On Typechecking B

Antti-Juhani Kaijanaho

University of Jyväskylä
Department of Mathematical Information Technology
PO BOX 35 (Agora)
40014 University of Jyväskylä
FINLAND
`antkaij@mit.jyu.fi`

Abstract. The typechecking system of the formal method B is discussed. An inconsistency in the public definition of the B method, attributable to a flaw in the typechecking system, is uncovered: the typechecking method expects the types of variables to be given in one membership predicate, such as $a, b, c \in A \times B \times C$, instead of with several membership predicates joined by conjunction, like $a \in A \wedge b \in B \wedge c \in C$, even though such constructions are liberally used in the literature. A set of modifications to the typechecking system fixing this flaw is presented and analyzed.

1 Introduction

The B method [1] is a formal method for software construction, supporting specification, refinement and implementation, all in the same language. Proof-based verification of all specifications, refinements and implementations is supported by software-based tools.

The author started in late 2001 to build new, free[1] software-based support tools for the B method, employing the cleanroom tactic, where only public documentation is used and no non-disclosable information is accepted. The tools are collectively called The Ebba Toolset, and its current state — not for general use — is visible at Savannah (http://www.nongnu.org/ebba/). During this effort, the author discovered several problems with the public definition of the B method. One of them, attributable to a flaw in the typechecking system, is discussed and elaborated in this paper.

This paper is organized as follows. Section 2 gives an overview of the B method. Section 3 demonstrates the problem under discussion. Section 4 describes the proposed remedy for the problem. Finally, in Sect. 5, the remedy is inspected and analyzed.

2 An Overview of B

The B method was given a comprehensive definition by Jean-Raymond Abrial in his B-Book [1]. The author has reformulated a large part of the definition in his

[1] Free as in speech, not as in beer.

2002 Master's Thesis [2], incorporating theoretical advances since the publication of the B-Book (for example Steve Dunne's work on generalized substitutions [3]) and correcting some of the problems reported in this paper. This section summarises the main parts of the definition as they are relevant to this paper.

B has a layered design. At the core there is first-order logic. On top of that a typed set theory is built. The notion of substitution inherent in predicate logic is the seed for the next layer, the Generalized Substitution Language (GSL), which is based on Dijkstra's wp-calculus [4]. Finally, an Abstract Machine Notation (AMN) is built on top of GSL.

We will in this paper mostly ignore the AMN and concentrate on the more foundational layers, especially set theory. A summary of the (abstract) syntax for the parts of B that we will discuss is given in Fig. 1, following the author's thesis [2]. Most of the rest of the notation can be defined using rewrite rules. The constructs given in the figure that are marked "derived" are associated with rewrite rules that can be used to remove them.

In this article, we will use an informal concrete representation of the abstract syntax, using parentheses for disambiguation. Following Abrial [1], we declare that the middle dot ($\cdot$) in quantifiers associates to the right and has the highest precedence, followed by the right-associative negation connective ($\neg$), the left-associative conjunction and disjunction connectives ($\wedge$ and $\vee$, which share precedence), the left-associative implication connective ($\Rightarrow$), and finally the left-associative equivalence connective ($\Leftrightarrow$). The precedence and associativity of other operators is not specified. We will use boldface to indicate metavariables; $\mathbf{p}$, $\mathbf{q}$ and $\mathbf{r}$ stand for predicates, $\mathbf{e}$ and $\mathbf{f}$ stand for expressions, $\mathbf{s}$ and $\mathbf{t}$ stand for substitutions, $\mathbf{E}$ stands for type assumptions and $\mathbf{S}$ and $\mathbf{T}$ stand for types.

The semantics of most of the predicate, variable and expression constructs in Fig. 1 should be self-explanatory. The rest is explained below:

- Predicates of the form $[\mathbf{x} := \mathbf{e}]\mathbf{p}$ and expressions of the form $[\mathbf{x} := \mathbf{e}]\mathbf{p}$ are instances of the familiar substitution operation of first-order logic.
- In general, a predicate of the form $[\mathbf{s}]\mathbf{p}$ denotes the weakest precondition for the generalized substitution $\mathbf{s}$ given the postcondition $\mathbf{p}$.
- Given a nonempty set $\mathbf{e}$, $\mathsf{choice}(\mathbf{e})$ denotes a deterministically but arbitrarily chosen element of the set $\mathbf{e}$.
- BIG is an infinite set, mostly only used to build natural numbers in the foundational theory.
- Set difference is denoted by "$-$".
- The nonatomic variable $\mathsf{frame}(\mathbf{s})$ contains exactly those identifiers which occur on the left side of an assignment substitution ($:=$) in $\mathbf{s}$.

The semantics of the substitutions are defined by Abrial [1] by a multitude of methods, generally by giving axioms for inferring with predicates of the form $[\mathbf{s}]\mathbf{p}$. The author [2] uses the following rewrite rules under the assumption that

$$
\begin{array}{lll}
\mathit{Predicate} = & \mathit{Predicate}, \text{'}\wedge\text{'}, \mathit{Predicate} & \\
\mid & \mathit{Predicate}, \text{'}\vee\text{'}, \mathit{Predicate} & \text{(* derived *)} \\
\mid & \mathit{Predicate}, \text{'}\Rightarrow\text{'}, \mathit{Predicate} & \\
\mid & \mathit{Predicate}, \text{'}\Leftrightarrow\text{'}, \mathit{Predicate} & \text{(* derived *)} \\
\mid & \text{'}\neg\text{'}, \mathit{Predicate} & \\
\mid & \text{'}\forall\text{'}, \mathit{Variable}, \text{'}\cdot\text{'}, \mathit{Predicate} & \\
\mid & \text{'}\exists\text{'}, \mathit{Variable}, \text{'}\cdot\text{'}, \mathit{Predicate} & \text{(* derived *)} \\
\mid & \text{'['}, \mathit{Substitution}, \text{']'}, \mathit{Predicate} & \text{(* derived *)} \\
\mid & \mathit{Expression}, \text{'='}, \mathit{Expression} & \\
\mid & \mathit{Expression}, \text{'}\in\text{'}, \mathit{Expression} & \\
\mid & \text{'}\mathsf{infinite}\text{'}, \text{'('}, \mathit{Expression}, \text{')'}; & \text{(* derived *)} \\
\mathit{Expression} = & \mathit{Variable} & \\
\mid & \text{'['}, \mathit{Variable}, \text{':='}, \mathit{Expression}, \text{']'}, \mathit{Expression} & \\
\mid & \mathit{Expression}, \text{','}, \mathit{Expression} & \\
\mid & \text{'}\mathsf{choice}\text{'}, \text{'('}, \mathit{Expression}, \text{')'} & \\
\mid & \mathit{Expression}, \text{'}\times\text{'}, \mathit{Expression} & \\
\mid & \text{'}\mathbb{P}\text{'},\text{'('}, \mathit{Expression}, \text{')'} & \\
\mid & \text{'}\{\text{'}, \mathit{Variable}, \text{'}\mid\text{'}, \mathit{Predicate}, \text{'}\}\text{'} & \\
\mid & \text{'}\mathsf{BIG}\text{'} & \\
\mid & \mathit{Expression}, \text{'}\cup\text{'}, \mathit{Expression} & \text{(* derived *)} \\
\mid & \mathit{Expression}, \text{'}\cap\text{'}, \mathit{Expression} & \text{(* derived *)} \\
\mid & \mathit{Expression}, \text{'}-\text{'}, \mathit{Expression} & \text{(* derived *)} \\
\mid & \text{'}\{\text{'}, \mathit{Expression}, \text{'}\}\text{'}; & \text{(* derived *)} \\
\mathit{Variable} = & \mathit{Identifier} & \\
\mid & \mathit{Variable}, \text{','}, \mathit{Variable} & \\
\mid & \text{'}\mathsf{frame}\text{'}, \text{'('}, \mathit{Substitution}, \text{')'}; & \text{(* derived *)} \\
\mathit{Substitution} = & \mathit{Variable}, \text{':='}, \mathit{Expression} & \\
\mid & \mathit{Predicate}, \text{'}\mid\text{'}, \mathit{Substitution} & \\
\mid & \mathit{Predicate}, \text{'}\Longrightarrow\text{'}, \mathit{Substitution} & \\
\mid & \mathit{Substitution}, \text{'[]'}, \mathit{Subsitution} & \\
\mid & \text{'@'}, \mathit{Variable}, \text{'}\cdot\text{'}, \mathit{Substitution} & \\
\mid & \mathit{Substitution}, \text{';'}, \mathit{Substitution} & \\
\mid & \text{'}\mathsf{skip}\text{'}; & \\
\mathit{Type} = & \text{'}\mathsf{type}\text{'}, \text{'('}, \mathit{Expression}, \text{')'} & \\
\mid & \text{'}\mathsf{super}\text{'}, \text{'('}, \mathit{Expression}, \text{')'} & \\
\mid & \mathit{Type}, \text{'}\times\text{'}, \mathit{Type} & \\
\mid & \text{'}\mathbb{P}\text{'}, \text{'('}, \mathit{Type}, \text{')'}; & \\
\mathit{Type\ predicate} = & \text{'}\mathsf{check}\text{'}, \text{'('}, \mathit{Predicate}, \text{')'} & \\
\mid & \mathit{Type}, \text{'}\equiv\text{'}, \mathit{Type}; & \\
\mathit{Type\ assumption} = & \text{'}\mathsf{given}\text{'}, \text{'('}, (\mathit{Identifier} \mid \text{'}\mathsf{BIG}\text{'}), \text{')'} & \\
\mid & \mathit{Identifier}, \text{'}\in\text{'}, \mathit{Expression}; & \\
\mathit{Type\ sequent} = & [\mathit{Type\ assumption}, \{\mathit{Type\ assumption}\}], \text{'}\vdash\text{'}, \mathit{Type\ predicate}; & \\
\mathit{Type\ rule} = & \dfrac{\{\mathit{Type\ sequent}\}}{\mathit{Type\ sequent}}; &
\end{array}
$$

Fig. 1. Abstract syntax for basic set notation, types and GSL.

substitutions are never used except as a part of predicates:[2]

$$[\mathbf{p} \mid \mathbf{s}]\mathbf{q} :\Rightarrow \mathbf{p} \wedge [\mathbf{s}]\mathbf{q} \tag{1}$$
$$[\mathbf{p} \Longrightarrow \mathbf{s}]\mathbf{q} :\Rightarrow \mathbf{p} \Rightarrow [\mathbf{s}]\mathbf{q} \tag{2}$$
$$[\mathbf{s} \mathbin{[]} \mathbf{t}]\mathbf{q} :\Rightarrow [\mathbf{s}]\mathbf{q} \wedge [\mathbf{t}]\mathbf{q} \tag{3}$$
$$[@\mathbf{x} \cdot \mathbf{s}]\mathbf{q} :\Rightarrow \forall \mathbf{x} \cdot [\mathbf{s}]\mathbf{q} \qquad \text{where } \mathbf{x} \setminus \mathbf{q} \tag{4}$$
$$[\mathbf{s}; \mathbf{t}]\mathbf{q} :\Rightarrow [\mathbf{s}][\mathbf{t}]\mathbf{q} \tag{5}$$
$$[\mathsf{skip}]\mathbf{q} :\Rightarrow \mathbf{q} \tag{6}$$

2.1 Typechecking

Well-formed formulae of set notation in B are restricted by typechecking as well as the syntax. The aim of types is to rule out all forms of paradox in the formal system by restricting which formulae are well-formed.

Typechecking a predicate **p** containing no free variables consists of applying Algorithm 1 to the type sequent $\mathsf{given}(\mathsf{BIG}) \vdash \mathsf{check}(\mathbf{p})$. Typechecking an expression **e** can be done by typechecking the predicate $\mathbf{e} = \mathbf{e}$. If there are free variables, then appropriate type assumptions (either declaring them given sets with given or giving them type with the $\in$ operator) have to be prepended to the type sequent before typechecking.

The metavariable **i** denotes an identifier, the metavariable **E** denotes a comma-separated list of type assumptions, and the metavariables **T** and **U** denote types.

The typechecking algorithm given by Abrial [1] can be explicated as follows:

Algorithm 1 (Typechecking B according to Abrial). In this algorithm and in the names of the typechecking rules, the apostrophe is a decoration of numbers, and so i' denotes the number i decorated with an apostrophe. The algorithm uses standard unification (c.f. [5]) as tailored for the type language.

Precondition: Input is a type sequent.

1. Eliminate all derived constructs from the input.
2. Let $i \leftarrow 1$.
3. Unify the input with the consequent of rule T i, and assign the unifier to σ. If unsuccessful, go to step 7.
4. For each antecedent of rule T i, instantiate its variables using σ.
5. Verify each non-type-sequent antecedent. (This may require further unification; if so, instantiate the unified variables in all antecedents.) If at least one of them fail, go to step 7.
6. Apply this algorithm recursively to each type sequent antecedent. If all succeed, halt with success.
7. If there exists a rule $T\ i'$, let $i \leftarrow i'$ and go to step 3.
8. If $i = j'$ for some j, let $i \leftarrow j$ and repeat this step.
9. If there is no number n strictly greater than i for which there exists a rule $T\ n$, halt with failure.

[2] The metalevel notation $\mathbf{x} \setminus \mathbf{p}$ is to be pronounced "**x** does not occur free in **p**".

10. Let $i \leftarrow j$, where j is the least number strictly greater than i for which there exists a rule $T\ j$, and go to step 3.

Postcondition: Output is an indication of success or failure.

Note that the algorithm is essentially a generic Prolog-like inference engine.

The typing rules for Algorithm 1 are given in Figs. 2 and 3. Composite predicates are decomposed using rules T 1–T 3. Quantification is removed by enlarging the environment in rules T 4–T 6. Rules T 7 and T 8 transform the typechecking of primitive predicates ("∈" and "=") to appropriate type equivalence conjectures. Rules T 9–T 18 and their primed variants decompose the expressions involved in a type equivalence conjecture (T 16 and T 16' enlarge the environment in the process). Rules T 19–T 20 decompose power set and cartesian product expressions. Rule T 21 acts as the base case.

$$\text{T 1} \qquad \frac{\mathbf{E} \vdash \mathsf{check}(\mathbf{p}) \quad \mathbf{E} \vdash \mathsf{check}(\mathbf{q})}{\mathbf{E} \vdash \mathsf{check}(\mathbf{p} \wedge \mathbf{q})}$$

$$\text{T 2} \qquad \frac{\mathbf{E} \vdash \mathsf{check}(\mathbf{p}) \quad \mathbf{E} \vdash \mathsf{check}(\mathbf{q})}{\mathbf{E} \vdash \mathsf{check}(\mathbf{p} \Rightarrow \mathbf{q})}$$

$$\text{T 3} \qquad \frac{\mathbf{E} \vdash \mathsf{check}(\mathbf{p})}{\mathbf{E} \vdash \mathsf{check}(\neg \mathbf{p})}$$

$$\text{T 4} \qquad \frac{\mathbf{i} \setminus \mathbf{s} \quad \mathbf{i} \setminus \mathbf{q} \text{ for each } \mathbf{q} \text{ in } \mathbf{E} \quad \mathbf{E}, \mathbf{i} \in \mathbf{s} \vdash \mathsf{check}(\mathbf{p})}{\mathbf{E} \vdash \mathsf{check}(\forall \mathbf{i} \cdot (\mathbf{i} \in \mathbf{s} \Rightarrow \mathbf{p}))}$$

$$\text{T 5} \qquad \frac{\mathbf{E} \vdash \mathsf{check}(\forall \mathbf{x} \cdot (\mathbf{x} \in \mathbf{s} \Rightarrow \forall \mathbf{y} \cdot (\mathbf{y} \in \mathbf{t} \Rightarrow \mathbf{p})))}{\mathbf{E} \vdash \mathsf{check}(\forall (\mathbf{x}, \mathbf{y}) \cdot (\mathbf{x}, \mathbf{y} \in \mathbf{s} \times \mathbf{t} \Rightarrow \mathbf{p}))}$$

$$\text{T 6} \qquad \frac{\mathbf{E} \vdash \mathsf{check}(\forall \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q} \wedge \mathbf{r}))}{\mathbf{E} \vdash \mathsf{check}(\forall \mathbf{x} \cdot (\mathbf{p} \wedge \mathbf{q} \Rightarrow \mathbf{r}))}$$

$$\text{T 7} \qquad \frac{\mathbf{E} \vdash \mathsf{type}(\mathbf{e}) \equiv \mathsf{type}(\mathbf{f})}{\mathbf{E} \vdash \mathsf{check}(\mathbf{e} = \mathbf{f})}$$

$$\text{T 8} \qquad \frac{\mathbf{E} \vdash \mathsf{type}(\mathbf{e}) \equiv \mathsf{super}(\mathbf{f})}{\mathbf{E} \vdash \mathsf{check}(\mathbf{e} \in \mathbf{f})}$$

Fig. 2. First eight typechecking rules for Algorithm 1.

3 The Typechecker Is Broken

Abrial defines in B-Book [1] a type checking method which, taken at face value, would report type error on many examples and even in some normative parts of the definition. The method was reproduced earlier in this paper.

For example, the *BASIC_CONSTANTS* machine specified on page 281 of the B-Book [1] fails to typecheck. A part of its typechecking process leads to the

T 9 $$\frac{\mathbf{i} \in \mathbf{s} \text{ occurs in } \mathbf{E} \quad \mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{type}(\mathbf{i}) \equiv \mathbf{U}}$$

T 10 $$\frac{\mathbf{E} \vdash \mathsf{type}(\mathbf{e}) \times \mathsf{type}(\mathbf{f}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{type}(\mathbf{e}, \mathbf{f}) \equiv \mathbf{U}}$$

T 11 $$\frac{\mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{type}(\mathsf{choice}(\mathbf{s})) \equiv \mathbf{U}}$$

T 12 $$\frac{\mathbf{E} \vdash \mathbb{P}(\mathsf{super}(\mathbf{s})) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{type}(\mathbf{s}) \equiv \mathbf{U}}$$

T 13 $$\frac{\mathbf{i} \in \mathbf{s} \text{ occurs in } \mathbf{E} \quad \mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbb{P}(\mathbf{U})}{\mathbf{E} \vdash \mathsf{super}(\mathbf{i}) \equiv \mathbf{U}}$$

T 14 $$\frac{\mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \times \mathsf{super}(\mathbf{t}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{super}(\mathbf{s} \times \mathbf{t}) \equiv \mathbf{U}}$$

T 15 $$\frac{\mathbf{E} \vdash \mathbb{P}(\mathsf{super}(\mathbf{s})) \equiv U}{\mathbf{E} \vdash \mathsf{super}(\mathbb{P}(\mathbf{s})) \equiv \mathbf{U}}$$

T 16 $$\frac{\mathbf{E} \vdash \mathsf{check}(\forall \mathbf{x} \cdot (\mathbf{x} \in \mathbf{s} \Rightarrow \mathbf{p}) \quad \mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{super}(\{\, \mathbf{x} \mid \mathbf{x} \in \mathbf{s} \wedge \mathbf{p} \,\}) \equiv \mathbf{U}}$$

T 17 $$\frac{\mathsf{given}\mathbf{i} \text{ occurs in } \mathbf{E} \quad \mathbf{E} \vdash \mathbf{i} \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{super}(\mathbf{i}) \equiv \mathbf{U}}$$

T 18 $$\frac{\mathbf{E} \vdash \mathsf{super}(\mathbf{s}) \equiv \mathbb{P}(\mathbf{U})}{\mathbf{E} \vdash \mathsf{super}(\mathsf{choice}(\mathbf{s})) \equiv \mathbf{U}}$$

T 19 $$\frac{\mathbf{T} \equiv \mathbf{U}}{\mathbf{E} \vdash \mathbb{P}(\mathbf{T}) \equiv \mathbb{P}(\mathbf{U})}$$

T 20 $$\frac{\mathbf{E} \vdash \mathbf{T} \equiv \mathbf{T}' \quad \mathbf{E} \vdash \mathbf{T}' \equiv \mathbf{U}'}{\mathbf{E} \vdash \mathbf{T} \times \mathbf{U} \equiv \mathbf{T}' \times U'}$$

T 21 $$\frac{\mathsf{given}(\mathbf{i}) \text{ occurs in } \mathbf{E}}{\mathbf{E} \vdash \mathbf{i} \equiv \mathbf{i}}$$

Rules $9'$–$18'$ can be obtained from the corresponding rules 9–18 by applying to their consequents the rewrite rule $\mathbf{E} \vdash \mathbf{T} \equiv \mathbf{U} :\Rightarrow \mathbf{E} \vdash \mathbf{U} \equiv \mathbf{T}$.

Fig. 3. Rest of the typechecking rules for Algorithm 1.

following conjectural type sequent:[3]

$$\mathsf{given}(\mathbf{Z}) \vdash \mathsf{check}(\forall minint, maxint \cdot (minint \in \mathbf{Z} \wedge maxint \in \mathbf{Z} \Rightarrow \mathbf{Z} = \mathbf{Z})) \quad (7)$$

Feeding this to Algorithm 1 results in a failure, as the following trace shows:

$$\begin{array}{ll} \mathsf{given}(\mathbf{Z}) \vdash \mathsf{check}(\forall minint, maxint \cdot (minint \in \mathbf{Z} \wedge maxint \in \mathbf{Z} \Rightarrow \mathbf{Z} = \mathbf{Z})) & \mathrm{T\ 6} \\ \mathsf{given}(\mathbf{Z}) \vdash \mathsf{check}(\forall minint, maxint \cdot (minint \in \mathbf{Z} \Rightarrow maxint \in \mathbf{Z} \wedge \mathbf{Z} = \mathbf{Z})) & \mathrm{fail} \end{array}$$

There is no typing rule in Figs. 2 and 3 whose consequent unifies with the last sequent of the trace. The only one that comes close is T 5, but $minint \in \mathbf{Z}$ fails to unify with $(\mathbf{x}, \mathbf{y}) \in \mathbf{s} \times \mathbf{t}$.

Similar typing failures apply to most of the informative example machines in the B-Book, and many published uses of the B method (such as [6]). Clearly this failure is unintended, and the existing tools do not implement Abrial's method to the letter.

4 Fixing the Typechecker

The heart of the problem indicated in the previous section is that Abrial's typechecker requires that whenever a nonatomic variable is defined, e.g. as a quantification variable, it must be given a type as a whole instead of giving types to each constituent atomic variable (i.e. identifier) separately. Thus, for example, instead of writing $\exists x, y \cdot (x \in \mathbf{Z} \wedge y \in \mathbf{Z} \Rightarrow x < y)$, one must write $\exists x, y \cdot ((x, y) \in \mathbf{Z} \times \mathbf{Z} \Rightarrow x < y)$. The same problem exists with set comprehensions (such as $\{\, x, y \mid x \in \mathbf{Z} \wedge y \in \mathbf{Z} \wedge x = y \,\}$), since their typechecking defers, due to rules T 16 and T 16', to typechecking a universal quantification predicate.

We call quantification predicates and comprehension expressions of the forms

$$\forall \mathbf{x} \cdot (\mathbf{x} \in \mathbf{e} \Rightarrow \mathbf{p}) \quad (8)$$

$$\exists \mathbf{x} \cdot (\mathbf{x} \in \mathbf{e} \wedge \mathbf{p}) \quad (9)$$

$$\{\, \mathbf{x} \mid \mathbf{x} \in \mathbf{e} \wedge \mathbf{p} \,\} \quad (10)$$

regular. We call other syntactically well-formed quantification predicates and comprehension expressions *irregular*. With this terminology, the task becomes enlarging the type system to allow for certain irregular formulae.

The idea of the proposed fix is to go and find candidate types for the constituent atomic variables of the quantification variable and to build a candidate type from them for the whole bound variable. This new type can then be added to the front of the quantified predicate which allows Abrial's original typechecker

[3] We elect to omit the lengthy and complicated exposition on typechecking AMN, so we also omit the details on how this sequent is arrived at. Inessential complexity has been removed from the sequent. Note that the predicate $minint \in \mathbf{Z} \wedge maxint \in \mathbf{Z}$ is an unmodified part of the machine's properties section, as given in the B-Book. The last part, $\mathbf{Z} = \mathbf{Z}$, stands as a tautology for sections omitted from the machine.

to verify the new quantification predicate's typing. The same procedure works essentially unmodified for set comprehensions.

Additional typing rules for fixing the problem are given below.

T 5.5 $$\frac{\mathbf{p} \text{ is not of the form } \mathbf{x} \in \mathbf{f} \quad \vdash \mathsf{typeQ}(\mathbf{x}, \mathbf{p}, \mathbf{e}) \quad \mathbf{E} \vdash \mathsf{check}(\forall \mathbf{x} \cdot (\mathbf{x} \in \mathbf{e} \Rightarrow (\mathbf{p} \Rightarrow \mathbf{q})))}{\mathbf{E} \vdash \mathsf{check}(\forall \mathbf{x} \cdot (\mathbf{p} \Rightarrow \mathbf{q}))}$$

T 16 $$\frac{\vdash \mathsf{typeQ}(\mathbf{x}, \mathbf{p}, \mathbf{e}) \quad \mathbf{E} \vdash \mathsf{check}(\forall \mathbf{x} \cdot (\mathbf{x} \in \mathbf{e} \Rightarrow \mathbf{p})) \quad \mathbf{E} \vdash \mathsf{super}(\mathbf{e}) \equiv \mathbf{U}}{\mathbf{E} \vdash \mathsf{super}(\{\, \mathbf{x} \mid \mathbf{p} \,\}) \equiv \mathbf{U}}$$

T 40 $$\frac{\vdash \mathsf{typeQ}(\mathbf{i}, \mathbf{p}, \mathbf{e})}{\vdash \mathsf{typeQ}(\mathbf{i}, \mathbf{p} \wedge \mathbf{q}, \mathbf{e})}$$

T 41 $$\frac{\vdash \mathsf{typeQ}(\mathbf{i}, \mathbf{q}, \mathbf{e})}{\vdash \mathsf{typeQ}(\mathbf{i}, \mathbf{p} \wedge \mathbf{q}, \mathbf{e})}$$

T 42 $$\frac{\vdash \mathsf{typeQ}(\mathbf{i}, \mathbf{p}, \mathbf{e})}{\vdash \mathsf{typeQ}(\mathbf{i}, \mathbf{p} \Rightarrow \mathbf{q}, \mathbf{e})}$$

T 43 $$\frac{}{\vdash \mathsf{typeQ}(\mathbf{i}, \mathbf{i} \in \mathbf{e}, \mathbf{e})}$$

T 44 $$\frac{\vdash \mathsf{typeQ}(\mathbf{x}, \mathbf{p}, \mathbf{e}) \quad \vdash \mathsf{typeQ}(\mathbf{y}, \mathbf{p}, \mathbf{f})}{\vdash \mathsf{typeQ}((\mathbf{x}, \mathbf{y}), \mathbf{p}, \mathbf{e} \times \mathbf{f})}$$

A new inference rule, T 5.5, is inserted to take care of an irregular quantification. Abrial's rules T 16 and T 16' are replaced with new rules that allow irregular set comprehensions. These new rules reference a new type predicate, typeQ, whose behaviour is defined in five new rules, T 40–T 44. The choice of numbering for these rules is arbitrary — 40 was chosen mainly so that this block of rules does not interfere with numbering other possible extensions of the type system — since their consequents do not overlap with the consequents of other rules, with a single exception. Rules T 40 and T 41 have the same consequent, and the intent is that angelic nondeterminism is used to choose between them. Algorithm 1 tries them in numeric order and backtracks if the first one fails.

We will call Algorithm 1 together with the new set of typing rules *extended typechecking* or *the extended typechecker.*

5 Analyzing the extended typechecker

The extended typechecker terminates for all input if Abrial's original does. The only likely candidate for causing nontermination is T 5.5, but it specifically forbids its use on a predicate that it itself generates.

It is fairly easy to see, due to the backtracking nature of Algorithm 1, that the new rules do not cause typechecking to reject any formulae Abrial's original system didn't reject.

Finally, we will show that the extended typechecker does not extend the expressive power of the language, thus ensuring that if the original system is consistent, the new system is consistent as well. To this end, we prove that every formula allowed by extended typechecking is logically equivalent to a formula allowed by Abrial's original typechecker:

Theorem 1. *Let* $\mathbf{e}_1, \ldots, \mathbf{e}_m$ *be expressions and let* $\mathbf{g}_1, \ldots, \mathbf{g}_n$ *and* $\mathbf{i}_1, \ldots \mathbf{i}_m$ *be identifiers. Let* $\mathbf{e}'_1, \ldots, \mathbf{e}'_m$ *be expressions such that* $\mathbf{e}_i = \mathbf{e}'_i$ *is a theorem for each* i*. Further, let* $\mathbf{e}$ *be an expression and let* $\mathbf{p}$ *be a predicate. Finally, let* $\mathbf{E}$ *be*

$$\mathsf{given}(\mathbf{g}_1), \ldots, \mathsf{given}(\mathbf{g}_n), \mathbf{i}_1 \in \mathbf{e}_1, \ldots, \mathbf{i}_m \in \mathbf{e}_m, \tag{11}$$

and let $\mathbf{E}'$ *be*

$$\mathsf{given}(\mathbf{g}_1), \ldots, \mathsf{given}(\mathbf{g}_n), \mathbf{i}_1 \in \mathbf{e}'_1, \ldots, \mathbf{i}_m \in \mathbf{e}'_m. \tag{12}$$

Then the following hold:

1. *If* $\mathbf{E} \vdash \mathsf{check}(\mathbf{p})$ *passes extended checking, then there exist expressions* $\mathbf{e}'_i$ *as specified above and a predicate* $\mathbf{p}'$ *such that* $\mathbf{E}' \vdash \mathsf{check}(\mathbf{p}')$ *passes Abrial's original typechecking and*

$$\forall(\mathbf{i}_1, \ldots \mathbf{i}_m) \cdot ((\mathbf{i}_1, \ldots \mathbf{i}_m) \in (\mathbf{e}'_1 \times \ldots \times \mathbf{e}'_m) \Rightarrow (\mathbf{p} \Leftrightarrow \mathbf{p}')) \tag{13}$$

   is a theorem.
2. *If* $\mathbf{E} \vdash \mathbf{e} \equiv \mathbf{e}$ *passes extended checking, then there exist expressions* $\mathbf{e}'_i$ *as specified above and an expression* $\mathbf{e}'$ *such that* $\mathbf{E}' \vdash \mathbf{e}' \equiv \mathbf{e}'$ *passes Abrial's original typechecking and*

$$\forall(\mathbf{i}_1, \ldots \mathbf{i}_m) \cdot ((\mathbf{i}_1, \ldots \mathbf{i}_m) \in (\mathbf{e}'_1 \times \ldots \times \mathbf{e}'_m) \Rightarrow (\mathbf{e} = \mathbf{e}')) \tag{14}$$

   is a theorem.

Proof: We will assume that all derived constructs have been eliminated from all formulae that we consider. Proof of this theorem proceeds by structural induction:

1. The cases where $\mathbf{p}$ is a conjunction, implication, negation, equality or set membership, and the cases where $\mathbf{e}$ is a variable, a pair, choice, Cartesian product, or BIG, are straightforward. We omit their proof.
2. Let $\mathbf{p}$ have the form $\forall \mathbf{i} \cdot (\mathbf{i} \in \mathbf{e} \Rightarrow \mathbf{q})$, where $\mathbf{i}$ is an identifier and $\mathbf{q}$ is a predicate. Now, due to T 5 and to the assumption that $\mathbf{p}$ passes extended typechecking, $\mathbf{E}, \mathbf{i} \in \mathbf{e} \vdash \mathsf{check}(\mathbf{q})$ passes extended typechecking. Thus, by induction, there exist $\mathbf{e}'_i$ and $\mathbf{e}'$ as given above and a predicate $\mathbf{q}'$ for which $\mathbf{E}', \mathbf{i} \in \mathbf{e}' \vdash \mathsf{check}(\mathbf{q}')$ passes Abrial's original typechecking and for which (13), after appropriate substitutions, holds. Now, $\forall \mathbf{i} \cdot (\mathbf{i} \in \mathbf{e}' \Rightarrow \mathbf{q}')$ will clearly qualify for $\mathbf{p}'$.
3. The case where $\mathbf{p}$ has the form $\forall \mathbf{x}, \mathbf{y} \cdot ((\mathbf{x}, \mathbf{y}) \in (\mathbf{e} \times \mathbf{f}) \Rightarrow \mathbf{q})$, follows similarly and its proof is omitted.
4. Let now $\mathbf{p}$ have the form $\forall \mathbf{x} \cdot (\mathbf{q} \Rightarrow \mathbf{r})$, where $\mathbf{x}$ is a variable, $\mathbf{q}$ is a predicate not of the form $\mathbf{x} \in \mathbf{f}$ for some expression $\mathbf{f}$, and $\mathbf{r}$ is a predicate. Now, due to T 5.5 and to the assumption that $\mathbf{p}$ passes extended typechecking,

$\mathbf{E} \vdash \mathsf{check}(\forall \mathbf{x} \cdot (\mathbf{x} \in \mathbf{e} \Rightarrow (\mathbf{q} \Rightarrow \mathbf{r})))$ passes extended typechecking, if $\mathbf{e}$ is an expression such that $\vdash \mathsf{typeQ}(\mathbf{x}, \mathbf{p}, \mathbf{e})$ passes extended typechecking. By induction, then, there exist expressions $\mathbf{e}'_i$ as given above and a predicate $\mathbf{p}'$ such that $\mathbf{E}' \vdash \mathsf{check}(\mathbf{p}')$ passes Abrial's original typechecking and for which (13), after appropriate substitutions, holds. Note that typeQ essentially keeps only one set membership constraint for each identifier, dropping any others off, and thus the predicate $\mathbf{x} \in \mathbf{e}$ will imply $\mathbf{q}$. Now $\mathbf{p}'$ qualifies as the $\mathbf{p}'$ of the theorem statement, if

$$\forall \mathbf{x} \cdot (\mathbf{q} \Rightarrow \mathbf{r}) \Leftrightarrow \forall \mathbf{x} \cdot (\mathbf{x} \in \mathbf{e} \Rightarrow (\mathbf{q} \Rightarrow \mathbf{r})) \tag{15}$$

is a theorem, and it is, since the following are equivalent:

$$\forall \mathbf{x} \cdot (\mathbf{q} \Rightarrow \mathbf{r}) \Leftrightarrow \forall \mathbf{x} \cdot (\mathbf{x} \in \mathbf{e} \Rightarrow (\mathbf{q} \Rightarrow \mathbf{r})) \tag{16}$$

$$\forall \mathbf{x} \cdot (\mathbf{q} \Rightarrow \mathbf{r} \Leftrightarrow \mathbf{x} \in \mathbf{e} \Rightarrow (\mathbf{q} \Rightarrow \mathbf{r})) \tag{17}$$

$$\forall \mathbf{x} \cdot (\mathbf{q} \Rightarrow \mathbf{r} \Leftrightarrow \mathbf{q} \Rightarrow \mathbf{r}) \tag{18}$$

and the last of them is obviously a theorem.

5. Let $\mathbf{e}$ have the form $\{\, \mathbf{x} \mid \mathbf{q} \,\}$, where $\mathbf{x}$ is a variable and $\mathbf{q}$ is a predicate. Now, the typechecking proposition follows in the same manner as in the previous case.
   To prove the other part of the proposition, we choose for $\mathbf{e}'$ the expression $\{\, \mathbf{x} \mid \mathbf{x} \in \mathbf{f} \wedge \mathbf{q} \,\}$, where $\mathbf{f}$ is an expression such that $\vdash \mathsf{typeQ}(\mathbf{x}, \mathbf{p}, \mathbf{f})$ passes extended typechecking. Now, the equality

$$\{\, \mathbf{x} \mid \mathbf{q} \,\} = \{\, \mathbf{x} \mid \mathbf{x} \in \mathbf{f} \wedge \mathbf{q} \,\} \tag{19}$$

   is, by comprehension, equivalent to

$$\forall \mathbf{x} \cdot (\mathbf{q} \Leftrightarrow \mathbf{x} \in \mathbf{f} \wedge \mathbf{q}) \tag{20}$$

   which can be seen to be a theorem in the manner outlined above. □

6 Conclusion

We pointed out a problem in the typechecking system of the B method, namely that, although the B literature is full of formal text where variables declared together are given types in separate membership predicates, joined by conjunctions, the B typechecking system rejects all such input. We augmented the system by additional typing rules and thus allowed the system to accept such input where it is semantically reasonable.

Acknowledgements

The author would like to thank professor Tommi Kärkkäinen for his advice and comments on drafts of this paper and Jonne Itkonen for his help in preparing the final version.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Kaijanaho, A.J.: The formal method known as B and a sketch for its implementation. Master's thesis, University of Jyväskylä, Department of Mathematical Information Technology (2002)
3. Dunne, S.: A theory of generalised substitutions. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: ZB2002: Formal Specification and Development in Z and B, Second International Conference of B and Z users, Grenoble, France. Number 2272 in Lecture Notes in Computer Science, Berlin, Springer (2002)
4. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs (1976)
5. Baader, F., Snyder, W.: Unification theory. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier (North-Holland), Amsterdam (2001)
6. Treharne, H., Schneider, S.: How to drive a B machine. In Bowen, J.P., Dunne, S., Galloway, A., King, S., eds.: ZB 2000: Formal Specification and Development in Z and B, First International Conference of B and Z Users, York, UK. Number 1878 in Lecture Notes in Computer Science, Berlin, Springer (2000)