The Event-B Mathematical Language

Christophe Métayer (Systerel) Laurent Voisin (Systerel)

March 26, 2009

Contents

1 Introduction **1**

2 Language Lexicon **3**
2.1 Whitespace 3
2.2 Identifiers 3
2.3 Integer Literals 4
2.4 Predicate symbols 5
2.5 Expression symbols 6

3 Language Syntax **8**
3.1 Notation 8
3.2 Predicates 8
3.2.1 A first attempt 8
3.2.2 Associativity of operators 9
3.2.3 Priority of operators 10
3.2.4 Final syntax for predicates 11
3.3 Expressions 12
3.3.1 Some Fine Points 12
3.3.2 A First Attempt 14
3.3.3 Operator Groups 15
3.3.4 Priority of Operator Groups 16
3.3.5 Associativity of operators 16
3.3.6 Final syntax for expressions 20

4 Static Checking **22**
4.1 Abstract Syntax 22
4.2 Legibility 23
4.3 Type Checking 27
4.3.1 Typing Concepts 27
4.3.2 Specification of Type Check 28
4.3.3 Examples 36

5 Dynamic Checking **41**
5.1 Predicate Well-Definedness 41
5.2 Expression Well-Definedness 41

1 Introduction

This document presents the technical aspects of the kernel mathematical language of event-B. Beyond the pure syntax of the language, it also describes its lexical structure and various checks (both static and dynamic) that can be done on formulas on the language.

The main design principle of the language is to have intuitive priorities for operators and to use a minimal set of parenthesis (except when needed to resolve common ambiguities). So, the emphasis is really on having formulas unambiguous and easy to read.

The first chapter describes the lexicon used by the language, then chapter two describes its (concrete) syntax. Chapter three introduces the notion of legible and well-typed formula (static checks). Finally, chapter four gives the well-definedness conditions for a formula (dynamic check).

Revision History

Date	**Contents**
2005/05/31	Initial revision (Rodin Deliverable D7).
2006/05/24	Added min and max unary operators.
2007/10/26	Minor corrections in the text.
2009/03/26	Added missing documentation for pred and succ. Replaced *well-formedness* by *legibility* Version 2 of the language (see below).

Changes in Version 2

Version 2 is an incompatible change of the original mathematical language. This new version has been supported by the Rodin platform since release 1.0.0, while the original language was supported until releases 0.9.x.

The changes introduced in version 2 are:

- Relation set arrows (set of relations, functions, injections, ...) become non-associative. This change was motivated by a discrepancy between this document (where these operators were right-associative) and the current implementation in the Rodin platform (where they were left associative).

- The id, prj_1 and prj_2 operators become generic. Formerly, these operators were unary, taking an argument giving their extent. However, after some experience, it was found out that this was not necessary and rather a legacy from classical B. In event-B, where types can be inferred by the type-checker, there is no need anymore for these parameters. The correspondance between the original mathematical language and version 2 is the following:

Original	**Version 2**
$\mathrm{id}(\mathbf{E})$	$\mathbf{E} \lhd \mathrm{id}$
$\mathrm{prj}_1(\mathbf{E})$	$\mathbf{E} \lhd \mathrm{prj}_1$
$\mathrm{prj}_2(\mathbf{E})$	$\mathbf{E} \lhd \mathrm{prj}_2$

- The new partition operator has been added to the language.

2 Language Lexicon

This chapter describes the lexicon of the mathematical language, that is the way that terminal tokens of the language grammar are built from a stream of characters.

Here, we assume that the input stream is made of Unicode characters, as defined in the Unicode standard 4.0 [4]. As we use only characters of the Basic Multilingual Plane, all characters are designated by their code points, that is an uppercase letter 'U' followed by a plus sign and an integer value (made of four hexadecimal digits). For instance, the classical space character is designated by `U+0020`.

Each token is formed by considering the longest sequence of characters that matches one of the definition below.

2.1 Whitespace

Whitespace characters are used to separate tokens or to improve the legibility of the formula. They are otherwise ignored during lexical analysis.

The whitespace characters of the mathematical language are the Unicode 4.0 space characters:

U+0020 U+00A0 U+1680 U+180E U+2000 U+2001
U+2002 U+2003 U+2004 U+2005 U+2006 U+2007
U+2008 U+2009 U+200A U+200B U+2028 U+2029
U+202F U+205F U+3000

together with the following control characters (these are the same as in the Java Language):

U+0009 U+000A U+000B U+000C U+000D
U+001C U+001D U+001E U+001F

2.2 Identifiers

The identifiers of the mathematical language are defined in the same way as in the Unicode standard [4, par. 5.15]. This definition is not repeated here. Basically, an identifier is a sequence of characters that enjoy some special property, like referring to a letter or a digit.

Some identifiers are reserved for the mathematical language, where a predefined meaning is assigned to them. These reserved keywords are the following identifiers made of ASCII letters and digits:

BOOL FALSE TRUE
bool card dom finite id
inter max min mod partition
pred prj1 prj2 ran succ
union

together with those other identifiers that use non-ASCII characters:

Token	Code points	Token name
$\mathbb{N}$	U+2115	SET OF NATURAL NUMBERS
$\mathbb{N}_1$	U+2115 U+0031	SET OF POSITIVE NUMBERS
$\mathbb{P}$	U+2119	POWERSET
$\mathbb{P}_1$	U+2119 U+0031	SET OF NON-EMPTY SUBSETS
$\mathbb{Z}$	U+2124	SET OF INTEGERS

2.3 Integer Literals

Integer literals consists of a non-empty sequence of ASCII decimal digits:

U+0030 U+0031 U+0032 U+0033 U+0034
U+0035 U+0036 U+0037 U+0038 U+0039

Note: There are two ways to tokenize integer literals: either signed or unsigned. The first case as the advantage that it corresponds to classical usage in mathematics. For instance, the string -1 is thought as representing a number, not a unary minus operator followed by a number. But, as we use the same character to designate both unary and binary minus, this causes problems: the lexical analysis is no longer context-free, but depends on the syntax of the language.

There are basically two solutions to this problem. One, taken in some functional languages in the ML family and in the Z notation, is to use different characters to represent the unary and binary minus operator. However, this comes against mathematical tradition and is thus rejected. The second solution is to consider that integer literals are unsigned. This second solution has been chosen here.

2.4 Predicate symbols

The tokens used in the pure predicate calculus are:

Token	Code point	Token name
(	U+0028	LEFT PARENTHESIS
)	U+0029	RIGHT PARENTHESIS
⇔	U+21D4	LOGICAL EQUIVALENCE
⇒	U+21D2	LOGICAL IMPLICATION
∧	U+2227	LOGICAL AND
∨	U+2228	LOGICAL OR
¬	U+00AC	NOT SIGN
⊤	U+22A4	TRUE PREDICATE
⊥	U+22A5	FALSE PREDICATE
∀	U+2200	FOR ALL
∃	U+2203	THERE EXISTS
,	U+002C	COMMA
·	U+00B7	MIDDLE DOT

The symbolic tokens used to build predicates from expressions are:

Token	Code point	Token name
=	U+003D	EQUALS SIGN
≠	U+2260	NOT EQUAL TO
<	U+003C	LESS-THAN SIGN
≤	U+2264	LESS THAN OR EQUAL TO
>	U+003E	GREATER-THAN SIGN
≥	U+2265	GREATER THAN OR EQUAL TO
∈	U+2208	ELEMENT OF
∉	U+2209	NOT AN ELEMENT OF
⊂	U+2282	SUBSET OF
⊄	U+2284	NOT A SUBSET OF
⊆	U+2286	SUBSET OF OR EQUAL TO
⊈	U+2288	NEITHER A SUBSET OF NOR EQUAL TO

2.5 Expression symbols

The following symbolic tokens are used to build sets of relations (or functions):

Token	Code point	Token name
↔	U+2194	RELATION
$\twoheadleftarrow\!\!\!\rightarrow$	U+E100	TOTAL RELATION
$\leftarrow\!\!\!\twoheadrightarrow$	U+E101	SURJECTIVE RELATION
$\twoheadleftarrow\!\!\!\twoheadrightarrow$	U+E102	TOTAL SURJECTIVE RELATION
⇸	U+21F8	PARTIAL FUNCTION
→	U+2192	TOTAL FUNCTION
⤔	U+2914	PARTIAL INJECTION
↣	U+21A3	TOTAL INJECTION
⤀	U+2900	PARTIAL SURJECTION
↠	U+21A0	TOTAL SURJECTION
⤖	U+2916	BIJECTION

The following symbolic tokens are used for manipulating sets:

Token	Code point	Token name
{	U+007B	LEFT CURLY BRACKET
}	U+007D	RIGHT CURLY BRACKET
↦	U+21A6	MAPLET
∅	U+2205	EMPTY SET
∩	U+2229	INTERSECTION
∪	U+222A	UNION
∖	U+2216	SET MINUS
×	U+00D7	CARTESIAN PRODUCT

The following symbolic tokens are used for manipulating relations and functions:

Token	Code point	Token name
[	U+005B	LEFT SQUARE BRACKET
]	U+005D	RIGHT SQUARE BRACKET
↦	U+21A6	MAPLET
$\lhd\!\!\!+$	U+E103	RELATION OVERRIDING
∘	U+2218	BACKWARD COMPOSITION
;	U+003B	FORWARD COMPOSITION
⊗	U+2297	DIRECT PRODUCT
∥	U+2225	PARALLEL PRODUCT
$^{-1}$	U+223C	TILDE OPERATOR
◁	U+25C1	DOMAIN RESTRICTION
⩤	U+2A64	DOMAIN SUBTRACTION
▷	U+25B7	RANGE RESTRICTION
⩥	U+2A65	RANGE SUBTRACTION

The following symbolic tokens are used in quantified expressions:

Token	Code point	Token name
λ	U+03BB	LAMBDA
⋂	U+22C2	N-ARY INTERSECTION
⋃	U+22C3	N-ARY UNION
∣	U+2223	SUCH THAT

The following symbolic tokens are used in arithmetic expressions:

Token	Code point	Token name
..	U+2025	UPTO OPERATOR
+	U+002B	PLUS SIGN
−	U+2212	MINUS SIGN
∗	U+2217	ASTERISK OPERATOR
÷	U+00F7	DIVISION SIGN
^	U+005E	EXPONENTIATION SIGN

3 Language Syntax

This chapter describes the syntax of the mathematical language, giving the rationale behind the design decisions made.

We first present the notation we use to describe the syntax of the mathematical language. Then, we present the syntax of predicates and of expressions. In each case, we first present a simple ambiguous grammar, then we tackle with associativity and priorities of operators, giving a rationale for each choice made. Finally, we give a complete and non-ambiguous syntax.

3.1 Notation

In this document, we use an Extended Backus-Naur Form (EBNF) to describe syntax. In that notation, non-terminals are surrounded by angle brackets and terminals surrounded by single quotes. The other symbols are meta-symbols:

- Symbol `::=` defines the non-terminal appearing on its left in terms of the syntax on its right.
- Parenthesis `(` and `)` are used for grouping.
- A vertical bar `|` denotes alternation.
- Square brackets `[` and `]` surround an optional part.
- Curly brackets `{` and `}` surround a part that can be repeated zero or more times.

3.2 Predicates

The point here is to define a grammar which is quite similar to the one used commonly when writing mathematical formulae but that should also be non-ambiguous to the (human) reader.

3.2.1 A first attempt

The grammar commonly used for predicates can loosely be defined as follows:

⟨*predicate*⟩ ::= ‘`(`’ ⟨*predicate*⟩ ‘`)`’
| ⟨*predicate*⟩ ‘⇔’ ⟨*predicate*⟩
| ⟨*predicate*⟩ ‘⇒’ ⟨*predicate*⟩
| ⟨*predicate*⟩ ‘∧’ ⟨*predicate*⟩

$$
\begin{array}{lcl}
 & \mid & \langle \textit{predicate} \rangle \text{ `}\lor\text{' } \langle \textit{predicate} \rangle \\
 & \mid & \text{`}\neg\text{' } \langle \textit{predicate} \rangle \\
 & \mid & \text{`}\top\text{'} \\
 & \mid & \text{`}\bot\text{'} \\
 & \mid & \text{`}\forall\text{' } \langle \textit{ident-list} \rangle \text{ `}\cdot\text{' } \langle \textit{predicate} \rangle \\
 & \mid & \text{`}\exists\text{' } \langle \textit{ident-list} \rangle \text{ `}\cdot\text{' } \langle \textit{predicate} \rangle \\
 & \mid & \text{`finite' `}\texttt{(}\text{' } \langle \textit{expression} \rangle \text{ `}\texttt{)}\text{'} \\
 & \mid & \text{`partition' `}\texttt{(}\text{' } \langle \textit{expression-list} \rangle \text{ `}\texttt{)}\text{'} \\
 & \mid & \langle \textit{expression} \rangle \text{ `}=\text{' } \langle \textit{expression} \rangle \\
 & \mid & \langle \textit{expression} \rangle \text{ `}\in\text{' } \langle \textit{expression} \rangle \\
 & \mid & \langle \textit{expression} \rangle \text{ `}\leq\text{' } \langle \textit{expression} \rangle \\
 & \mid & \ldots \\
 & & \\
\langle \textit{ident-list} \rangle & ::= & \langle \textit{ident-list} \rangle \text{ `}\texttt{,}\text{' } \langle \textit{ident} \rangle \\
 & \mid & \langle \textit{ident} \rangle \\
 & & \\
\langle \textit{expression-list} \rangle & ::= & \langle \textit{expression-list} \rangle \text{ `}\texttt{,}\text{' } \langle \textit{expression} \rangle \\
 & \mid & \langle \textit{expression} \rangle
\end{array}
$$

The ellipsis which appears at the end of the ⟨*predicate*⟩ production rule means that there are still more alternatives combining two expressions into a predicate. All those alternatives are not really relevant at this point of the document, but will be fully listed in the final syntax (see section 3.2.4 on page 11).

3.2.2 Associativity of operators

Caution

In this document, we use the term *associativity* with somewhat two different meanings. In a mathematical context, when we write that an operator, say $\circ$, is associative, we mean that it has a special mathematical property, namely that $(x \circ y) \circ z$ has the same value as $x \circ (y \circ z)$. In a syntactical context, we say that an operator is left-associative when formula $x \circ y \circ z$ (without any parenthesis) is parsed as if it would have been written $(x \circ y) \circ z$. To avoid any ambiguity, we will always write *associative in the algebraic sense* when we refer to the first meaning, the bare word *associative* always having the syntactical meaning.

Getting back to our predicate grammar defined above, we see that it is somewhat ambiguous. A first point is that it doesn't specify how one should parse formulae containing twice the same binary predicate operator without any parenthesis such as

$$P \Rightarrow Q \Rightarrow R$$

$$P \land Q \land R$$

To solve that ambiguity, one specifies that each binary operator has a property called *associativity*. The associativities defined for the event-B language are the following:

Operator	**Associativity**
$\Leftrightarrow$	none
$\Rightarrow$	none
$\wedge$	left
$\vee$	left

As a consequence, formula $P \Rightarrow Q \Rightarrow R$ is considered as ill-formed and not part of the event-B language, whereas formula $P \wedge Q \wedge R$ will be parsed as if it actually were written as $(P \wedge Q) \wedge R$.

The rationale for these associativities is quite simple. Operator $\wedge$ is associative in the algebraic sense, so formulae $(P \wedge Q) \wedge R$ and $P \wedge (Q \wedge R)$ have the same meaning. Hence, one can pick up either left or right associativity for this operator. We arbitrarily chose left associativity as it is the most commonly used to our knowledge. The same rationale explains the choice of left associativity for operator $\vee$.

On the other hand, operator $\Rightarrow$ is not associative in the algebraic sense $(P \Rightarrow Q) \Rightarrow R$ is not the same as $P \Rightarrow (Q \Rightarrow R)$ (just suppose that predicates P, Q and R are all $\perp$). As a consequence, we keep it non associative in the language, rather than choosing an arbitrary associativity.

The case of operator $\Leftrightarrow$ is somewhat special. This operator is indeed associative in the algebraic sense. However, mathematicians often write formula $P \Leftrightarrow Q \Leftrightarrow R$ when they actually mean $(P \Leftrightarrow Q) \wedge (Q \Leftrightarrow R)$. Hence, we chose to make that operator non associative in the event-B language to avoid any ambiguity.

Finally, for the operators that build a predicate from two expressions (such as $=$, $\in$, etc.), the grammar given above doesn't allow formulae like $x = y = z$, so these operator can not be associative.

3.2.3 Priority of operators

Another source of ambiguity is the case where formulae contain two different predicate operators without any parenthesis such as

$$P \Rightarrow Q \Leftrightarrow R$$

$$P \wedge Q \vee R$$

$$\neg P \wedge Q$$

$$\forall x \cdot P \vee Q$$

This kind of ambiguity is generally resolved by defining priorities among operators which define how much *binding power* each operator has. We will use that mechanism here, retaining the most commonly used priorities. But, with the addition that we want to forbid cases where those priorities are not so well-accepted.

For instance, some people expect operators '$\wedge$' and '$\vee$' to have the same priority, while others expect operator '$\wedge$' to have higher priority. So when faced with formula $P \vee Q \wedge R$, some people read it as $(P \vee Q) \wedge R$ while others read it as $P \lor (Q \land R)$, which is quite different (just replace P and Q by $\top$ and R by $\bot$ to convince yourself).

To solve that ambiguity, we decided that operators '$\land$' and '$\lor$' indeed have the same priority, but that one cannot mix them together without using parenthesis. So, $P \land Q \lor R$ is considered ill-formed. One should write either $(P \land Q) \lor R$ or $P \land (Q \lor R)$.

The priorities defined for the event-B language are the following (from lower to higher priority)

$$\begin{array}{rl}
\forall x \cdot P \text{ and } \exists x \cdot P & \text{(mixing allowed)} \\
P \Rightarrow Q \text{ and } P \Leftrightarrow Q & \text{(mixing not allowed)} \\
P \land Q \text{ and } P \lor Q & \text{(mixing not allowed)} \\
\neg P &
\end{array}$$

We choose to give quantified predicates the lowest priority in order to ease their reading when embedded in long formulae. The main consequence of this choice is that the scope of the variables introduced by a quantifier is the longest sub-formula. For instance, in formula $(\forall x \cdot P \Rightarrow Q) \Rightarrow R$, the scope of variable x extends until predicate Q as can be easily seen by looking at matching parenthesis.

The following formulae show some examples of how those priorities are used to replace parenthesis in some common cases:

$$\begin{array}{rcl}
P \land Q \Rightarrow R & \text{is parsed as} & (P \land Q) \Rightarrow R \\
\forall x \cdot \exists y \cdot P & \text{is parsed as} & \forall x \cdot (\exists y \cdot P) \\
\forall x \cdot P \Rightarrow Q & \text{is parsed as} & \forall x \cdot (P \Rightarrow Q) \\
\forall x \cdot P \land Q & \text{is parsed as} & \forall x \cdot (P \land Q) \\
\forall x \cdot \neg P & \text{is parsed as} & \forall x \cdot (\neg P) \\
\neg P \Rightarrow Q & \text{is parsed as} & (\neg P) \Rightarrow Q \\
\neg P \land Q & \text{is parsed as} & (\neg P) \land Q
\end{array}$$

One should notice the difference with *classical* B [1] where $\forall x \cdot P \Rightarrow Q$ is parsed as $(\forall x \cdot P) \Rightarrow Q$ whereas, again, it is parsed here as $\forall x \cdot (P \Rightarrow Q)$.

3.2.4 Final syntax for predicates

As a result, we obtain the following non ambiguous grammar for predicates:

⟨*predicate*⟩ ::= { ⟨*quantifier*⟩ } ⟨*unquantified-predicate*⟩

⟨*quantifier*⟩ ::= '∀' ⟨*ident-list*⟩ '·'
| '∃' ⟨*ident-list*⟩ '·'

⟨*ident-list*⟩ ::= ⟨*ident*⟩ { '**,**' ⟨*ident*⟩ }

$$\begin{array}{lcl}
\langle \textit{unquantified-predicate} \rangle & ::= & \langle \textit{simple-predicate} \rangle \; [\text{ ‘}\Rightarrow\text{’ } \langle \textit{simple-predicate} \rangle \;] \\
 & | & \langle \textit{simple-predicate} \rangle \; [\text{ ‘}\Leftrightarrow\text{’ } \langle \textit{simple-predicate} \rangle \;] \\
 & & \\
\langle \textit{simple-predicate} \rangle & ::= & \langle \textit{literal-predicate} \rangle \; \{ \text{ ‘}\wedge\text{’ } \langle \textit{literal-predicate} \rangle \; \} \\
 & | & \langle \textit{literal-predicate} \rangle \; \{ \text{ ‘}\vee\text{’ } \langle \textit{literal-predicate} \rangle \; \} \\
 & & \\
\langle \textit{literal-predicate} \rangle & ::= & \{ \text{ ‘}\neg\text{’ } \} \; \langle \textit{atomic-predicate} \rangle \\
 & & \\
\langle \textit{atomic-predicate} \rangle & ::= & \text{‘}\bot\text{’} \\
 & | & \text{‘}\top\text{’} \\
 & | & \text{‘finite’ ‘}\texttt{(}\text{’ } \langle \textit{expression} \rangle \text{ ‘}\texttt{)}\text{’} \\
 & | & \text{‘partition’ ‘}\texttt{(}\text{’ } \langle \textit{expression-list} \rangle \text{ ‘}\texttt{)}\text{’} \\
 & | & \langle \textit{pair-expression} \rangle \; \langle \textit{relop} \rangle \; \langle \textit{pair-expression} \rangle \\
 & | & \text{‘}\texttt{(}\text{’ } \langle \textit{predicate} \rangle \text{ ‘}\texttt{)}\text{’} \\
 & & \\
\langle \textit{relop} \rangle & ::= & \text{‘}=\text{’} \mid \text{‘}\neq\text{’} \\
 & | & \text{‘}\in\text{’} \mid \text{‘}\notin\text{’} \mid \text{‘}\subset\text{’} \mid \text{‘}\not\subset\text{’} \mid \text{‘}\subseteq\text{’} \mid \text{‘}\not\subseteq\text{’} \\
 & | & \text{‘}<\text{’} \mid \text{‘}\leq\text{’} \mid \text{‘}>\text{’} \mid \text{‘}\geq\text{’} \\
 & & \\
\langle \textit{expression-list} \rangle & ::= & \langle \textit{expression} \rangle \; \{ \text{ ‘}\texttt{,}\text{’ } \langle \textit{expression} \rangle \; \}
\end{array}$$

Please note that for relational predicates, we are using ⟨*pair-expression*⟩ instead of ⟨*expression*⟩. That change will only allow expressions without quantifiers on each side of the relational operator. As a consequence, when one wants to use a quantified expression on either side, one will have to surround it with parenthesis. For instance, predicate $\lambda x \cdot x \in \mathbb{Z} \mid x = \mathrm{id}$ is not well-formed, one must write instead $(\lambda x \cdot x \in \mathbb{Z} \mid x) = \mathrm{id}$.

3.3 Expressions

The design principle for the syntax of expressions is the same as that of predicates, namely to enhance readability. To fulfill this goal, we use the same techniques: minimize the need for parenthesis where they are not really needed and prevent mixing operators when such a mix would be ambiguous.

3.3.1 Some Fine Points

Before presenting a first attempt of the syntax of expressions, we shall study some fine points about pairs, set comprehension, lambda abstraction, quantified expressions, and first and second projections.

Pair Construction. Pairs of expressions are constructed using the *maplet* operator ‘$\mapsto$’. Contrary to classical B [1], it is not possible to use a comma anymore. This change is due to the ambiguity of using commas for two different purposes in classical B: as a pair constructor and as a separator. For instance, set $\{1, 2\}$ can be seen as either a set containing the pair $(1, 2)$ or as a set containing the two elements 1 and 2. That was very confusing.

In event-B, a comma is always a separator and a maplet is a pair constructor. Below are some examples showing the consequences of this new approach:

Classical-B	Event-B
$x, y \in S$	$x \mapsto y \in S$
$x, y = z, t$	$x \mapsto y = z \mapsto t$
$f(x, y)$	$f(x \mapsto y)$

The last example is particularly blatant of the confusion between separator and pair constructor in classical B. When looking at formula $f(x, y)$, one has the impression that function f takes two separate arguments. But, this is not always true. For instance, variable x could hide a non scalar value. For instance, suppose that $x = a \mapsto b$, then the function application could be rewritten as either $f(a \mapsto b, y)$ or even as $f(a, b, y)$. In that latter case, function f now appears to take three arguments. This is clearly not satisfactory. In fact, function f only takes one argument, which can happen to be a pair. In that latter case, one should use a pair constructor to create that pair, that is use a maplet operator.

Set Comprehension. There are now two forms of set comprehension. The most general one is $\{x \cdot P(x) \mid E(x)\}$ which describes the set whose elements are $E(x)$, for all x such that $P(x)$ holds. For instance, the set of all even natural numbers can be written as $\{x \cdot x \in \mathbb{N} \mid 2 * x\}$.

The second form $\{E \mid P\}$ is just a short-hand for the first-one, which allows to write things more compactly. The difference from the first form is that the variables that are bound by the construct are not listed explicitly. They are inferred from the expression part. Continuing with our previous example, the set of all even natural numbers can then be written more compactly as $\{2 * x \mid x \in \mathbb{N}\}$, which corresponds more to the classical mathematical notation.

The rule for determining the variables which are bound by this second form is to take all variables that occur free in E. Thus, if we denote by x the list of the variables that occur free in E, then the second form is equivalent to $\{x \cdot P \mid E\}$.

Lambda Abstraction. For lambda abstraction, classical B [1] uses the form $(\lambda x \cdot P \mid E)$ where x is a list of variables, P a predicate and E an expression. This notation is fine when x is reduced to only one variable. For instance, expression $(\lambda x \cdot x \in \mathbb{N} \mid x + 1)$ denotes the classical succesor function on natural numbers. It is equal by definition to the set $\{x \cdot x \in \mathbb{N} \mid x + 1\}$.

But things get more complicated when x represents more than one variable. For instance, what is the meaning of expression $(\lambda a, b \cdot P \mid E)$. In classical B, the latter expression is defined as being the set $\{a, b \cdot P \mid a \mapsto b \mapsto E\}$. This is clearly unsatisfactory for event-B, as it turns out that, in the former expression, the comma that appears between a and b is not only a separator between two variables, but also a hidden pair constructor, as one can see when writing the equivalent set comprehension.

The crux of the matter is that the list of variables x introduced above, is much more than a simple list. Indeed, it describes the structure of the domain of the function defined by the lambda abstraction. For instance, when one writes, in classical B, the expression $(\lambda a, b \cdot P \mid E)$, one means that the domain of that function is $A \times B$ (where A and B are the types of bound variables a and b). Hence, the use of a comma is not appropriate here, as advocated in the paragraph above about *Pair Construction*.

The cure is easy, just say that x is not a list of variables, but a pattern that specifies the structure of the domain of the lambda abstraction. The example above is then to be written as $(\lambda a \mapsto b \cdot P \mid E)$. Moreover, this can be generalized to arbitrary domain structure by allowing arbitrary patterns after the lambda operator. The only constraints are that those patterns should be constructed out of distinct variables, pair constructors and parenthesis. The definition of the lambda abstraction $(\lambda x \cdot P \mid E)$ becomes $\{X \cdot P \mid x \mapsto E\}$ where X is the list of the variables that occur in x.

Other Quantified Expressions. The other quantified expressions are the quantified union and intersection. In this paragraph, we shall only consider quantified intersection, but everything will also apply to quantified union, mutatis mutandis.

A quantified intersection expression has the form $(\bigcap x \cdot P \mid E)$ where x is a list of variables, P a predicate and E an expression. It's defined as being a short form for the equivalent expression $\mathrm{inter}(\{x \cdot P \mid E\})$ which mixes generalized intersection and set comprehension. But, as we have seen above, we also have a short form for writing set comprehension. The question then arises whether we could also define a short form for generalized intersection. The answer is yes. We then have a second form which is $(\bigcap E \mid P)$ and which is defined has being equal to $\mathrm{inter}(\{E \mid P\})$.

Identity and Projections. In classical B [1], the identity operator takes one argument, like for instance in the expression $\mathrm{id}(S)$. In this expression, argument S serves two different purposes. On the one end, it allows to infer the type associated with the instantiated operator, which is $\mathbb{P}(\mathsf{super}(S) \times \mathsf{super}(S))$. On the other hand, it defines the domain of the instantiated operator, which is S.

In event-B, where types are inferred, this approach is unnecessarily restrictive as most of the time one does not care much in specifying the domain of the operator. This is particularly true in idioms such as $r \cap \mathrm{id} = \varnothing$ which says that r is irreflexive. The upgrade path from classical B is quite straightforward, just replace $\mathrm{id}(S)$ by $S \lhd \mathrm{id}$.

Similarly, the first and second projection operators are generic in event-B, while they take two sets as arguments in classical B, like for instance in the expression $\mathrm{prj}_1(A, B)$. In event-B, this expression is written as $(A \times B) \lhd \mathrm{prj}_1$.

3.3.2 A First Attempt

An ambiguous grammar for event-B expressions can loosely be defined as follows:

$$\begin{array}{lll}
\langle\textit{expression}\rangle & ::= & \langle\textit{expression}\rangle\ \langle\textit{binary-operator}\rangle\ \langle\textit{expression}\rangle \\
 & \mid & \langle\textit{unary-operator}\rangle\ \langle\textit{expression}\rangle \\
 & \mid & \langle\textit{expression}\rangle\ \text{`}{}^{-1}\text{'} \\
 & \mid & \langle\textit{expression}\rangle\ \text{`['}\ \langle\textit{expression}\rangle\ \text{`]'} \\
 & \mid & \langle\textit{expression}\rangle\ \text{`('}\ \langle\textit{expression}\rangle\ \text{`)'} \\
 & \mid & \text{`}\lambda\text{'}\ \langle\textit{ident-pattern}\rangle\ \text{`}\cdot\text{'}\ \langle\textit{predicate}\rangle\ \text{`}|\text{'}\ \langle\textit{expression}\rangle
\end{array}$$

| ⟨*quantifier*⟩ ⟨*ident-list*⟩ ‘·’ ⟨*predicate*⟩ ‘|’ ⟨*expression*⟩
| ⟨*quantifier*⟩ ⟨*expression*⟩ ‘|’ ⟨*predicate*⟩
| ‘{’ ⟨*ident-list*⟩ ‘·’ ⟨*predicate*⟩ ‘|’ ⟨*expression*⟩ ‘}’
| ‘{’ ⟨*expression*⟩ ‘|’ ⟨*predicate*⟩ ‘}’
| ‘bool’ ‘(’ ⟨*predicate*⟩ ‘)’
| ‘{’ [⟨*expression-list*⟩] ‘}’
| ‘(’ ⟨*expression*⟩ ‘)’
| ‘∅’ | ‘id’ | ‘prj_1’ | ‘prj_2’
| ‘ℤ’ | ‘ℕ’ | ‘$\mathbb{N}_1$’ | ‘pred’ | ‘succ’
| ‘BOOL’ | ‘TRUE’ | ‘FALSE’
| ⟨*ident*⟩
| ⟨*integer-literal*⟩

⟨*binary-operator*⟩ ::= ‘↦’ | ‘↔’ | ‘$\trel$’ | ‘$\srel$’ | ‘$\strel$’ | ‘⇸’ | ‘→’ | ‘⤔’ | ‘↣’ | ‘⤀’
| ‘↠’ | ‘⤖’ | ‘∪’ | ‘∩’ | ‘\’ | ‘×’ | ‘⊗’ | ‘∥’ | ‘∘’ | ‘;’ | ‘$\ovl$’ |
‘◁’ | ‘⩤’ | ‘▷’ | ‘⩥’ | ‘‥’ | ‘+’ | ‘−’ | ‘∗’ | ‘÷’ | ‘mod’ | ‘ˆ’

⟨*unary-operator*⟩ ::= ‘−’ | ‘card’ | ‘ℙ’ | ‘$\mathbb{P}_1$’ | ‘union’ | ‘inter’ | ‘dom’ | ‘ran’ |
‘min’ | ‘max’

⟨*quantifier*⟩ ::= ‘⋃’ | ‘⋂’

⟨*ident-pattern*⟩ ::= ⟨*ident-pattern*⟩ ‘↦’ ⟨*ident-pattern*⟩
| ‘(’ ⟨*ident-pattern*⟩ ‘)’
| ⟨*ident*⟩

As can be seen, there are many expression operators in the event-B language. So, we'll need to take a divide and conquer approach: to make things easier to grasp, we will first try to group all those operators into some categories.

3.3.3 Operator Groups

Basically, there are several kinds of expressions. The most important ones are shown in Figure 3.1. This figure reads as follows: there are three top-level kinds of expressions: sets, pairs and scalars. Relations and sets of relations are some special kinds of set. For instance, a relation between a set A and a set B is a subset of $A \times B$. The set of all relations between A and B is the set of all subsets of $A \times B$. Integers and booleans are also some special kind of scalar expression.

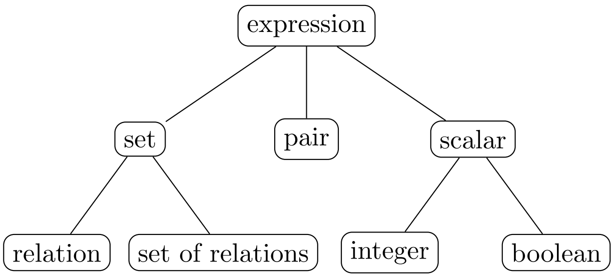

Figure 3.1: Kinds of expressions

We now define groups of similar expression operators (see Table 3.1 on the following page). The groups are defined by considering the shape of the operator (binary, unary, quantified, etc.) but also the kind of operator arguments and result. For each group, we will give one operator which will be used in the sequel as a distinguished representative of its group.

When examining that table, we can remark an interesting point: the operators that belong to the last three groups have the special property of being bounded: when one encounters such an operator, one can find easily where the expression involving that operator starts and where it ends: unary and 'bool' operators are always followed by a formula enclosed within parenthesis; set enumerations and comprehensions are enclosed within curly brackets. This is also the case of atomic expressions like integer and boolean literals or identifiers.

On the other hand, the operators of the other groups are not bounded by themselves, so one needs to define priorities and associativity laws for them in order to resolve potential ambiguities. We will first start by defining priorities between groups, then we will refine each group separately.

3.3.4 Priority of Operator Groups

We arbitrarily choose to define relative priorities such that groups of operators are sorted by increasing priority in table 3.1 on the next page. As a consequence, quantification operators have the lowest priority.

That order has been chosen because it reduces the number of needed parenthesis when writing most common expressions. Here are a few example to illustrate this. Each expression is stated twice, first without parenthesis, then fully parenthesized:

$$\begin{array}{rll}
A \cup B \mapsto C & \text{is parsed as} & (A \cup B) \mapsto C \\
a + b \mapsto c & \text{is parsed as} & (a + b) \mapsto c \\
a \mathbin{..} b \cup C & \text{is parsed as} & (a \mathbin{..} b) \cup C \\
a + b \mathbin{..} c & \text{is parsed as} & (a + b) \mathbin{..} c \\
r^{-1} \cup s & \text{is parsed as} & (r^{-1}) \cup s \\
r^{-1}(s) & \text{is parsed as} & (r^{-1})(s)
\end{array}$$

Also, we give the lowest priority to quantification operators so that, when embedded in a formula, they have to be written surrounded by parenthesis. This is consistent with the choice made for quantified predicates. An example formula is

$$(\lambda x \cdot x \in \mathbb{Z} \mid x + 1)^{-1}(3) = 2$$

3.3.5 Associativity of operators

Now, that priorities of groups have been defined, we will resolve remaining ambiguities separately for each group, defining how operators of each group can be mixed.

Group	Description	Repr.
Quantification operators	Given a list of quantified identifiers, a predicate and an expression, these operators produce a new expression.	$\lambda x \cdot P \mid E$
Pair constructor	Given two expressions, it produces a pair.	$E \mapsto F$
Set of relations constructors	Given two sets, these operators produce a set of relations.	$S \mathrel{+\!\!\!\!\rightarrow} T$
Binary set operators	Given two sets, these operators produce a new set.	$S \cup T$
Interval constructor	Given two integers, this operator produces a set.	$i \mathinner{\ldotp\ldotp} j$
Arithmetic operators	Given one or two integers, these operators produce a new integer.	$i + j$
Relational and functional image	Given a relation and an expression, these operators produce a new expression.	$r[s]$
Unary relation operator	Given a relation, this operator produces a new relation.	r^{-1}
Tightly bound unary operators	Given an expression, these operators produce another expression.	$\mathbb{P}(S)$
Predicate conversion	Given a predicate, this operator produces a new boolean expression.	$\mathrm{bool}(P)$
Set enumeration and comprehension	Given a list of expressions, or a list of quantified variables, a predicate and an expression, this operator produces a set.	$\{\ldots\}$

Table 3.1: Groups of similar expression operators

Quantification Operators. In this group, there is not much room for ambiguity, as when we encounter two quantification operators, it comes right from their syntax that the second one will be embedded in the first one. The only option left is whether the second quantified expression should be enclosed within parenthesis or not. We decide not to enforce parenthesis in this case. As a consequence, formula

$$\bigcap x \cdot x \subseteq \mathbb{Z} \mid \lambda y \cdot y = x \mid y \cup \{0\}$$

is parsed as

$$\bigcap x \cdot x \subseteq \mathbb{Z} \mid (\lambda y \cdot y = x \mid y \cup \{0\}) \ .$$

Pair Constructor. This group contains only the maplet operator, so we only have to define an associativity property for that operator. Although the maplet operator is not associative in the algebraic sense, it is very common usage to parse it as left-associative, so we shall keep that property. Then, an expression of the form $a \mapsto b \mapsto c$ will be parsed as $(a \mapsto b) \mapsto c$.

Set of Relations Constructors. No operator in this group is associative in the algebraic sense. Therefore, we decide to parse them as non-associative.

Binary Set Operators. This group contains various operators which are more or less compatible each with the other. So, let's first see how one can safely mix these operators in a formula, from a mathematical point of view. Table 3.2 on the following page shows operator compatibility. We write a cross at the intersection of a row and a column if the two operators are compatible in the following sense: operator $\mathsf{op}_{\mathsf{row}}$ is compatible with operator $\mathsf{op}_{\mathsf{col}}$ if and only if the following equality holds

$$(A \ \mathsf{op}_{\mathsf{row}} \ B) \ \mathsf{op}_{\mathsf{col}} \ C \ = \ A \ \mathsf{op}_{\mathsf{row}} \ (B \ \mathsf{op}_{\mathsf{col}} \ C).$$

For instance, the cross at the intersection of row two and column three tells us that $(A \cap B) \setminus C = A \cap (B \setminus C)$ and the cross at the intersection of row nine and column seven tells us that $(A \lhd r) \otimes s = A \lhd (r \otimes s)$.

We can see that the shape is quite irregular and that there are not so many cases where operators are compatible. So, to have an unambiguous language, we should stick to that compatibility relation and forbid any unparenthesized combination of incompatible operators. When two operators are compatible, we parse them as left-associative. Otherwise, one needs to use parenthesis to resolve ambiguities. For instance, formula $S \cup T \cup U$ is parsed as $(S \cup T) \cup U$, while formula $S \cup T \cap U$ is ill-formed and is rejected. One has to make precise the meaning of that last formula, writing either $(S \cup T) \cap U$ or $S \cup (T \cap U)$.

There is only one case where we want to allow the combination of two incompatible operators: we parse the cartesian product operator as left-associative. This exception to the above rule is justified by the fact that we want to be consistent with the left-associativity we have given to the maplet operator. Then, one can write $a \mapsto b \mapsto c \in A \times B \times C$ when one actually means $(a \mapsto b) \mapsto c \in (A \times B) \times C$.

	$\cup$	$\cap$	$\setminus$	$\times$	$\circ$	;	$\otimes$	$\parallel$	$\lhd\!\!\!+$	$\lhd$	⩤	$\rhd$	⩥
$\cup$	×												
$\cap$		×	×									×	×
$\setminus$													
$\times$													
$\circ$					×								
;						×						×	×
$\otimes$													
$\parallel$													
$\lhd\!\!\!+$									×				
$\lhd$		×	×			×	×					×	×
⩤		×	×			×	×					×	×
$\rhd$													
⩥													

Table 3.2: Compatibility of binary set operators

Interval Constructor. This group contains only one operator: '..'. There is no point in having this operator used twice in the same formula (which would give the nonsensical formula $a\ ..\ b\ ..\ c$). So, this operator is parsed as non-associative.

Arithmetic Operators. For these operators, we choose to retain the Ada language specification for defining priorities and associativity: operators '+' and '−' both have the same priority and are parsed as left-associative, operators '$*$', '$\div$' and 'mod' have higher priority and are also parsed as left-associative. Note that this choice is different from the one made for instance in the C language, where there is a special priority for unary '−'. We did not retain that last point as it can lead to valid but hard to read expressions like $a + - - - - b$ which means $a + b$.

Finally, the exponentiation operator has the least priority and is parsed as non-associative.

Relational and Functional Image. We choose to make these operations left-associative, although they are not associative in the algebraic sense. This follows common usage and is indeed important to have easy to read formulas. If these operators were not associative, one would have to write quite intricate formulas just to express successive function application: $((f(a))(b))(c)$. With the left-associativity we've added, this becomes $f(a)(b)(c)$.

Unary Relation Operator. This group contains only one operator '$^{-1}$', which can be repeated, obviously, so that r^{-1-1} is parsed as $(r^{-1})^{-1}$.

3.3.6 Final syntax for expressions

As a result, we obtain the following non ambiguous grammar for expressions. An important point is that non-terminals are named after the group of the top-level operators appearing in their production rule. This can be somewhat misleading as, for instance, ⟨*pair-expression*⟩ can be derived as formula ℤ, which is clearly not a pair. This naming policy was chosen not to leave any information (just numbering non-terminals 1, 2,. . . would miss some structural property of the grammar).

```
⟨expression⟩          ::= ‘λ’ ⟨ident-pattern⟩ ‘·’ ⟨predicate⟩ ‘|’ ⟨expression⟩
                       |  ‘⋃’ ⟨ident-list⟩ ‘·’ ⟨predicate⟩ ‘|’ ⟨expression⟩
                       |  ‘⋃’ ⟨expression⟩ ‘|’ ⟨predicate⟩
                       |  ‘⋂’ ⟨ident-list⟩ ‘·’ ⟨predicate⟩ ‘|’ ⟨expression⟩
                       |  ‘⋂’ ⟨expression⟩ ‘|’ ⟨predicate⟩
                       |  ⟨pair-expression⟩

⟨ident-pattern⟩       ::= ⟨ident-pattern⟩ { ‘↦’ ⟨ident-pattern⟩ }
                       |  ‘(’ ⟨ident-pattern⟩ ‘)’
                       |  ⟨ident⟩

⟨pair-expression⟩     ::= ⟨relation-set-expr⟩ { ‘↦’ ⟨relation-set-expr⟩ }

⟨relation-set-expr⟩   ::= ⟨set-expr⟩ [ ⟨relational-set-op⟩ ⟨set-expr⟩ ]

⟨relational-set-op⟩   ::= ‘↔’ | ‘↞→’ | ‘←↠’ | ‘↞↠’
                       |  ‘⇸’ | ‘→’ | ‘⤔’ | ‘↣’ | ‘⤀’ | ‘↠’ | ‘⤖’

⟨set-expr⟩            ::= ⟨interval-expr⟩ { ‘∪’ ⟨interval-expr⟩ }
                       |  ⟨interval-expr⟩ { ‘×’ ⟨interval-expr⟩ }
                       |  ⟨interval-expr⟩ { ‘⊲+’ ⟨interval-expr⟩ }
                       |  ⟨interval-expr⟩ { ‘∘’ ⟨interval-expr⟩ }
                       |  ⟨interval-expr⟩ ‘∥’ ⟨interval-expr⟩
                       |  [ ⟨domain-modifier⟩ ] ⟨relation-expr⟩

⟨domain-modifier⟩     ::= ⟨interval-expr⟩ ( ‘◁’ | ‘⩤’ )

⟨relation-expr⟩       ::= ⟨interval-expr⟩ ‘⊗’ ⟨interval-expr⟩
                       |  ⟨interval-expr⟩ { ‘;’ ⟨interval-expr⟩ }
                          [ ⟨range-modifier⟩ ]
                       |  ⟨interval-expr⟩ { ‘∩’ ⟨interval-expr⟩ }
                          [ ‘∖’ ⟨interval-expr⟩ | ⟨range-modifier⟩ ]

⟨range-modifier⟩      ::= ( ‘▷’ | ‘⩥’ ) ⟨interval-expr⟩

⟨interval-expr⟩       ::= ⟨arithmetic-expr⟩ [ ‘..’ ⟨arithmetic-expr⟩ ]

⟨arithmetic-expr⟩     ::= [ ‘−’ ] ⟨term⟩ { ( ‘+’ | ‘−’ ) ⟨term⟩ }

⟨term⟩                ::= ⟨factor⟩ { ( ‘∗’ | ‘÷’ | ‘mod’ ) ⟨factor⟩ }

⟨factor⟩              ::= ⟨image⟩ [ ‘⌢’ ⟨image⟩ ]
```

⟨*image*⟩	::=	⟨*primary*⟩ { ‘[’ ⟨*expression*⟩ ‘]’ | ‘(’ ⟨*expression*⟩ ‘)’ }
⟨*primary*⟩	::=	⟨*simple-expr*⟩ { ‘$^{-1}$’ }
⟨*simple-expr*⟩	::=	‘bool’ ‘(’ ⟨*predicate*⟩ ‘)’
	|	⟨*unary-op*⟩ ‘(’ ⟨*expression*⟩ ‘)’
	|	‘(’ ⟨*expression*⟩ ‘)’
	|	‘{’ ⟨*ident-list*⟩ ‘·’ ⟨*predicate*⟩ ‘|’ ⟨*expression*⟩ ‘}’
	|	‘{’ ⟨*expression*⟩ ‘|’ ⟨*predicate*⟩ ‘}’
	|	‘{’ [⟨*expression-list*⟩] ‘}’
	|	‘∅’ | ‘id’ | ‘prj_1’ | ‘prj_2’
	|	‘ℤ’ | ‘ℕ’ | ‘$\mathbb{N}_1$’ | ‘pred’ | ‘succ’
	|	‘BOOL’ | ‘TRUE’ | ‘FALSE’
	|	⟨*ident*⟩
	|	⟨*integer-literal*⟩
⟨*unary-op*⟩	::=	‘card’ | ‘ℙ’ | ‘$\mathbb{P}_1$’ | ‘union’ | ‘inter’ | ‘dom’ | ‘ran’ | ‘min’ | ‘max’

4 Static Checking

This chapter describes how mathematical formulae (predicates and expressions) are to be statically checked for being meaningful. We first describe an abstract syntax for formulae. Then, we state the static checks that are to be done, based on that abstract syntax:

- legibility,
- type-check.

4.1 Abstract Syntax

In this section, we specify an abstract syntax for mathematical formulae. This abstract syntax is based on the concrete syntax described in Section 3.2.4 on page 11 and Section 3.3.6 on page 20. The difference is that the abstract syntax only conserves the essence of the concrete syntax. So, all concrete matter like priorities and tokens do not appear anymore.

The abstract syntax is described using production rules. Each rule has its own label. It is made of a left-hand part which denotes some kind of formula (predicate, expression, identifier list, expression list) and a right hand part which denotes a list of sub-formulae together with some attributes. To distinguish an attribute from a sub-formulae, we enclose the former within square brackets. Moreover, to make rules short, we use single letters, possibly subscripted, to denote formulae: a P denotes a predicate, E an expression, L a list of identifiers, I an identifier, M a list of expressions, and Q a pattern for lambda abstraction.

The production rules for predicates are:

$$\begin{aligned}
\textsf{pred-bin:}\ P &::= P_1\ P_2\ [\textit{pred-binop}]\\
\textsf{pred-una:}\ P &::= P_1\\
\textsf{pred-quant:}\ P &::= L_1\ P_1\ [\textit{pred-quant}]\\
\textsf{pred-lit:}\ P &::= [\textit{pred-lit}]\\
\textsf{pred-simp:}\ P &::= E_1\\
\textsf{pred-mult:}\ P &::= M_1\\
\textsf{pred-rel:}\ P &::= E_1\ E_2\ [\textit{pred-relop}]
\end{aligned}$$

where

$$\begin{aligned}
&\textit{pred-binop} \in \{\mathrm{land}, \mathrm{lor}, \mathrm{limp}, \mathrm{leqv}\}\\
&\textit{pred-quant} \in \{\mathrm{forall}, \mathrm{exists}\}\\
&\textit{pred-lit} \in \{\mathrm{btrue}, \mathrm{bfalse}\}\\
&\textit{pred-relop} \in \left\{\begin{array}{l}\mathrm{equal}, \mathrm{notequal}, \mathrm{lt}, \mathrm{le}, \mathrm{gt}, \mathrm{ge},\\ \mathrm{in}, \mathrm{notin}, \mathrm{subset}, \mathrm{notsubset}, \mathrm{subseteq}, \mathrm{notsubseteq}\end{array}\right\}.
\end{aligned}$$

The production rules for lists of identifiers and identifiers are:

$$\begin{array}{rl} \textsf{ident-list:}\ L & ::= I_1\ I_2\ \ldots\ I_n \\ \textsf{ident:}\ I & ::= [\mathit{name}] \end{array}$$

where

$1 \leq n$
name is a string of characters.

The production rules for expressions are:

$$\begin{array}{rl} \textsf{expr-bin:}\ E & ::= E_1\ E_2\ [\mathit{expr\text{-}binop}] \\ \textsf{expr-una:}\ E & ::= E_1\ [\mathit{expr\text{-}unop}] \\ \textsf{expr-lambda:}\ E & ::= Q_1\ P_1\ E_1 \\ \textsf{expr-quant1:}\ E & ::= L_1\ P_1\ E_1\ [\mathit{expr\text{-}quant}] \\ \textsf{expr-quant2:}\ E & ::= E_1\ P_1\ [\mathit{expr\text{-}quant}] \\ \textsf{expr-bool:}\ E & ::= P_1 \\ \textsf{expr-eset:}\ E & ::= M_1 \\ \textsf{expr-ident:}\ E & ::= I_1 \\ \textsf{expr-atom:}\ E & ::= [\mathit{expr\text{-}lit}] \\ \textsf{expr-int:}\ E & ::= [\mathit{int\text{-}lit}] \\ \\ \textsf{pattern:}\ Q & ::= Q_1\ Q_2 \\ \textsf{pattern-ident:}\ Q & ::= I_1 \\ \\ \textsf{expr-list:}\ M & ::= E_1\ E_2\ \ldots\ E_n \end{array}$$

where

$$\begin{array}{rl} \mathit{expr\text{-}binop} & \in \left\{ \begin{array}{l} \text{funimage, relimage, mapsto,} \\ \text{rel, trel, srel, strel,} \\ \text{pfun, tfun, pinj, tinj, psur, tsur, tbij,} \\ \text{bunion, binter, setminus, cprod, dprod, pprod,} \\ \text{bcomp, fcomp, ovl, domres, domsub, ranres, ransub,} \\ \text{upto, plus, minus, mul, div, mod, expn} \end{array} \right\} \\ \mathit{expr\text{-}unop} & \in \left\{ \begin{array}{l} \text{uminus, converse, card, pow, pow1,} \\ \text{union, inter, dom, ran, min, max} \end{array} \right\} \\ \mathit{expr\text{-}quant} & \in \left\{ \text{qunion, qinter, cset} \right\} \\ \mathit{expr\text{-}lit} & \in \left\{ \begin{array}{l} \text{integer, natural, natural1, pred, succ,} \\ \text{bool, true, false,} \\ \text{emptyset, id, prj1, prj2} \end{array} \right\} \\ \mathit{int\text{-}lit} & \text{is an integer number.} \end{array}$$

4.2 Legibility

Each occurrence of an identifier in a formula (that is a predicate or an expression) can be either free or bound. Intuitively, a free occurrence of an identifier refers to a declaration of that identifier in a scope outside of the formula, while a bound occurrence corresponds to a local declaration introduced by a quantifier in the formula itself.

For a formula to be considered legible, we ask that, beyond being syntactically correct, it also satisfies the two following conditions:

1. Any identifier that occurs in the formula, should have only free occurrences or bound occurrences, but not both.

2. Any identifier that occurs bound in the formula, should be bound in exactly one place (i.e., by only one quantifier).

These conditions have been coined so that any occurrence of an identifier in a formula always denotes exactly the same data.

For instance, the following formula is illegible (it doesn't satisfy the first condition)

$$(\lambda x \cdot x \in \mathbb{Z} \mid x + 1)\,(x) = x + 1$$

it should be written

$$(\lambda y \cdot y \in \mathbb{Z} \mid y + 1)\,(x) = x + 1\ .$$

And the following formula is also illegible (failing to satisfy the second condition)

$$(\lambda x \cdot x \in \mathbb{Z} \mid x + 1) = (\lambda x \cdot x \in \mathbb{Z} \mid x + 1)$$

it should be written

$$(\lambda x \cdot x \in \mathbb{Z} \mid x + 1) = (\lambda y \cdot y \in \mathbb{Z} \mid y + 1)\ \ .$$

The rest of this section formalizes these legibility conditions using an attribute grammar formalism on the abstract syntax of formulae. For that, we add three attributes to the nodes of the abstract syntax tree:

- Attribute *bound* is synthesized and contains the set of identifiers that occur bound in the formula rooted at the current node.

- Attribute *free* is synthesized and contains the set of identifiers that occur free in the formula rooted at the current node.

- Attribute *leg* is synthesized and contains a boolean value which is TRUE if and only if the formula rooted at the current node is legible.

The value of these three attributes are given by the following set of equations on the production rules of the abstract syntax:

pred-bin: $P ::= P_1\ P_2\ [\mathit{pred\text{-}binop}]$

$$P.\mathit{bound} = P_1.\mathit{bound} \cup P_2.\mathit{bound}$$

$$P.\mathit{free} = P_1.\mathit{free} \cup P_2.\mathit{free}$$

$$P.\mathit{leg} = \mathrm{bool}\left(\begin{array}{ll} & P_1.\mathit{leg} = \mathrm{TRUE} \\ \wedge & P_2.\mathit{leg} = \mathrm{TRUE} \\ \wedge & P_1.\mathit{free} \cap P_2.\mathit{bound} = \varnothing \\ \wedge & P_1.\mathit{bound} \cap P_2.\mathit{free} = \varnothing \\ \wedge & P_1.\mathit{bound} \cap P_2.\mathit{bound} = \varnothing \end{array}\right)$$

pred-una: $P ::= P_1$

$$P.\mathit{bound} = P_1.\mathit{bound}$$

$$P.\mathit{free} = P_1.\mathit{free}$$

$$P.\mathit{leg} = P_1.\mathit{leg}$$

pred-quant: $P ::= L_1\ P_1\ [\mathit{pred\text{-}quant}]$

$$\begin{aligned} P.\mathit{bound} &= P_1.\mathit{bound} \cup L_1.\mathit{free} \\ P.\mathit{free} &= P_1.\mathit{free} \setminus L_1.\mathit{free} \\ P.\mathit{leg} &= \mathrm{bool}\left(\begin{array}{ll} & L_1.\mathit{leg} = \mathrm{TRUE} \\ \wedge & P_1.\mathit{leg} = \mathrm{TRUE} \\ \wedge & P_1.\mathit{bound} \cap L_1.\mathit{free} = \varnothing \end{array}\right) \end{aligned}$$

pred-lit: $P ::= [\mathit{pred\text{-}lit}]$

$$\begin{aligned} P.\mathit{bound} &= \varnothing \\ P.\mathit{free} &= \varnothing \\ P.\mathit{leg} &= \mathrm{TRUE} \end{aligned}$$

pred-simp: $P ::= E_1$

$$\begin{aligned} P.\mathit{bound} &= E_1.\mathit{bound} \\ P.\mathit{free} &= E_1.\mathit{free} \\ P.\mathit{leg} &= E_1.\mathit{leg} \end{aligned}$$

pred-mult: $P ::= M_1$

$$\begin{aligned} P.\mathit{bound} &= M_1.\mathit{bound} \\ P.\mathit{free} &= M_1.\mathit{free} \\ P.\mathit{leg} &= M_1.\mathit{leg} \end{aligned}$$

pred-rel: $P ::= E_1\ E_2\ [\mathit{pred\text{-}relop}]$

$$\begin{aligned} P.\mathit{bound} &= E_1.\mathit{bound} \cup E_2.\mathit{bound} \\ P.\mathit{free} &= E_1.\mathit{free} \cup E_2.\mathit{free} \\ P.\mathit{leg} &= \mathrm{bool}\left(\begin{array}{ll} & E_1.\mathit{leg} = \mathrm{TRUE} \\ \wedge & E_2.\mathit{leg} = \mathrm{TRUE} \\ \wedge & E_1.\mathit{free} \cap E_2.\mathit{bound} = \varnothing \\ \wedge & E_1.\mathit{bound} \cap E_2.\mathit{free} = \varnothing \\ \wedge & E_1.\mathit{bound} \cap E_2.\mathit{bound} = \varnothing \end{array}\right) \end{aligned}$$

ident-list: $L ::= I_1\ I_2\ \ldots\ I_n$

$$\begin{aligned} L.\mathit{bound} &= \varnothing \\ L.\mathit{free} &= \{k \cdot k \in 1 \mathinner{..} n \mid I_k.\mathit{name}\} \\ L.\mathit{leg} &= \mathrm{bool}(\forall i, j \cdot i \in 1 \mathinner{..} n \wedge j \in 1 \mathinner{..} n \wedge i \neq j \Rightarrow I_i.\mathit{name} \neq I_j.\mathit{name}) \end{aligned}$$

expr-bin: $E ::= E_1\ E_2\ [\mathit{expr\text{-}binop}]$

$$\begin{aligned} E.\mathit{bound} &= E_1.\mathit{bound} \cup E_2.\mathit{bound} \\ E.\mathit{free} &= E_1.\mathit{free} \cup E_2.\mathit{free} \\ E.\mathit{leg} &= \mathrm{bool}\left(\begin{array}{ll} & E_1.\mathit{leg} = \mathrm{TRUE} \\ \wedge & E_2.\mathit{leg} = \mathrm{TRUE} \\ \wedge & E_1.\mathit{free} \cap E_2.\mathit{bound} = \varnothing \\ \wedge & E_1.\mathit{bound} \cap E_2.\mathit{free} = \varnothing \\ \wedge & E_1.\mathit{bound} \cap E_2.\mathit{bound} = \varnothing \end{array}\right) \end{aligned}$$

expr-una: $E ::= E_1\ [\mathit{expr\text{-}unop}]$

$$\begin{aligned} E.\mathit{bound} &= E_1.\mathit{bound} \\ E.\mathit{free} &= E_1.\mathit{free} \\ E.\mathit{leg} &= E_1.\mathit{leg} \end{aligned}$$

expr-lambda: $E ::= Q_1\ P_1\ E_1$

$$\begin{aligned}
E.\mathit{bound} &= P_1.\mathit{bound} \cup E_1.\mathit{bound} \cup Q_1.\mathit{free} \\
E.\mathit{free} &= (P_1.\mathit{free} \cup E_1.\mathit{free}) \setminus Q_1.\mathit{free} \\
E.\mathit{leg} &= \text{bool} \left(\begin{array}{ll}
 & Q_1.\mathit{leg} = \text{TRUE} \\
\wedge & P_1.\mathit{leg} = \text{TRUE} \\
\wedge & E_1.\mathit{leg} = \text{TRUE} \\
\wedge & P_1.\mathit{free} \cap E_1.\mathit{bound} = \varnothing \\
\wedge & P_1.\mathit{bound} \cap E_1.\mathit{free} = \varnothing \\
\wedge & P_1.\mathit{bound} \cap E_1.\mathit{bound} = \varnothing \\
\wedge & P_1.\mathit{bound} \cap Q_1.\mathit{free} = \varnothing \\
\wedge & E_1.\mathit{bound} \cap Q_1.\mathit{free} = \varnothing
\end{array} \right)
\end{aligned}$$

expr-quant1: $E ::= L_1\ P_1\ E_1\ [\mathit{expr\text{-}quant}]$

$$\begin{aligned}
E.\mathit{bound} &= P_1.\mathit{bound} \cup E_1.\mathit{bound} \cup L_1.\mathit{free} \\
E.\mathit{free} &= (P_1.\mathit{free} \cup E_1.\mathit{free}) \setminus L_1.\mathit{free} \\
E.\mathit{leg} &= \text{bool} \left(\begin{array}{ll}
 & L_1.\mathit{leg} = \text{TRUE} \\
\wedge & P_1.\mathit{leg} = \text{TRUE} \\
\wedge & E_1.\mathit{leg} = \text{TRUE} \\
\wedge & P_1.\mathit{free} \cap E_1.\mathit{bound} = \varnothing \\
\wedge & P_1.\mathit{bound} \cap E_1.\mathit{free} = \varnothing \\
\wedge & P_1.\mathit{bound} \cap E_1.\mathit{bound} = \varnothing \\
\wedge & P_1.\mathit{bound} \cap L_1.\mathit{free} = \varnothing \\
\wedge & E_1.\mathit{bound} \cap L_1.\mathit{free} = \varnothing
\end{array} \right)
\end{aligned}$$

expr-quant2: $E ::= E_1\ P_1\ [\mathit{expr\text{-}quant}]$

$$\begin{aligned}
E.\mathit{bound} &= P_1.\mathit{bound} \cup E_1.\mathit{bound} \cup E_1.\mathit{free} \\
E.\mathit{free} &= P_1.\mathit{free} \setminus E_1.\mathit{free} \\
E.\mathit{leg} &= \text{bool} \left(\begin{array}{ll}
 & E_1.\mathit{leg} = \text{TRUE} \\
\wedge & P_1.\mathit{leg} = \text{TRUE} \\
\wedge & P_1.\mathit{bound} \cap E_1.\mathit{bound} = \varnothing \\
\wedge & P_1.\mathit{bound} \cap E_1.\mathit{free} = \varnothing
\end{array} \right)
\end{aligned}$$

expr-bool: $E ::= P_1$

$$\begin{aligned}
E.\mathit{bound} &= P_1.\mathit{bound} \\
E.\mathit{free} &= P_1.\mathit{free} \\
E.\mathit{leg} &= P_1.\mathit{leg}
\end{aligned}$$

expr-eset: $E ::= M_1$

$$\begin{aligned}
E.\mathit{bound} &= M_1.\mathit{bound} \\
E.\mathit{free} &= M_1.\mathit{free} \\
E.\mathit{leg} &= M_1.\mathit{leg}
\end{aligned}$$

expr-ident: $E ::= I_1$

$$\begin{aligned}
E.\mathit{bound} &= \varnothing \\
E.\mathit{free} &= \{I_1.\mathit{name}\} \\
E.\mathit{leg} &= \text{TRUE}
\end{aligned}$$

expr-atom: $E ::= [\mathit{expr\text{-}lit}]$

$$\begin{aligned}
E.\mathit{bound} &= \varnothing \\
E.\mathit{free} &= \varnothing \\
E.\mathit{leg} &= \text{TRUE}
\end{aligned}$$

expr-int: $E ::= [\mathit{int\text{-}lit}]$

$$E.\mathit{bound} = \varnothing$$
$$E.\mathit{free} = \varnothing$$
$$E.\mathit{leg} = \mathrm{TRUE}$$

pattern: $Q ::= Q_1\ Q_2$

$$Q.\mathit{bound} = \varnothing$$
$$Q.\mathit{free} = Q_1.\mathit{free} \cup Q_2.\mathit{free}$$
$$Q.\mathit{leg} = \mathrm{TRUE}$$

pattern-ident: $Q ::= I_1$

$$Q.\mathit{bound} = \varnothing$$
$$Q.\mathit{free} = \{I_1.\mathit{name}\}$$
$$Q.\mathit{leg} = \mathrm{TRUE}$$

expr-list: $M ::= E_1\ E_2\ \ldots\ E_n$

$$M.\mathit{bound} = (\textstyle\bigcup k \cdot k \in 1\mathinner{..}n \mid E_k.\mathit{bound})$$
$$M.\mathit{free} = (\textstyle\bigcup k \cdot k \in 1\mathinner{..}n \mid E_k.\mathit{free})$$
$$M.\mathit{leg} = \mathrm{bool}\left(\begin{array}{cc} & (\forall k \cdot k \in 1\mathinner{..}n \Rightarrow E_k.\mathit{leg} = \mathrm{TRUE}) \\ \wedge & \left(\begin{array}{c} \forall i, j \cdot i \in 1\mathinner{..}n \wedge j \in 1\mathinner{..}n \wedge i \neq j \\ \Rightarrow E_i.\mathit{bound} \cap E_j.\mathit{bound} = \varnothing \end{array}\right) \\ \wedge & \left(\begin{array}{c} \forall i, j \cdot i \in 1\mathinner{..}n \wedge j \in 1\mathinner{..}n \wedge i \neq j \\ \Rightarrow E_i.\mathit{bound} \cap E_j.\mathit{free} = \varnothing \end{array}\right) \end{array}\right)$$

4.3 Type Checking

Type checking consists of checking, statically, that a formula is meaningful in a certain context. For that, we associate a type with each expression that occurs in a formula. This type is the set of all values that the expression can take. Then, we check that the formula abides by some type checking rules. Those rules enforce that the operators used can be meaningful. Unfortunately, type checking, as it is a static check, cannot by itself prove that a formula is meaningful. For some operators, like integer division, we will also need to check some additional dynamic constraints (e.g., that the denominator is not zero). This will be specified in the well-definedness dynamic checks (see chapter 5 on page 41).

The result of type checking is twofold. Firstly, it says whether a given formula is well-typed (that is abides by the type checking rules). Secondly, it computes an enriched context that associates a type with every identifier occurring free in the formula.

In the sequel of this section, we shall first specify more formally concepts such as type, type variable, typing environment and typing equation. Then, we shall specify type checking using an attribute grammar formalism as was done for legibility. Finally, we give some illustrating examples of type-checking.

4.3.1 Typing Concepts

As said previously, a type denotes the set of values that an expression can take. Moreover, we want this set to be derived statically, based on the form of the expression and the context in which it appears. As a consequence, a type can take one of the three following forms:

- a basic set, that is a predefined set ($\mathbb{Z}$ or BOOL) or a carrier set provided by the user (i.e., an identifier);
- a power set of another type, such as $\mathbb{P}(\mathbb{Z})$;
- a cartesian product of two types, such as $\mathbb{Z} \times \mathrm{BOOL}$.

A type variable is a meta-variable that can denote any type. In the sequel, we shall use lowercase Greek letters (α, β, γ, ...) to denote type variables.

A typing environment represents the context in which a formula is to be type checked. A typing environment is a partial function from the set of all identifiers to the set of all possible types. For instance, the typing environment

$$\{\text{‘}\mathtt{a}\text{’} \mapsto \mathbb{Z}, \text{‘}\mathtt{b}\text{’} \mapsto \mathbb{P}(\mathbb{Z} \times \mathrm{BOOL}), \text{‘}\mathtt{c}\text{’} \mapsto \alpha\}$$

says that identifier ‘a’ has type $\mathbb{Z}$, identifier ‘b’ has type $\mathbb{P}(\mathbb{Z} \times \mathrm{BOOL})$ (i.e., is a relation between integers and booleans) and identifier ‘c’ is typed by type variable α.

If an identifier i has been defined as a carrier set, then it will appear in the typing environment as the pair $i \mapsto \mathbb{P}(i)$.

A typing equation is a pair of types. In the sequel, we will write typing equations as $\tau_1 \equiv \tau_2$, instead of the more classical pair $\tau_1 \mapsto \tau_2$. This is mere syntactical sugar to enhance legibility.

A typing equation is said to be *satisfiable* if, and only if, there exists an assignment to the type variables it contains such that, when replacing these type variables by their value, the two components of the pair are equal (i.e., denote the same type). For instance, typing equation $\alpha \times \mathrm{BOOL} \equiv \mathbb{Z} \times \beta$ is satisfiable (take $\mathbb{Z}$ for α and BOOL for β). In contrast, type equation $\mathbb{P}(\alpha) \equiv \mathbb{Z}$ and $\mathbb{Z} \equiv$ ‘S’ are unsatisfiable (in the last sentence, remember that ‘S’ denotes a carrier set).

Similarly, a typing equation is said to be *uniquely satisfiable* if, and only if, there exists a unique assignment of type variables that satisfies it. For instance, $\alpha \equiv \mathbb{Z}$ is uniquely satisfiable (the only assignment that satisfies it is to take $\mathbb{Z}$ for α), while the type equation $\alpha \equiv \beta$, although satisfiable, is not uniquely satisfiable (to satisfy it, we only need that α and β are assigned the same type, but that type is arbitrary).

These two notions of satisfiability are extended to sets of type equations, with the additional proviso, that the satisfying assignment of type variables is done globally for all type equations in the set. For instance, the set $\{\alpha \equiv \mathbb{Z},\ \beta \equiv \mathrm{BOOL}\}$ is (uniquely) satisfiable, while the set $\{\alpha \equiv \mathbb{Z},\ \alpha \equiv \mathrm{BOOL}\}$ is not satisfiable, although each equation, taken separately, is satisfiable.

4.3.2 Specification of Type Check

The abstract grammar of expressions is extended with the following attributes:

- Attribute *ityvars* (resp. *styvars*) is inherited (resp. synthesized) and contains the set of type variables that have been used so far.

- Attribute *ityenv* (resp. *styenv*) is inherited (resp. synthesized) and contains the current typing environment.
- Attribute *ityeqs* (resp. *styeqs*) is inherited (resp. synthesized) and contains the set of typing equations that have been collected so far.
- Attribute *type* is synthesized and contains a type.

These attributes are added to all non-terminals, except *type* which is not defined for predicates (there is no type associated with a predicate) nor list of identifiers.

Type checking then consists of initializing the attribute grammar by giving values to inherited attributes of the root R of the tree and then evaluating the attribute grammar. Type check succeeds iff, after evaluation, the set of typing equations $R.\mathit{styeqs}$ is uniquely satisfiable. Moreover, in case of success, the resulting typing environment is $R.\mathit{styenv}$, where all type variables have been replaced by the values that satisfy the latter set of typing equations.

Initialization of the attribute grammar consists of the following three equations (where R denotes the root of the tree):

$$\begin{aligned} R.\mathit{ityvars} &= \varnothing \\ R.\mathit{ityenv} &= \text{initial typing environment} \\ R.\mathit{ityeqs} &= \varnothing \end{aligned}$$

Please note that the initial typing environment must not contain any type variable.

The rest of this section describes the equations for each production rule of the attribute grammar. In some places, we use a shortcut to denote some set of equations. The notation

$$A.\mathit{inherited} = B.\mathit{synthesized}$$

means

$$\begin{aligned} A.\mathit{ityvars} &= B.\mathit{styvars} \\ A.\mathit{ityenv} &= B.\mathit{styenv} \\ A.\mathit{ityeqs} &= B.\mathit{styeqs} \end{aligned}$$

We also use the term *fresh type variable* to denote a type variable which doesn't occur in attribute *ityvars* of the left hand side of a production rule. For instance, in the equations of production rule pred-rel, α denotes a type variable such that $\alpha \notin P.\mathit{ityvars}$.

The set of equations of the attribute grammar is:

pred-bin: $P ::= P_1\ P_2\ [\mathit{pred\text{-}binop}]$

$$\begin{aligned} P_1.\mathit{inherited} &= P.\mathit{inherited} \\ P_2.\mathit{inherited} &= P_1.\mathit{synthesized} \\ P.\mathit{synthesized} &= P_2.\mathit{synthesized} \end{aligned}$$

pred-una: $P ::= P_1$

$$\begin{aligned} P_1.\mathit{inherited} &= P.\mathit{inherited} \\ P.\mathit{synthesized} &= P_1.\mathit{synthesized} \end{aligned}$$

pred-quant: $P ::= L_1 \ P_1 \ [\mathit{pred\text{-}quant}]$

$$\begin{aligned} L_1.\mathit{inherited} &= P.\mathit{inherited} \\ P_1.\mathit{inherited} &= L_1.\mathit{synthesized} \\ P.\mathit{synthesized} &= P_1.\mathit{synthesized} \end{aligned}$$

pred-lit: $P ::= [\mathit{pred\text{-}lit}]$

$$P.\mathit{synthesized} = P.\mathit{inherited}$$

pred-simp: $P ::= E_1$

Let α be a fresh type variable in

$$\begin{aligned} E_1.\mathit{ityvars} &= P.\mathit{ityvars} \cup \{\alpha\} \\ E_1.\mathit{ityenv} &= P.\mathit{ityenv} \\ E_1.\mathit{ityeqs} &= P.\mathit{ityeqs} \\ P.\mathit{styvars} &= E_1.\mathit{styvars} \\ P.\mathit{styenv} &= E_1.\mathit{styenv} \\ P.\mathit{styeqs} &= E_1.\mathit{styeqs} \cup \{E_1.\mathit{type} \equiv \mathbb{P}(\alpha)\} \end{aligned}$$

pred-mult: $P ::= M_1$

Let α be a fresh type variable in

$$\begin{aligned} M_1.\mathit{ityvars} &= P.\mathit{ityvars} \cup \{\alpha\} \\ M_1.\mathit{ityenv} &= P.\mathit{ityenv} \\ M_1.\mathit{ityeqs} &= P.\mathit{ityeqs} \\ P.\mathit{styvars} &= M_1.\mathit{styvars} \\ P.\mathit{styenv} &= M_1.\mathit{styenv} \\ P.\mathit{styeqs} &= M_1.\mathit{styeqs} \cup \{M_1.\mathit{type} \equiv \mathbb{P}(\alpha)\} \end{aligned}$$

pred-rel: $P ::= E_1 \ E_2 \ [\mathit{pred\text{-}relop}]$

Let α be a fresh type variable in

$$\begin{aligned} E_1.\mathit{ityvars} &= P.\mathit{ityvars} \cup \{\alpha\} \\ E_1.\mathit{ityenv} &= P.\mathit{ityenv} \\ E_1.\mathit{ityeqs} &= P.\mathit{ityeqs} \\ E_2.\mathit{inherited} &= E_1.\mathit{synthesized} \\ P.\mathit{styvars} &= E_2.\mathit{styvars} \\ P.\mathit{styenv} &= E_2.\mathit{styenv} \\ P.\mathit{styeqs} &= E_2.\mathit{styeqs} \cup \mathcal{E} \end{aligned}$$

where $\mathcal{E}$ is defined in the following table.

$P.\mathit{pred\text{-}relop}$	$\mathcal{E}$
equal, notequal	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \alpha \\ E_2.\mathit{type} \equiv \alpha \end{array} \right\}$
lt, le, gt, ge	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{Z} \\ E_2.\mathit{type} \equiv \mathbb{Z} \end{array} \right\}$
in, notin	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \alpha \\ E_2.\mathit{type} \equiv \mathbb{P}(\alpha) \end{array} \right\}$
subset, notsubset, subseteq, notsubseteq	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha) \\ E_2.\mathit{type} \equiv \mathbb{P}(\alpha) \end{array} \right\}$

ident-list: $L ::= I_1\ I_2\ \ldots\ I_n$

$$\begin{aligned}
I_1.\mathit{inherited} &= L.\mathit{inherited}\\
I_2.\mathit{inherited} &= I_1.\mathit{synthesized}\\
&\vdots\\
I_n.\mathit{inherited} &= I_{n-1}.\mathit{synthesized}\\
L.\mathit{synthesized} &= I_n.\mathit{synthesized}
\end{aligned}$$

ident: $I ::= [\mathit{name}]$

if $I.\mathit{name} \in \mathrm{dom}(I.\mathit{ityenv})$ then

$$\begin{aligned}
I.\mathit{synthesized} &= I.\mathit{inherited}\\
I.\mathit{type} &= I.\mathit{ityenv}(I.\mathit{name})
\end{aligned}$$

else let α be a fresh type variable in

$$\begin{aligned}
I.\mathit{styvars} &= I.\mathit{ityvars} \cup \{\alpha\}\\
I.\mathit{styenv} &= I.\mathit{ityenv} \cup \{I.\mathit{name} \mapsto \alpha\}\\
I.\mathit{styeqs} &= I.\mathit{ityeqs}\\
I.\mathit{type} &= \alpha
\end{aligned}$$

expr-bin: $E ::= E_1\ E_2\ [\mathit{expr\text{-}binop}]$

Let α, β, γ and δ be distinct fresh type variables in

$$\begin{aligned}
E_1.\mathit{ityvars} &= E.\mathit{ityvars} \cup \{\alpha, \beta, \gamma, \delta\}\\
E_1.\mathit{ityenv} &= E.\mathit{ityenv}\\
E_1.\mathit{ityeqs} &= E.\mathit{ityeqs}\\
E_2.\mathit{inherited} &= E_1.\mathit{synthesized}\\
E.\mathit{styvars} &= E_2.\mathit{styvars}\\
E.\mathit{styenv} &= E_2.\mathit{styenv}\\
E.\mathit{styeqs} &= E_2.\mathit{styeqs} \cup \mathcal{E}\\
E.\mathit{type} &= \tau
\end{aligned}$$

where $\mathcal{E}$ and τ are defined in Table 4.1 on the following page.

expr-una: $E ::= E_1\ [\mathit{expr\text{-}unop}]$

Let α and β be distinct fresh type variables in

$$\begin{aligned}
E_1.\mathit{ityvars} &= E.\mathit{ityvars} \cup \{\alpha, \beta\}\\
E_1.\mathit{ityenv} &= E.\mathit{ityenv}\\
E_1.\mathit{ityeqs} &= E.\mathit{ityeqs}\\
E.\mathit{styvars} &= E_1.\mathit{styvars}\\
E.\mathit{styenv} &= E_1.\mathit{styenv}\\
E.\mathit{styeqs} &= E_1.\mathit{styeqs} \cup \mathcal{E}\\
E.\mathit{type} &= \tau
\end{aligned}$$

where $\mathcal{E}$ and τ are defined in Table 4.2 on page 33.

expr-lambda: $E ::= Q_1\ P_1\ E_1$

$$\begin{aligned}
Q_1.\mathit{inherited} &= E.\mathit{inherited}\\
P_1.\mathit{inherited} &= Q_1.\mathit{synthesized}\\
E_1.\mathit{inherited} &= P_1.\mathit{synthesized}\\
E.\mathit{synthesized} &= E_1.\mathit{synthesized}\\
E.\mathit{type} &= \mathbb{P}(Q_1.\mathit{type} \times E_1.\mathit{type})
\end{aligned}$$

$E.\mathit{expr\text{-}binop}$	$\mathcal{E}$	τ
funimage	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.\mathit{type} \equiv \alpha \end{array} \right\}$	β
relimage	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.\mathit{type} \equiv \mathbb{P}(\alpha) \end{array} \right\}$	$\mathbb{P}(\beta)$
mapsto	$\varnothing$	$E_1.\mathit{type} \times E_2.\mathit{type}$
rel, trel, srel, strel, pfun, tfun, pinj, tinj, psur, tsur, tbij	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha) \\ E_2.\mathit{type} \equiv \mathbb{P}(\beta) \end{array} \right\}$	$\mathbb{P}(\mathbb{P}(\alpha \times \beta))$
bunion, binter, setminus	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha) \\ E_2.\mathit{type} \equiv \mathbb{P}(\alpha) \end{array} \right\}$	$\mathbb{P}(\alpha)$
cprod	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha) \\ E_2.\mathit{type} \equiv \mathbb{P}(\beta) \end{array} \right\}$	$\mathbb{P}(\alpha \times \beta)$
dprod	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.\mathit{type} \equiv \mathbb{P}(\alpha \times \gamma) \end{array} \right\}$	$\mathbb{P}(\alpha \times (\beta \times \gamma))$
pprod	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha \times \gamma) \\ E_2.\mathit{type} \equiv \mathbb{P}(\beta \times \delta) \end{array} \right\}$	$\mathbb{P}((\alpha \times \beta) \times (\gamma \times \delta))$
bcomp	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\beta \times \gamma) \\ E_2.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta) \end{array} \right\}$	$\mathbb{P}(\alpha \times \gamma)$
fcomp	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.\mathit{type} \equiv \mathbb{P}(\beta \times \gamma) \end{array} \right\}$	$\mathbb{P}(\alpha \times \gamma)$
ovl	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta) \end{array} \right\}$	$\mathbb{P}(\alpha \times \beta)$
domres, domsub	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha) \\ E_2.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta) \end{array} \right\}$	$\mathbb{P}(\alpha \times \beta)$
ranres, ransub	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta) \\ E_2.\mathit{type} \equiv \mathbb{P}(\beta) \end{array} \right\}$	$\mathbb{P}(\alpha \times \beta)$
upto	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{Z} \\ E_2.\mathit{type} \equiv \mathbb{Z} \end{array} \right\}$	$\mathbb{P}(\mathbb{Z})$
plus, minus, mul, div, mod, expn	$\left\{ \begin{array}{l} E_1.\mathit{type} \equiv \mathbb{Z} \\ E_2.\mathit{type} \equiv \mathbb{Z} \end{array} \right\}$	$\mathbb{Z}$

Table 4.1: Typing equations and resulting type for binary expressions.

$E.\mathit{expr\text{-}unop}$	$\mathcal{E}$	τ
uminus	$\{\ E_1.\mathit{type} \equiv \mathbb{Z}\ \}$	$\mathbb{Z}$
converse	$\{\ E_1.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta)\ \}$	$\mathbb{P}(\beta \times \alpha)$
card	$\{\ E_1.\mathit{type} \equiv \mathbb{P}(\alpha)\ \}$	$\mathbb{Z}$
pow, pow1	$\{\ E_1.\mathit{type} \equiv \mathbb{P}(\alpha)\ \}$	$\mathbb{P}(\mathbb{P}(\alpha))$
union, inter	$\{\ E_1.\mathit{type} \equiv \mathbb{P}(\mathbb{P}(\alpha))\ \}$	$\mathbb{P}(\alpha)$
dom	$\{\ E_1.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta)\ \}$	$\mathbb{P}(\alpha)$
ran	$\{\ E_1.\mathit{type} \equiv \mathbb{P}(\alpha \times \beta)\ \}$	$\mathbb{P}(\beta)$
min, max	$\{\ E_1.\mathit{type} \equiv \mathbb{P}(\mathbb{Z})\ \}$	$\mathbb{Z}$

Table 4.2: Typing equations and resulting type for unary expressions.

expr-quant1: $E ::= L_1\ P_1\ E_1\ [\mathit{expr\text{-}quant}]$

Let α be a fresh type variable in

$$\begin{aligned}
L_1.\mathit{ityvars} &= E.\mathit{ityvars} \cup \{\alpha\}\\
L_1.\mathit{ityenv} &= E.\mathit{ityenv}\\
L_1.\mathit{ityeqs} &= E.\mathit{ityeqs}\\
P_1.\mathit{inherited} &= L_1.\mathit{synthesized}\\
E_1.\mathit{inherited} &= P_1.\mathit{synthesized}\\
E.\mathit{styvars} &= E_1.\mathit{styvars}\\
E.\mathit{styenv} &= E_1.\mathit{styenv}\\
E.\mathit{styeqs} &= E_1.\mathit{styeqs} \cup \mathcal{E}\\
E.\mathit{type} &= \tau
\end{aligned}$$

where $\mathcal{E}$ and τ are defined in the following table.

$E.\mathit{expr\text{-}quant}$	$\mathcal{E}$	τ
qunion, qinter	$\{\ E_1.\mathit{type} \equiv \mathbb{P}(\alpha)\ \}$	$\mathbb{P}(\alpha)$
cset	$\varnothing$	$\mathbb{P}(E_1.\mathit{type})$

expr-quant2: $E ::= E_1\ P_1\ [\mathit{expr\text{-}quant}]$
Let α be a fresh type variable in

$$\begin{aligned}
E_1.\mathit{ityvars} &= E.\mathit{ityvars} \cup \{\alpha\} \\
E_1.\mathit{ityenv} &= E.\mathit{ityenv} \\
E_1.\mathit{ityeqs} &= E.\mathit{ityeqs} \\
P_1.\mathit{inherited} &= E_1.\mathit{synthesized} \\
E.\mathit{styvars} &= P_1.\mathit{styvars} \\
E.\mathit{styenv} &= P_1.\mathit{styenv} \\
E.\mathit{styeqs} &= P_1.\mathit{styeqs} \cup \mathcal{E} \\
E.\mathit{type} &= \tau
\end{aligned}$$

where $\mathcal{E}$ and τ are defined in the following table.

$E.\mathit{expr\text{-}quant}$	$\mathcal{E}$	τ
qunion, qinter	$\{\ E_1.\mathit{type} \equiv \mathbb{P}(\alpha)\ \}$	$\mathbb{P}(\alpha)$
cset	$\varnothing$	$\mathbb{P}(E_1.\mathit{type})$

expr-bool: $E ::= P_1$

$$\begin{aligned}
P_1.\mathit{inherited} &= E.\mathit{inherited} \\
E.\mathit{synthesized} &= P_1.\mathit{synthesized} \\
E.\mathit{type} &= \mathrm{BOOL}
\end{aligned}$$

expr-eset: $E ::= M_1$

$$\begin{aligned}
M_1.\mathit{inherited} &= E.\mathit{inherited} \\
E.\mathit{synthesized} &= M_1.\mathit{synthesized} \\
E.\mathit{type} &= \mathbb{P}(M_1.\mathit{type})
\end{aligned}$$

expr-ident: $E ::= I_1$

$$\begin{aligned}
I_1.\mathit{inherited} &= E.\mathit{inherited} \\
E.\mathit{synthesized} &= I_1.\mathit{synthesized} \\
E.\mathit{type} &= I_1.\mathit{type}
\end{aligned}$$

expr-atom: $E ::= [\mathit{expr\text{-}lit}]$

Let α and β be distinct fresh type variables in

$$\begin{aligned} E.\mathit{styvars} &= E.\mathit{ityvars} \cup \{\alpha, \beta\} \\ E.\mathit{styenv} &= E.\mathit{ityenv} \\ E.\mathit{styeqs} &= E.\mathit{ityeqs} \\ E.\mathit{type} &= \tau \end{aligned}$$

where τ is defined in the following table.

$E.\mathit{expr\text{-}lit}$	τ
integer, natural, natural1	$\mathbb{P}(\mathbb{Z})$
pred, succ	$\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$
bool	$\mathbb{P}(\mathrm{BOOL})$
true, false	BOOL
emptyset	$\mathbb{P}(\alpha)$
id	$\mathbb{P}(\alpha \times \alpha)$
prj1	$\mathbb{P}(\alpha \times \beta \times \alpha)$
prj2	$\mathbb{P}(\alpha \times \beta \times \beta)$

expr-int: $E ::= [\mathit{int\text{-}lit}]$

$$\begin{aligned} E.\mathit{synthesized} &= E.\mathit{inherited} \\ E.\mathit{type} &= \mathbb{Z} \end{aligned}$$

pattern: $Q ::= Q_1\ Q_2$

$$\begin{aligned} Q_1.\mathit{inherited} &= Q.\mathit{inherited} \\ Q_2.\mathit{inherited} &= Q_1.\mathit{synthesized} \\ Q.\mathit{synthesized} &= Q_2.\mathit{synthesized} \\ Q.\mathit{type} &= Q_1.\mathit{type} \times Q_2.\mathit{type} \end{aligned}$$

pattern-ident: $Q ::= I_1$

$$\begin{aligned} I_1.\mathit{inherited} &= Q.\mathit{inherited} \\ Q.\mathit{synthesized} &= I_1.\mathit{synthesized} \\ Q.\mathit{type} &= I_1.\mathit{type} \end{aligned}$$

expr-list: $M ::= E_1\ E_2\ \ldots\ E_n$

$$\begin{aligned}
E_1.\mathit{inherited} &= M.\mathit{inherited}\\
E_2.\mathit{inherited} &= E_1.\mathit{synthesized}\\
&\vdots\\
E_n.\mathit{inherited} &= E_{n-1}.\mathit{synthesized}\\
M.\mathit{styvars} &= E_n.\mathit{ityvars}\\
M.\mathit{styenv} &= E_n.\mathit{ityenv}\\
M.\mathit{styeqs} &= E_n.\mathit{ityeqs} \cup \left\{\begin{array}{ll} E_1.\mathit{type} & \equiv E_2.\mathit{type}\\ E_2.\mathit{type} & \equiv E_3.\mathit{type}\\ & \vdots\\ E_{n-1}.\mathit{type} & \equiv E_n.\mathit{type}\end{array}\right\}\\
M.\mathit{type} &= E_n.\mathit{type}
\end{aligned}$$

4.3.3 Examples

In this subsection, we present a few examples of the type-checking algorithm in action on various formulae.

Formula $x \in \mathbb{Z} \land 1 \leq x$. Figure 4.1 shows the dataflow for the type-checking of this formula. Each step of the type-checking algorithm is shown as a circled number, with edges relating them. The numbers appearing on the left of a node corresponds to the computation of inherited attributes, numbers on the right to the computation of synthesized attributes.

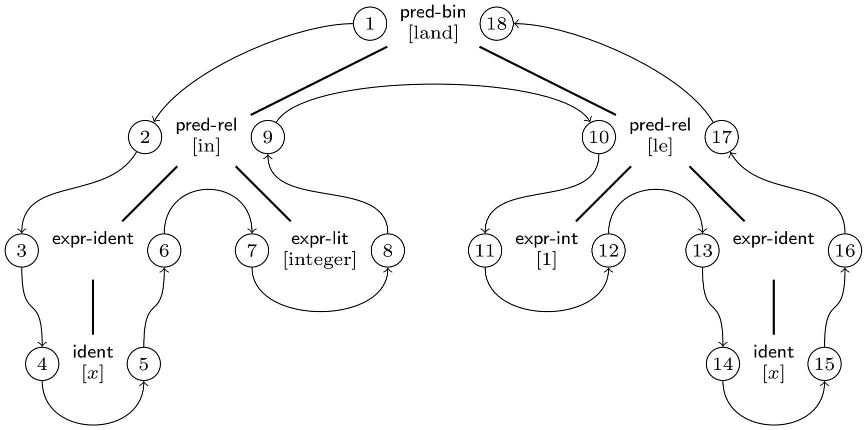

Figure 4.1: Type-check of formula $x \in \mathbb{Z} \land 1 \leq x$.

Assuming that the typing environment is initially empty, the initial computation at step 1 is:

$$1\!:\ \left|\ \begin{aligned} \mathit{ityvars} &= \varnothing\\ \mathit{ityenv} &= \varnothing\\ \mathit{ityeqs} &= \varnothing \end{aligned}\right.$$

Then, we process down the tree, adding a type variable at the $\in$ operator:

$$2:\left|\begin{array}{lcl} \mathit{ityvars} &=& \varnothing \\ \mathit{ityenv} &=& \varnothing \\ \mathit{ityeqs} &=& \varnothing \end{array}\right. \qquad\qquad 3,\ 4:\left|\begin{array}{lcl} \mathit{ityvars} &=& \{\alpha\} \\ \mathit{ityenv} &=& \varnothing \\ \mathit{ityeqs} &=& \varnothing \end{array}\right.$$

Examining the first occurrence of variable x, we find that it is not present in the environment, so we create a new type variable for it. This is then propagated in the tree:

$$5,\ 6:\left|\begin{array}{lcl} \mathit{styvars} &=& \{\alpha, \beta\} \\ \mathit{styenv} &=& \{x \mapsto \beta\} \\ \mathit{styeqs} &=& \varnothing \\ \mathit{type} &=& \beta \end{array}\right. \quad 7:\left|\begin{array}{lcl} \mathit{ityvars} &=& \{\alpha, \beta\} \\ \mathit{ityenv} &=& \{x \mapsto \beta\} \\ \mathit{ityeqs} &=& \varnothing \end{array}\right. \quad 8:\left|\begin{array}{lcl} \mathit{styvars} &=& \{\alpha, \beta\} \\ \mathit{styenv} &=& \{x \mapsto \beta\} \\ \mathit{styeqs} &=& \varnothing \\ \mathit{type} &=& \mathbb{P}(\mathbb{Z}) \end{array}\right.$$

We now reach the $\in$ operator again, where we add our first type equations and propagate the attribute values:

$$9:\left|\begin{array}{lcl} \mathit{styvars} &=& \{\alpha, \beta\} \\ \mathit{styenv} &=& \{x \mapsto \beta\} \\ \mathit{styeqs} &=& \left\{\begin{array}{ll} \beta & \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) & \equiv \mathbb{P}(\alpha) \end{array}\right\} \end{array}\right. \qquad 10,\ 11:\left|\begin{array}{lcl} \mathit{ityvars} &=& \{\alpha, \beta, \gamma\} \\ \mathit{ityenv} &=& \{x \mapsto \beta\} \\ \mathit{ityeqs} &=& \left\{\begin{array}{ll} \beta & \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) & \equiv \mathbb{P}(\alpha) \end{array}\right\} \end{array}\right.$$

Continuing our traversal of the tree, we get:

$$12:\left|\begin{array}{lcl} \mathit{styvars} &=& \{\alpha, \beta, \gamma\} \\ \mathit{styenv} &=& \{x \mapsto \beta\} \\ \mathit{styeqs} &=& \left\{\begin{array}{ll} \beta & \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) & \equiv \mathbb{P}(\alpha) \end{array}\right\} \\ \mathit{type} &=& \mathbb{Z} \end{array}\right. \qquad 13,\ 14:\left|\begin{array}{lcl} \mathit{ityvars} &=& \{\alpha, \beta, \gamma\} \\ \mathit{ityenv} &=& \{x \mapsto \beta\} \\ \mathit{ityeqs} &=& \left\{\begin{array}{ll} \beta & \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) & \equiv \mathbb{P}(\alpha) \end{array}\right\} \end{array}\right.$$

We now reach the second occurrence of variable x and, now, it is present in the typing environment, so we just read its type from there, and propagate it:

$$15,\ 16:\left|\begin{array}{lcl} \mathit{styvars} &=& \{\alpha, \beta, \gamma\} \\ \mathit{styenv} &=& \{x \mapsto \beta\} \\ \mathit{styeqs} &=& \left\{\begin{array}{ll} \beta & \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) & \equiv \mathbb{P}(\alpha) \end{array}\right\} \\ \mathit{type} &=& \beta \end{array}\right.$$

Reaching operator $\leq$, we add two new typing equations and propagate them to the root:

$$17,\ 18:\left|\begin{array}{lcl} \mathit{styvars} &=& \{\alpha, \beta, \gamma\} \\ \mathit{styenv} &=& \{x \mapsto \beta\} \\ \mathit{styeqs} &=& \left\{\begin{array}{ll} \beta & \equiv \alpha, \\ \mathbb{P}(\mathbb{Z}) & \equiv \mathbb{P}(\alpha) \\ \mathbb{Z} & \equiv \mathbb{Z} \\ \beta & \equiv \mathbb{Z} \end{array}\right\} \end{array}\right.$$

In the end, we obtain a system of four typing equations with two type variables. This system is uniquely satisfiable by taking $\alpha = \mathbb{Z}$ and $\beta = \mathbb{Z}$. Hence, the formula type checks. Moreover, its resulting typing environment is $\{x \mapsto \mathbb{Z}\}$.

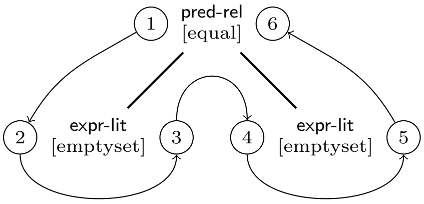

Figure 4.2: Type-check of formula $\varnothing = \varnothing$.

Formula $\varnothing = \varnothing$. The type-checking dataflow for this formula is given in Figure 4.2.

The attribute values computed by the algorithm are (supposing that the initial typing environment is empty):

$$1:\left|\begin{array}{l} \mathit{ityvars} = \varnothing \\ \mathit{ityenv} \ = \varnothing \\ \mathit{ityeqs} \ \ = \varnothing \end{array}\right. \qquad 2:\left|\begin{array}{l} \mathit{ityvars} = \{\alpha\} \\ \mathit{ityenv} \ = \varnothing \\ \mathit{ityeqs} \ \ = \varnothing \end{array}\right. \qquad 3:\left|\begin{array}{l} \mathit{styvars} = \{\alpha, \beta\} \\ \mathit{styenv} \ = \varnothing \\ \mathit{styeqs} \ \ = \varnothing \\ \mathit{type} \ \ \ \ = \beta \end{array}\right.$$

$$4:\left|\begin{array}{l} \mathit{ityvars} = \{\alpha, \beta\} \\ \mathit{ityenv} \ = \varnothing \\ \mathit{ityeqs} \ \ = \varnothing \end{array}\right. \qquad 5:\left|\begin{array}{l} \mathit{styvars} = \{\alpha, \beta, \gamma\} \\ \mathit{styenv} \ = \varnothing \\ \mathit{styeqs} \ \ = \varnothing \\ \mathit{type} \ \ \ \ = \gamma \end{array}\right. \qquad 6:\left|\begin{array}{l} \mathit{styvars} = \{\alpha, \beta, \gamma\} \\ \mathit{styenv} \ = \varnothing \\ \mathit{styeqs} \ \ = \left\{\begin{array}{l} \beta \equiv \alpha, \\ \gamma \equiv \alpha \end{array}\right\} \end{array}\right.$$

In the end, we obtain a system of two typing equations with three typing variables. This system is satisfiable, but not uniquely. Hence formula $\varnothing = \varnothing$ does not type-check.

Formula $x \subseteq S \ \land \ \varnothing \subset x$. The type-checking dataflow for this formula is given in Figure 4.3.

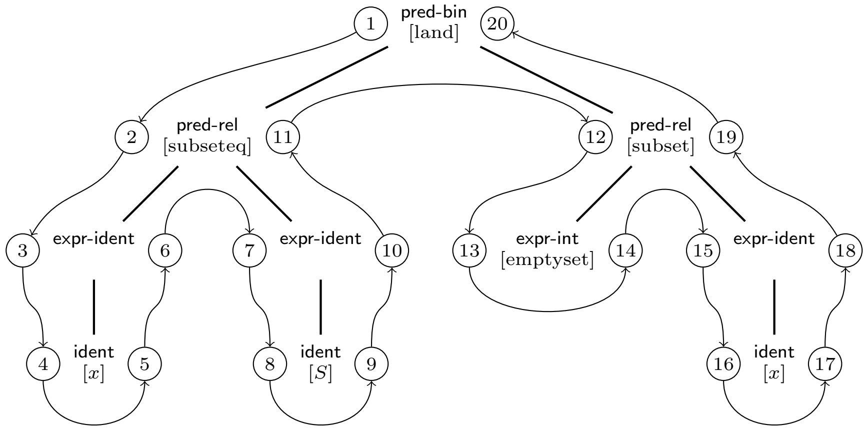

Figure 4.3: Type-check of formula $x \subseteq S \ \land \ \varnothing \subset x$.

Here, we assume that variable S denotes a given set. Thus, our initial typing environment is $\{S \mapsto \mathbb{P}(S)\}$. The attribute values computed by the type-checking algorithm are:

$$1,\ 2{:}\left|\begin{array}{lcl}\mathit{ityvars} &=& \varnothing\\ \mathit{ityenv} &=& \{S \mapsto \mathbb{P}(S)\}\\ \mathit{ityeqs} &=& \varnothing\end{array}\right. \qquad 3,\ 4{:}\left|\begin{array}{lcl}\mathit{ityvars} &=& \{\alpha\}\\ \mathit{ityenv} &=& \{S \mapsto \mathbb{P}(S)\}\\ \mathit{ityeqs} &=& \varnothing\end{array}\right.$$

$$5,\ 6{:}\left|\begin{array}{lcl}\mathit{styvars} &=& \{\alpha, \beta\}\\ \mathit{styenv} &=& \left\{\begin{array}{l}S \mapsto \mathbb{P}(S),\\ x \mapsto \beta\end{array}\right\}\\ \mathit{styeqs} &=& \varnothing\\ \mathit{type} &=& \beta\end{array}\right. \qquad 7,\ 8{:}\left|\begin{array}{lcl}\mathit{ityvars} &=& \{\alpha, \beta\}\\ \mathit{ityenv} &=& \left\{\begin{array}{l}S \mapsto \mathbb{P}(S),\\ x \mapsto \beta\end{array}\right\}\\ \mathit{ityeqs} &=& \varnothing\end{array}\right.$$

$$9,\ 10{:}\left|\begin{array}{lcl}\mathit{styvars} &=& \{\alpha, \beta\}\\ \mathit{styenv} &=& \left\{\begin{array}{l}S \mapsto \mathbb{P}(S),\\ x \mapsto \beta\end{array}\right\}\\ \mathit{styeqs} &=& \varnothing\\ \mathit{type} &=& \mathbb{P}(S)\end{array}\right. \qquad 11{:}\left|\begin{array}{lcl}\mathit{styvars} &=& \{\alpha, \beta\}\\ \mathit{styenv} &=& \left\{\begin{array}{l}S \mapsto \mathbb{P}(S),\\ x \mapsto \beta\end{array}\right\}\\ \mathit{styeqs} &=& \left\{\begin{array}{lcl}\beta &\equiv& \mathbb{P}(\alpha),\\ \mathbb{P}(S) &\equiv& \mathbb{P}(\alpha)\end{array}\right\}\end{array}\right.$$

$$12{:}\left|\begin{array}{lcl}\mathit{ityvars} &=& \{\alpha, \beta\}\\ \mathit{ityenv} &=& \left\{\begin{array}{l}S \mapsto \mathbb{P}(S),\\ x \mapsto \beta\end{array}\right\}\\ \mathit{ityeqs} &=& \left\{\begin{array}{lcl}\beta &\equiv& \mathbb{P}(\alpha),\\ \mathbb{P}(S) &\equiv& \mathbb{P}(\alpha)\end{array}\right\}\end{array}\right. \qquad 13{:}\left|\begin{array}{lcl}\mathit{ityvars} &=& \{\alpha, \beta, \gamma\}\\ \mathit{ityenv} &=& \left\{\begin{array}{l}S \mapsto \mathbb{P}(S),\\ x \mapsto \beta\end{array}\right\}\\ \mathit{ityeqs} &=& \left\{\begin{array}{lcl}\beta &\equiv& \mathbb{P}(\alpha),\\ \mathbb{P}(S) &\equiv& \mathbb{P}(\alpha)\end{array}\right\}\end{array}\right.$$

$$14{:}\left|\begin{array}{lcl}\mathit{styvars} &=& \{\alpha, \beta, \gamma, \delta\}\\ \mathit{styenv} &=& \left\{\begin{array}{l}S \mapsto \mathbb{P}(S),\\ x \mapsto \beta\end{array}\right\}\\ \mathit{styeqs} &=& \left\{\begin{array}{lcl}\beta &\equiv& \mathbb{P}(\alpha),\\ \mathbb{P}(S) &\equiv& \mathbb{P}(\alpha)\end{array}\right\}\\ \mathit{type} &=& \mathbb{P}(\delta)\end{array}\right. \qquad 15,\ 16{:}\left|\begin{array}{lcl}\mathit{ityvars} &=& \{\alpha, \beta, \gamma, \delta\}\\ \mathit{ityenv} &=& \left\{\begin{array}{l}S \mapsto \mathbb{P}(S),\\ x \mapsto \beta\end{array}\right\}\\ \mathit{ityeqs} &=& \left\{\begin{array}{lcl}\beta &\equiv& \mathbb{P}(\alpha),\\ \mathbb{P}(S) &\equiv& \mathbb{P}(\alpha)\end{array}\right\}\end{array}\right.$$

$$17,\ 18{:}\left|\begin{array}{lcl}\mathit{styvars} &=& \{\alpha, \beta, \gamma, \delta\}\\ \mathit{styenv} &=& \left\{\begin{array}{l}S \mapsto \mathbb{P}(S),\\ x \mapsto \beta\end{array}\right\}\\ \mathit{styeqs} &=& \left\{\begin{array}{lcl}\beta &\equiv& \mathbb{P}(\alpha),\\ \mathbb{P}(S) &\equiv& \mathbb{P}(\alpha)\end{array}\right\}\\ \mathit{type} &=& \beta\end{array}\right. \qquad 19,\ 20{:}\left|\begin{array}{lcl}\mathit{styvars} &=& \{\alpha, \beta, \gamma, \delta\}\\ \mathit{styenv} &=& \left\{\begin{array}{l}S \mapsto \mathbb{P}(S),\\ x \mapsto \beta\end{array}\right\}\\ \mathit{styeqs} &=& \left\{\begin{array}{lcl}\beta &\equiv& \mathbb{P}(\alpha),\\ \mathbb{P}(S) &\equiv& \mathbb{P}(\alpha),\\ \mathbb{P}(\delta) &\equiv& \mathbb{P}(\gamma),\\ \beta &\equiv& \mathbb{P}(\gamma)\end{array}\right\}\end{array}\right.$$

In the end, we obtain a system of four typing equations with four typing variables. This system is uniquely satisfiable taking $\alpha = \gamma = \delta = S$ and $\beta = \mathbb{P}(S)$. Hence formula $x \subseteq S \ \wedge\ \varnothing \subset x$ type-checks and the resulting typing environment is $\{S \mapsto \mathbb{P}(S), x \mapsto \mathbb{P}(S)\}$.

Formula $x = \mathrm{TRUE}$. The type-checking dataflow for this formula is given in Figure 4.4 on the next page.

Assuming that initially x denotes an integer (non empty initial typing environment), we obtain the following values for attributes:

$$1{:}\left|\begin{array}{lcl}\mathit{ityvars} &=& \varnothing\\ \mathit{ityenv} &=& \{x \mapsto \mathbb{Z}\}\\ \mathit{ityeqs} &=& \varnothing\end{array}\right. \qquad 2,\ 3{:}\left|\begin{array}{lcl}\mathit{ityvars} &=& \{\alpha\}\\ \mathit{ityenv} &=& \{x \mapsto \mathbb{Z}\}\\ \mathit{ityeqs} &=& \varnothing\end{array}\right.$$

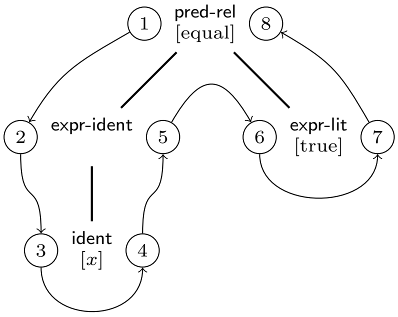

Figure 4.4: Type-check of formula $x = \text{TRUE}$.

$$4,\ 5\colon \left|\begin{array}{ll} \mathit{styvars} & = \{\alpha\} \\ \mathit{styenv} & = \{x \mapsto \mathbb{Z}\} \\ \mathit{styeqs} & = \varnothing \\ \mathit{type} & = \mathbb{Z} \end{array}\right. \qquad 6\colon \left|\begin{array}{ll} \mathit{ityvars} & = \{\alpha\} \\ \mathit{ityenv} & = \{x \mapsto \mathbb{Z}\} \\ \mathit{ityeqs} & = \varnothing \end{array}\right.$$

$$7\colon \left|\begin{array}{ll} \mathit{styvars} & = \{\alpha\} \\ \mathit{styenv} & = \{x \mapsto \mathbb{Z}\} \\ \mathit{styeqs} & = \varnothing \\ \mathit{type} & = \text{BOOL} \end{array}\right. \qquad 8\colon \left|\begin{array}{ll} \mathit{styvars} & = \{\alpha\} \\ \mathit{styenv} & = \{x \mapsto \mathbb{Z}\} \\ \mathit{styeqs} & = \left\{\begin{array}{ll} \mathbb{Z} & \equiv \alpha \\ \text{BOOL} & \equiv \alpha \end{array}\right\} \end{array}\right.$$

In the end, we obtain a system of two typing equations with one typing variable. This system is not satisfiable, therefore the formula does not type-check (remember that we initially assumed that variable x denotes an integer). If the initial typing environment would have been empty, then the formula would type-check.

5 Dynamic Checking

Static checks are not enough to ensure that a formula is meaningful. For instance, expression $x \div y$ passes all the static checks described above, nevertheless it is meaningless if y is zero. The aim of dynamic checking [2, 3] is to detect these kind of meaningless formulas. This is done by generating (and then proving) some well-definedness lemma.

The rest of this chapter specifies how to produce these well-definedness lemmas. This is done by specifying a WD operator that takes a formula as argument and the result of which is the well-definedness lemma of that formula.

5.1 Predicate Well-Definedness

Table 5.1 on the following page specifies the WD operator for predicates. In that table, letters P and Q denote arbitrary predicates, letters E and F denote expressions, and letter L denotes a list of identifiers.

5.2 Expression Well-Definedness

Tables 5.2 on page 43 and 5.3 on page 44 specify the WD operator for expressions. In these tables, letter P denotes an arbitrary predicate, letters E and F denote expressions, letter Q denotes a lambda pattern, letter L denotes a list of identifiers, letter I denotes an identifier, letter n denotes a literal integer. We also denote by $\mathcal{F}_E$ the list of the free variables that appear in expression E (that is $E.\mathit{free}$) and by $\mathcal{F}_Q$ the list of the free variables that appear in pattern Q. Finally, letter x denotes a fresh variable (that is a variable that does not occur free in the formula for which we compute the well-definedness lemma).

Predicate	**WD Lemma**
$P \wedge Q \qquad P \Rightarrow Q$	$\mathsf{WD}(P) \wedge (P \Rightarrow \mathsf{WD}(Q))$
$P \vee Q$	$\mathsf{WD}(P) \wedge (P \vee \mathsf{WD}(Q))$
$P \Leftrightarrow Q$	$\mathsf{WD}(P) \wedge \mathsf{WD}(Q)$
$\neg P$	$\mathsf{WD}(P)$
$\forall L {\cdot} P \qquad \exists L {\cdot} P$	$\forall L {\cdot}\, \mathsf{WD}(P)$
$\top \qquad \bot$	$\top$
$\text{finite}(E)$	$\mathsf{WD}(E)$
$\text{partition}(E_1, E_2, \ldots, E_n)$	$\mathsf{WD}(E_1) \wedge \mathsf{WD}(E_2) \wedge \cdots \wedge \mathsf{WD}(E_n)$
$E = F \qquad E \neq F$ $E \in F \qquad E \notin F$ $E \subset F \qquad E \not\subset F$ $E \subseteq F \qquad E \not\subseteq F$	$\mathsf{WD}(E) \wedge \mathsf{WD}(F)$

Table 5.1: WD lemmas for predicates.

Expression	**WD Lemma**
$F(E)$	$\mathsf{WD}(F) \land \mathsf{WD}(E)$ $\land\ E \in \mathrm{dom}(F) \land F \in S \pfun T$ assuming $F.\mathit{type} \equiv \mathbb{P}(S \times T)$
$E[F] \qquad E \mapsto F$ $E \rel F \qquad E \trel F$ $E \srel F \qquad E \strel F$ $E \pfun F \qquad E \tfun F$ $E \pinj F \qquad E \tinj F$ $E \psur F \qquad E \tsur F$ $E \tbij F \qquad E \cup F$ $E \cap F \qquad E \setminus F$ $E \times F \qquad E \otimes F$ $E \parallel F \qquad E \circ F$ $E \,;\, F \qquad E \ovl F$ $E \lhd F \qquad E \domsub F$ $E \rhd F \qquad E \ransub F$ $E \,..\, F \qquad E + F$ $E - F \qquad E * F$	$\mathsf{WD}(E) \land \mathsf{WD}(F)$
$E \div F \qquad E \bmod F$	$\mathsf{WD}(E) \land \mathsf{WD}(F) \land F \neq 0$
$E \,\hat{}\, F$	$\mathsf{WD}(E) \land 0 \leq E \land \mathsf{WD}(F) \land 0 \leq F$
$-E \qquad E^{-1}$ $\mathbb{P}(E) \qquad \mathbb{P}_1(E)$ $\mathrm{dom}(E) \qquad \mathrm{ran}(E)$ $\mathrm{union}(E)$	$\mathsf{WD}(E)$
$\mathrm{card}(E)$	$\mathsf{WD}(E) \land \mathrm{finite}(E)$
$\mathrm{inter}(E)$	$\mathsf{WD}(E) \land E \neq \varnothing$
$\min(E)$	$\mathsf{WD}(E) \land E \neq \varnothing \land (\exists b \cdot \forall x \cdot x \in E \Rightarrow b \leq x)$
$\max(E)$	$\mathsf{WD}(E) \land E \neq \varnothing \land (\exists b \cdot \forall x \cdot x \in E \Rightarrow x \leq b)$

Table 5.2: WD lemmas for binary and unary expressions.

Expression	**WD Lemma**
$\lambda Q \cdot P \mid E$	$\forall \mathcal{F}_Q \cdot \ \mathsf{WD}(P) \wedge (P \Rightarrow \mathsf{WD}(E))$
$\bigcup L \cdot P \mid E$ $\{L \cdot P \mid E\}$	$\forall L \cdot \ \mathsf{WD}(P) \wedge (P \Rightarrow \mathsf{WD}(E))$
$\bigcup E \mid P$ $\{E \mid P\}$	$\forall \mathcal{F}_E \cdot \ \mathsf{WD}(P) \wedge (P \Rightarrow \mathsf{WD}(E))$
$\bigcap L \cdot P \mid E$	$(\forall L \cdot \ \mathsf{WD}(P) \wedge (P \Rightarrow \mathsf{WD}(E)))$ $\wedge \quad (\exists L \cdot P)$
$\bigcap E \mid P$	$(\forall \mathcal{F}_E \cdot \ \mathsf{WD}(P) \wedge (P \Rightarrow \mathsf{WD}(E)))$ $\wedge \quad (\exists \mathcal{F}_E \cdot P)$
$\mathrm{bool}(P)$	$\mathsf{WD}(P)$
$\{E_1, E_2, \ldots, E_n\}$	$\mathsf{WD}(E_1) \wedge \mathsf{WD}(E_2) \wedge \cdots \wedge \mathsf{WD}(E_n)$
I $\quad \mathbb{Z}$ $\mathbb{N}$ $\quad \mathbb{N}_1$ pred $\quad$ succ BOOL $\quad$ TRUE FALSE $\quad \varnothing$ prj_1 $\quad \mathrm{prj}_2$ id $\quad n$	$\top$

Table 5.3: WD lemmas for other expressions.

Bibliography

[1] Abrial, J.-R. (1996). *The B-Book. Assigning Programs to Meanings.* Cambridge University Press.

[2] Abrial, J.-R and Mussat, L. (2002). *On Using Conditional Definitions in Formal Theories.* In D. Bert et al. (Eds), *ZB2002: Formal Specification and Development in Z and B*, LNCS 2272, pp. 242–269, Springer-Verlag.

[3] Burdy, L. (2000). *Traitement des expressions dépourvues de sens de la théorie des ensembles. Application à la méthode B.* Thèse de doctorat. Conservatoire National des Arts et Métiers.

[4] The Unicode Consortium (2003). *The Unicode Standard 4.0.* Addison-Wesley.