unnested such that only variables occur at designated positions.

E ::= `s = t` $\Longrightarrow$

```
local X in X = s X = t end
```

if s is no variable.

E ::= `proc {`$t\ y^*$`} `E` end` $\Longrightarrow$

```
local X in
   X = t proc {X y*} E end
end
```

if t is no variable.

E ::= `{`$t_1 \cdots t_n$`}` $\Longrightarrow$

```
local X1 ··· Xn in
   X1 = t1 ··· Xn = tn
   {X1 ··· Xn}
end
```

if one of t_i is no variable.

E ::= `s <- t` $\Longrightarrow$

```
local X Y in
   X = s Y = t X <- Y
end
```

if one of s or t is no variable.

E ::= `x = @ t` $\Longrightarrow$

```
local Y in Y = t x = @ Y end
```

if t is no variable.

E ::= `<< s t >>` $\Longrightarrow$

```
local X Y in
   X = s Y = t << X Y >>
end
```

if one of s or t is no variable.

Terms in tuple and record construction and matching are unnested such that only variables occur as subterms, features and defaults.

$$
\begin{array}{lll}
E ::= & x = l(s_1 \cdots s_k & \Longrightarrow \\
& \quad [s_{k+1} <= t_{k+1} & \\
& \quad \vdots & \\
& \quad s_n <= t_n] [\ldots]) &
\end{array}
$$

```
local X1 ··· Xk
      [Xk+1 Yk+1 ··· Xn Yn] in
   x = l(X1 ··· Xk
         [Xk+1 <= Yk+1
          ⋮
          Xn <= Yn] [...])
   X1 = s1 ··· Xk = sk
   [Xk+1 = sk+1 Yk+1 = tk+1
   ⋮
   Xn = sn Yn = tn]
end
```

if one of s_i or t_i is no variable.

$$
\begin{array}{lll}
E ::= & x = l(r_1{:}s_1 \cdots r_k{:}s_k & \Longrightarrow \\
& \quad [r_{k+1}{:}s_{k+1} <= t_{k+1} & \\
& \quad \vdots & \\
& \quad r_n{:}s_n <= t_n] & \\
& \quad [\ldots]) &
\end{array}
$$

```
local X1 Y1 ...Xk Yk
      [Xk+1 Yk+1 Zk+1
       ⋮
       Xn Yn Zn] in
   X1 = r1 ··· Xk = rk
   [Xk+1 = rk+1 ··· Xn = rn]
   x = l(X1:Y1 ··· Xk:Yk
         [Xk+1:Yk+1 <= Zk+1
          ⋮
          Xn:Yn <= Zn] [...])
   Y1 = s1 ··· Yk = sk
   [Yk+1 = sk+1 Zk+1 = tk+1
   ⋮
   Yn = sn Zn = tn]
end
```

if one of r_i, s_i or t_i is no variable.

Equations

Equations of the form x `=` t are further rewritten using the following rules:

E ::= x `=` `_` $\Longrightarrow$ **`true`**

E ::= x `=` `!`y $\Longrightarrow$ x `=` y

E ::= x `=` s `=` t $\Longrightarrow$ x `=` s $\quad$ x `=` t

E ::= x `=` **`proc`** `{$` $y_1 \cdots y_n$`}` E **`end`** $\Longrightarrow$ **`proc`** `{`x $y_1 \cdots y_n$`}` E **`end`**

E ::= x `=` **`local`** y^* **`in`** `[`E`]` t **`end`** $\Longrightarrow$ **`local`** `Z` **`in`** `Z` `=` x **`local`** y^* **`in`** `[`E`]` `Z` `=` t **`end`** **`end`**

E ::= x `=` `{`$t_1 \cdots t_n$`}` $\Longrightarrow$ `{`$t_1 \cdots t_n$ x`}`

if no `$` occurs at a pattern position[1] in $t_1 \cdots t_n$.

E ::= x `=` `{`$t_1 \cdots t_k \cdots t_n$`}` $\Longrightarrow$ `{`$t_1 \cdots t_k[x]_p \cdots t_n$`}`

if `$` occurs in t_k at a pattern position[1] p
and no other `$` occurs at a pattern position in $t_1 \cdots t_n$.

The following three rules use the function $[\![\cdot]\!]^x$ which is defined as:

$$[\![y^* \texttt{ in } E_1 \texttt{ then } [E_2]\ t]\!]^x \quad = \quad y^* \texttt{ in } E_1 \texttt{ then } [E_2]\ x = t$$

E ::= x `=` **`if`** CT_1 `[]` $\cdots$ `[]` CT_n **`else`** `[`E`]` t **`fi`** $\Longrightarrow$ **`local`** `Z` **`in`** x `=` `Z` **`if`** $[\![CT_1]\!]^{\texttt{Z}}$ `[]` $\cdots$ `[]` $[\![CT_n]\!]^{\texttt{Z}}$ **`else`** `[`E`]` `Z` `=` t **`fi`** **`end`**

Program 6.1 Procedures for Construction

```
fun {`matchLiteral` X={IsLiteral}} X end
`extendTuple` = ExtendTuple
`adjoinAt` = AdjoinAt
```

E ::= `x =` DT_1 `[]` $\cdots$ `[]` DT_n `ro` $\Longrightarrow$ `local Z in` x `= Z` `or` $[\![DT_1]\!]^{\texttt{Z}}$ `[]` $\cdots$ `[]` $[\![DT_n]\!]^{\texttt{Z}}$ `ro` `end`

E ::= `x =` `OR` DT_1 `[]` $\cdots$ `[]` DT_n `RO` $\Longrightarrow$ `local Z in` x `= Z` `OR` $[\![DT_1]\!]^{\texttt{Z}}$ `[]` $\cdots$ `[]` $[\![DT_n]\!]^{\texttt{Z}}$ `RO` `end`

E ::= `x = <<` s t `>>` $\Longrightarrow$ `<<` s $t[x]_p$ `>>`

if `$` occurs in t at a pattern position[1] p,
and no other `$` occurs at a pattern position in t.

Equations where the right hand side consists of tuple and record construction are rewritten as follows. The rules for tuple and record construction use the procedures `` `matchLiteral` ``, `` `extendTuple` ``, and `` `adjoinAt` `` that are defined in the notational prelude (see Section 7) as shown in Program 6.2.

E ::= `x =` l`()` $\Longrightarrow$ `{`` `matchLiteral` `` `}` l x`}`

E ::= `x =` l`(`$x_1 \cdots x_n$`)` $\Longrightarrow$ `{`` `extendTuple` `` l`(`$x_1 \cdots x_{n-1}$`)` x_n x`}`

where $n > 0$.

E ::= `x =` l`(`x_1`:`$y_1 \cdots x_n$`:`y_n`)` $\Longrightarrow$ `{`` `adjoinAt` ``
l`(`x_1`:`$y_1 \cdots x_{n-1}$`:`y_{n-1}`)` x_n y_n x`}`

where $n > 0$.

Equations where the right hand side consists of tuple and record matching are rewritten as follows. The rules for tuple and record matching use the procedures

Program 6.2 Procedures for Matching

```
%%% NoNumbers
`...` = Label

%%% Tuples
proc {`tuple...` L Xs T={IsTuple}}
   {Label T L}
   {List.forAllInd Xs proc {$ I X} X=T.I end}
end
proc {`tuple<=` L Xs Ys T={IsTuple}}
   {Label T L}
   K={List.foldL Xs fun {$ I X} T.I=X I+1 end 1}
   N={List.foldL Ys
      fun {$ I Y#D}
         Y=if W in T.I=W then W else D fi
         I+1
      end K}
in
   {Width T} =< N
end
proc {`tuple<=...` L Xs Ys T={IsTuple}}
   {Label T L}
   K={List.foldL Xs fun {$ I X} T.I=X I+1 end 1}
in
   {List.foldL Ys
    fun {$ I Y#D}
       Y=if W in T.I=W then W else D fi
       I+1
    end K _}
end

%%% Records
`adjoin` = Adjoin
proc {`record...` L Xs R={IsRecord}}
   {Label R L}
   {List.forAll Xs proc {$ X} !X=A#B in R.A=B end}
end
```

`` `...` ``, `` `tuple...` ``, `` `tuple<=` ``, `` `tuple<=...` ``, `` `adjoin` ``, and `` `record...` `` that are defined in the notational prelude (see Section 7) as shown in Program 6.2.

E ::= x `=` l`(...)` $\Longrightarrow$ `{`` `...` `` l x`}`

E ::= x `=` l`(`$x_1 \cdots x_n$`...)` $\Longrightarrow$ `{`` `tuple...` `` l `[`$x_1 \cdots x_n$`]` x`}`

E ::= x `=` l`(`$x_1 \cdots x_k$
x_{k+1} `<=` y_{k+1}
$\vdots$
x_n `<=` y_n`)` $\Longrightarrow$ `{`` `tuple<=` `` l `[`$x_1 \cdots x_k$`]`
`[`x_{k+1}`#`$y_{k+1} \cdots x_n$`#`y_n`]` x`}`

E ::= x `=` l`(`$x_1 \cdots x_k$
x_{k+1} `<=` y_{k+1}
$\vdots$
x_n `<=` y_n`...)` $\Longrightarrow$ `{`` `tuple<=...` `` l `[`$x_1 \cdots x_k$`]`
`[`x_{k+1}`#`$y_{k+1} \cdots x_n$`#`y_n`]` x`}`

E ::= x `=`
l`(`x_1`:`$y_1 \cdots x_n$`:`y_n`...)` $\Longrightarrow$ `{`` `record...` ``
l `[`x_1 `#` $y_1 \cdots x_n$ `#` y_n`]` x`}`

E ::= x `=` l`(`x_1`:`$y_1 \cdots x_k$`:`y_k
x_{k+1}`:`y_{k+1} `<=` z_{k+1}
$\vdots$
x_n`:`y_n `<=` z_n`[...]`) $\Longrightarrow$ `{`` `adjoin` `` l`(`x_{k+1}`:`$z_{k+1} \cdots x_n$`:`z_n`)` x
l`(`x_1`:`$y_1 \cdots x_n$`:`y_n`[...])}`

6.2 State Threading

State-free expressions are expressions that contain no attribute access, assignment and method application outside of state threading.

[1] A position p in a term t is a pattern position if the variable x in $t[x]_p$ is a pattern variable. $t[x]_p$ denotes the result of replacing in t the subtree at position p by x.

Program 6.3 State Threading Procedures

```
proc {`<-` RIn L X ?ROut}
   case {HasSubtreeAtB RIn L}
   then {AdjoinAt RIn L X ROut}
   end
end
proc {`@` RIn L X ROut}
   if Y in Y=RIn.L then ROut=RIn X=Y fi
end
proc {`<<>>` Object RIn={IsRecord} Message Self ?ROut}
   {Object.`ooClass`.`ooAllMethods`.{Label Message}
    RIn Message Self ROut}
end
```

$$\lbag E \rbag_z^y \implies y = z\ E \quad \text{if } E \text{ is state-free expression}$$

otherwise:

$$\begin{aligned}
\lbag E_1\ E_2 \rbag_z^y &\implies \texttt{local W in } \lbag E_1 \rbag_{\texttt{W}}^y\ \lbag E_2 \rbag_z^{\texttt{W}} \texttt{end} \\
\lbag \texttt{local } x^* \texttt{ in } E \texttt{ end} \rbag_z^y &\implies \texttt{local } x^* \texttt{ in } \lbag E \rbag_z^y \texttt{ end} \\
\lbag \texttt{if } C_1 \texttt{ [] } \cdots \texttt{ [] } C_n \texttt{ else } E \texttt{ fi} \rbag_z^y &\implies \texttt{if } \lbag C_1 \rbag_z^y \texttt{ [] } \cdots \texttt{ [] } \lbag C_n \rbag_z^y \texttt{ else } \lbag E \rbag_z^y \texttt{ fi} \\
\lbag \texttt{or } C_1 \texttt{ [] } \cdots \texttt{ [] } C_n \texttt{ ro} \rbag_z^y &\implies \texttt{or } \lbag C_1 \rbag_z^y \texttt{ [] } \cdots \texttt{ [] } \lbag C_n \rbag_z^y \texttt{ ro} \\
\lbag \texttt{OR } C_1 \texttt{ [] } \cdots \texttt{ [] } C_n \texttt{ RO} \rbag_z^y &\implies \texttt{OR } \lbag C_1 \rbag_z^y \texttt{ [] } \cdots \texttt{ [] } \lbag C_n \rbag_z^y \texttt{ RO} \\
\lbag x^* \texttt{ in } E_1 \texttt{ then } E_2 \rbag_z^y &\implies x^* \texttt{ W in } \lbag E_1 \rbag_{\texttt{W}}^y \texttt{ then } \lbag E_2 \rbag_z^{\texttt{W}} \\
\lbag v \texttt{ = @}w \rbag_z^y &\implies \texttt{\{`@` } y\ w\ v\ z\texttt{\}} \\
\lbag v \texttt{ <- } w \rbag_z^y &\implies \texttt{\{`<-` } y\ v\ w\ z\texttt{\}} \\
\lbag \texttt{<< } v\ w \texttt{ >>} \rbag_z^y &\implies \texttt{\{`<<>>` } v\ y\ w\ \texttt{`ooSelf`}\ z\texttt{\}}
\end{aligned}$$

The procedures `` `@` ``, `` `<-` ``, `` `<<>>` `` are shown in Program 6.3.

6.3 Rewrite Errors

The following errors may occur:

- **Using the nesting marker.** In a nested application occurs more than one nesting marker at a pattern position, or in a nested method application occurs

no or more than one nesting marker, or the nesting marker occurs not in a nested application or method application or not in pattern position.

- **Using state outside of state threading.** Attribute access, assignment and method application occur outside of method definitions or inside of procedure and function definitions.

7 Notational Prelude

The notational prelude is given in Program 7.1

Program 7.1 Notational Prelude.

```
declare
   %%% Finite Domains
   `FD+`      = {NewName}
   `FD-`      = {NewName}
   `FD*`      = {NewName}
   `FD~`      = {NewName}
   `FDint`    = {NewName}
   `FDother`  = {NewName}
   `<:`       = FD.´<´
   `=<:`      = FD.´=<´
   `>:`       = FD.´>´
   `>=:`      = FD.´>=´
   `=:`       = FD.´=´
   `\\=:`      = FD.´\\=´
   `::`       = FD.dom
   `\\::`     = FD.coDom
   `:::`      = FD.doms
   `\\:::`    = FD.coDoms

   %%% Comparison  (see Program 5.1)
   `==`   `\\=`   `<`   `>`   `>=`
   `==B`   `\\=B`   `<B`   `=<B`   `>B`   `>=B`

   %%% Case Expression
   `true` = Bool.´true´

   %%% Threads  (see Program 5.2)
   `adjustPriority`

   %%% Object Creation  (see Programs 5.3, 5.4)
   `create` `class`
   `ooPrivate`        = NewName
   `ooAllAttr`        = {NewName}
   `ooNewAttr`        = {NewName}
   `ooNewGenericAttr` = {NewName}
   `ooNewFeat`        = {NewName}
   `ooNewGenericFeat` = {NewName}
   `ooAllMethods`     = {NewName}
   `ooNewMethods`     = {NewName}
   `ooParents`        = {NewName}
```

```
   `ooPrintName`      = {NewName}
   `ooClosed`         = {NewName}
   `ooClassId`        = {NewName}
   `ooCreate`         = {NewName}
   `ooNoMessage`      = {NewName}
   `ooExtern`         = {NewName}

   %%% Arithmetic
   `~`   = Int.´~´
   `div` = Int.´div´
   `mod` = Int.´mod´

   %%% Tuple and Record Construction and Matching (see Programs 6.1, 6.2)
   `matchLiteral`   `extendTuple`   `adjoinAt`
   `...`   `tuple...`   `tuple<=`   `tuple<=...`   `adjoin`   `record...`

   %%% State Threading (see Program 6.3)
   `@` `<-` `<<>>`

   %%% Standard Modules used in Notational Prelude
   Int   Record   Tuple   Bool   List   Object   Inheritable   FD

in
⟨ here go Programs 5.1, 5.2, 5.3, 5.4, 6.1, 6.2, 6.3 ⟩
```

References

[1] Martin Henz, Martin Müller, Christian Schulte, and Jörg Würtz. The Oz standard modules. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[2] Tobias Müller, Konstantin Popow, Christian Schulte, and Jörg Würtz. Constraint programming in Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[3] Gert Smolka. The definition of Kernel Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

List of Oz Programs