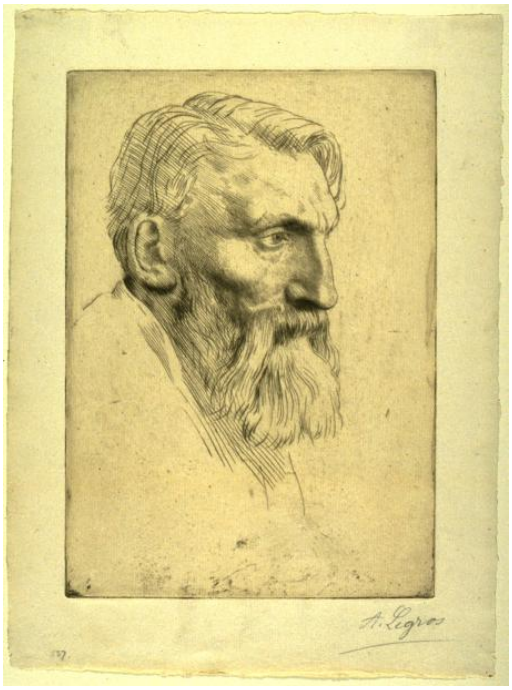
Project IST-511599

RODIN

"Rigorous Open Development Environment for Complex Systems"



RODIN Deliverable 3.2

# Event-B Language

*C. Métayer (ClearSy) J.-R. Abrial, L. Voisin (ETH Zürich)*

Public Document

31st May 2005

`http://rodin.cs.ncl.ac.uk`

# (I) Event-B: Informal Presentation

J.-R. Abrial

April 2005

Version 1.1

# (I) Event-B: Informal Presentation

## 1 Introduction

It is our belief that the people in charge of the development of large and complex computer systems should adopt a point of view shared by all mature engineering disciplines, namely that of using an artifact to reason about their future system during its construction. In these disciplines, people use *blue-prints* in the wider sense of the term, which allows them to reason formally during the very construction process.

Most of the time, in our discipline, we do not use such artifacts. This results in a very heavy testing phase on the final product, which is well known to happen quite often too late. The blue-print drawing of our discipline consists of *building models* of our future systems. But in no way is the model of a program the program itself for the simple reason that the model, like the blue-print, must not be executable: you cannot drive the blue-print of a car. But the model of a program and more generally of a complex computer system, although not executable, allows you to clearly identify the properties of the future system and to prove that they will be present in it.

Building models of large systems however is not an easy task. First of all because we lack experience in this activity. Such a discipline does not exist in Academia, where quite often model building is confused with using a very high level programming language where execution is thus still present. Moreover, reasoning means ensuring that the properties which define the future system can be *proved* to be consistent. As a matter of fact, doing a proof on a model replaces an impossible execution. But again, the mastering of formal proving techniques has not yet entered the standard curriculum of our discipline. As a consequence, people are quite reluctant to adopt such an approach, simply because they do not know how to do it.

We present here the way this approach can be encoded within a formal language, whose semantics is simply given by formalizing what can be proved from a corresponding formal text.

In this introduction, we state in very general terms the problems we want to address. We are essentially concerned with the development of complex systems (section 1.1), which behave in a discrete fashion (section 1.2). We are interested in exhibiting some form of reasoning which definitely departs from that implied by testing in the broad sense of the term (section 1.3). We shall then give (section 2) an informal view of the notion of discrete model which is called here *Event-B.*

This document is the first one of five companion documents describing different facets of Event-B. The other documents are the following:

- (II) Event-B: Structure and Laws. This document describes the way Event-B models can be defined and also the rules that must be followed in order to guarantee that an Event-B model is coherent.

- (III) Event-B: Mathematical Model. This document gives a mathematical model of Event-B thus justifying the laws that were given in the previous document. The reading of this document can be skipped without losing the full understanding of the others.

- (IV) Event-B: Examples. This document provides a number of simple examples in order to illustrate the material presented in the previous documents.

- (V) Event-B: Mathematical Language. This document presents the mathematical language we are going to use: First Order Predicate Calculus extended with Set Theory.

- (VI) Event-B: Syntax of Mathematical Language. This document covers in detail the syntactic structure of the Mathematical Language presented in the previous document. It also shows how the type checking is done. Finally, it shows what is to be proved in order to ensure the well-definedness of a mathematical text.

### 1.1 Complex Systems

Here are the kinds of questions we might ask to begin with. What is common to, say, an electronic circuit, a file transfer protocol, an airline seat booking system, a sorting program, a PC operating system, a network routing program, a nuclear plant control system, a SmartCard electronic purse, a launch vehicle flight controller, etc.? Does there exist any kind of unified approach to in depth study and formal proof of the requirements, the specification, the design and the implementation of *systems* that are so different in size and purpose?

We shall only give for the moment a very general answer. Almost all such systems are *complex* in that they are made of many parts interacting with a highly evolving and sometimes hostile environment. They also quite often involve several concurrent executing agents. They require a high degree of correctness. Finally, most of them are the result of a construction process which is spread over several years and which requires a large and talented team of engineers and technicians.

### 1.2 Discrete Systems

Although their behavior is certainly ultimately continuous, the systems which were listed in the previous section are most of the time operating in a *discrete fashion*. This means that their behavior can be faithfully *abstracted* by a succession of steady states intermixed with jumps which make their state suddenly change to others. Of course, the number of such possible changes is enormous, and they are occurring in a concurrent fashion at an unthinkable frequency. But this number and this high frequency do not change the very nature of the problem: such systems are intrinsically discrete. They fall under the generic name of *transition systems*. Having said this does not make us moving very much towards a methodology, but it gives us at least *a common point of departure*.

Some of the examples envisaged above are pure programs. In other words, their transitions are essentially concentrated in *one medium* only. The electronic circuit and the sorting program clearly fall into this category. Most of the other examples however are far more complex than just pure programs because they involve many different executing agents and also a heavy interaction with their environment. This means that the transitions are executed by different kinds of entities acting concurrently. But, again, this does not change the very discrete nature of the problem, it only complicates matters.

### 1.3 Test Reasoning versus Model (Blue Print) Reasoning

A very important activity, at least in terms of time and money, concerned with the construction of such discrete systems is certainly that consisting of verifying that their final implementations are operating in a, so called, *correct* fashion. Most of the time nowadays, this activity is realized during a very heavy testing phase, which we shall call a “laboratory execution”.

The validation of a discrete system by means of such “laboratory executions” is certainly far more complicated to realize, if not impossible in practice, in the multiple medium case than in the single medium one. And we already know however that program testing, used as a validation process in almost all programming projects, is by far an incomplete process. Not so much, in fact, because of the impossibility to achieve a total cover of all executing cases. The incompleteness is rather, for us, the consequence of the frequent *lack of oracles* which would give, *beforehand* and independently of the tested objects, the expected results of a future testing session.

It is nevertheless the case that today the basic ingredients for complex system construction still are a very small design team of smart people, managing an army of implementers, eventually concluding the construction process with a long and heavy testing phase. And it is a well known fact that the testing cost is at least twice that of the pure development effort. Is this a reasonable attitude nowadays? Our opinion is that a technology using such an approach is still in its infancy. This was the case at the beginning of last century for some technologies, which have now reached a more mature status (e.g. avionics).

The technology we consider in this short presentation is that concerned with the construction of *complex discrete systems*. As long as the main validation method used is that of testing, we consider that this technology will remain in an underdeveloped state. Testing does not involve any kind of sophisticated reasoning. It rather consists of *always postponing any serious thinking* during the specification and design phase. The construction of the system will always be re-adapted and re-shaped according to the testing results (trial and error). But, as one knows, it is quite often too late.

In conclusion, testing always gives a shortsighted operational view over the system in construction: that of execution. In other technologies, say again avionics, it is certainly the case that people eventually do test what they are constructing, but the testing is just the *routine confirmation* of a sophisticated design process rather than a fundamental phase in it. As a matter of fact, most of the reasoning is done *before* the very construction of the final object. It is performed on various "blue prints", in the broad sense of the term, by applying on them some well defined practical theories.

The purpose of this study is to incorporate such a "blue print" approach in the design of complex discrete systems. It also aims at presenting a theory able to facilitate the elaboration of some *proved reasoning* on such blue prints. Such reasoning will thus take place far before the final construction. In the present context, the "blue prints" are called *discrete models*. We shall now give a brief informal overview of the notion of discrete model.

## 2 Informal Overview of Discrete Models

In this section, we give an informal description of discrete models. A discrete model is made of a state and a number of transitions (section 2.1). For the sake of understanding, we then give an operational interpretation of discrete models (section 2.2). We then present the kind of formal reasoning we want to express (section 2.3). Finally we briefly address the problem of mastering the complexity of models (section 2.4) by means of three concepts: refinement (section 2.5), decomposition (section 2.6) and generic development (section 2.7),

### 2.1 State and Transitions

Roughly speaking, a discrete model is made of a *state* represented by some significant constants and variables at a certain level of abstraction with regards to the real system under study. Such variables are very much the same as those used in applied sciences (physics, biology, operational research) for studying natural systems. In such sciences, people also build models. It helps them to infer some laws on the reality by means of some reasoning, which they undertake on these models.

Besides the state, the model also contains a number of *transitions* that can occur under certain circumstances. Such transitions are called here "events". Each event is first made of a *guard*, which is a predicate built on the state constants and variables. It represents the *necessary* conditions for the event to occur. Each event is also made of an *action*, which describes the way certain state variables are modified as a consequence of the event occurrence.

### 2.2 Operational Interpretation

As can be seen, a discrete dynamical model thus indeed constitutes a kind of state transition machine. We can give such a machine an extremely simple *operational interpretation*. Notice that such an interpretation should not be considered as providing any operational semantics to our models (this will be given later by means of a proof system), it is just given here to support their *informal understanding*.

First of all, the execution of an event, which describes a certain observable transition of the state variables, is considered to take *no time*. As an immediate consequence, no two events can occur simultaneously. The execution is then the following:

– When no event guard is true, then the model execution stops: *it is said to have deadlocked*.

– When some event guards are true, then one of the corresponding events necessarily occurs and the state is modified accordingly, finally the guards are checked again, and so on.

This behavior clearly shows some possible non-determinism (called external non-determinism) as several guards might be true simultaneously. We make *no assumption* concerning the specific event which is indeed executed among those whose guards are true. When only one guard is true at a time, the model is said to be deterministic.

Note that the fact that a model eventually deadlocks is *not at all mandatory*. As a matter of fact, most of the systems we study never deadlock: they run for ever.

### 2.3 Formal Reasoning

The very elementary machine we have described in the previous section, although primitive, is nevertheless sufficiently elaborate to allow us to undertake some interesting formal reasoning. In the following we envisage two kinds of discrete model properties.

The first kind of properties that we want to prove about our models, and hence ultimately about our real systems, are, so called, *invariant properties*. An invariant is a condition on the state variables that must hold permanently. In order to achieve this, it is just required to *prove* that, under the invariant in question and under the guard of each event, the invariant still holds after being modified according to the transition associated with that event.

We might also consider more complicated forms of reasoning involving conditions which, in contrast with the invariants, do not hold permanently. The corresponding statements are called *modalities*. In our approach we only consider a very special form of modality called *reachability*. What we would like to prove is that an event whose guard is not necessarily true now will nevertheless certainly occur within a certain finite time.

### 2.4 Managing the Complexity of Closed Models

Note that the models we are going to construct will not just describe the control part of our intended system. It will also contain a certain representation of the environment within which the system we build is supposed to behave. In fact, we shall quite often essentially construct *closed models* able to exhibit the actions and reactions which take place between a certain environment and a corresponding, possibly distributed, controller, which we intend to construct.

In doing so, we shall be able to plunge the model of the controller within an abstraction of its environment, which is formalized as yet another model. The state of such a closed system thus contains physical variables, describing the environment state, as well as logical variables, describing the controller state. And, in the same way, the transitions will fall into two groups: those concerned by the environment and those concerned by the controller. We shall also have to put into the model the way these two entities communicate.

But as we mentioned earlier, the number of transitions in the real systems under study is certainly enormous. And, needless to say, the number of variables describing the state of such systems is also extremely large. How are we going to practically manage such a complexity? The answer to this question lies in three concepts: *refinement* (section 2.5), *decomposition* (section 2.6), and *generic instantiation* (section 2.7). It is important to notice here that these concepts are linked together. As a matter of fact, one refines a model to later decompose it, and, more importantly, one decomposes it to further more freely refine it. And finally, a generic model development can be later instantiated, thus saving the user of redoing almost similar proofs.

### 2.5 Refinement

Refinement allows us to build a model *gradually* by making it more and more precise, that is closer to the reality. In other words, we are not going to build a single model representing once and for all our reality in a flat manner: this is clearly impossible due to the size of the state and the number of its transitions. It would also make the resulting model very difficult to master, if not just to read. We are rather going to construct an ordered sequence of embedded models, where each of them is supposed to be a refinement of the one preceding it in that sequence. This means that a refined, more concrete, model will have more variables than its abstraction: such new variables are the consequence of a closer look at our system.

A useful analogy here is that of the scientist looking through a microscope. In doing so, the reality is the same, the microscope does not change it, *our look at it is only more accurate*: some previously invisible parts of the reality are now revealed by the microscope. An even more powerful microscope will reveal more parts, etc. A refined model is thus one which is spatially larger than its previous abstractions.

And correlatively to this *spatial extension*, there is a corresponding *temporal extension*: this is because the new variables are now able to be modified by some transitions, which could not have been present in the previous abstractions simply because the variables concerned did not exist in them. Practically this is realized by means of *new events* involving the new variables only. Such new events refine some implicit events doing nothing on the abstraction. Refinement will thus result in a discrete observation of our reality, which is now performed using a *finer time granularity*.

Refinement is also used in order to modify the state so that it can be implemented on a computer by means of some programming language. This second usage of refinement is called *data-refinement*. It is used as a second technique, once the model has been gradually constructed.

### 2.6 Decomposition

Refinement does not solve completely the mastering of the complexity. As a model is more and more refined, the number of its state variables and that of its transitions may augment in such a way that it becomes impossible to manage the whole. At this point, it is necessary to cut our single refined model into several almost independent pieces.

Decomposition is precisely the process by which a single model can be split into various component models in a systematic fashion. In doing so, we reduce the complexity of the whole by studying, and thus refining, each part independently of the others. The very definition of such a decomposition implies that independent refinements of the parts could always be put together again to form a single model that is guaranteed to be a refinement of the original one. This decomposition process can be further applied on the components, and so on. Note that the decomposed model could already exist and be developed, thus allowing to mix a top down approach with a bottom up one.

### 2.7 Generic Development

Any model development done by applying refinement and decomposition, is parameterized by some carrier sets and constants defined by means of a number of properties.

Such a generic model could then be instantiated within another development in the same way as a mathematical theory like, say, group theory, can be instantiated in a more specific mathematical theory. This can be done provided one has been able to prove that the axioms of the abstract theory are mere theorems in the second one.

The interest of this approach of generic instantiation is that it saves us redoing the proofs already done in the abstract development.

# (II) Event-B: Structure and Laws

J.-R. Abrial

April 2005

Version 2

# (II) Event-B: Structure and Laws

## 1 Introduction

This document contains a complete description of the structure of *Event-B*. It also contains the rules that must be proved in order to ensure that an Event-B development is correct. It is decomposed into four sections dealing with models and contexts (section 2), refinements (section 3), decomposition (section 4), and generic instantiation (section 5).

## 2 Models and Contexts

This section contains the description of an Event-B model and of the associated context. In section 2.1, the state and event structures of a model are described. Then in sections 2.2, 2.3, and 2.4 you will find a description of generalized substitutions defining the transition associated with an event. In section 2.5, the notion of context, allowing us to define the parametric structure of a model, is described. Finally, in section 2.6 you will find the consistency rules which must be proved for an event model to be correct.

### 2.1 State and Events.

A *formal discrete model* is made of four elements: (1) a name, (2) a list of distinct state variables, collectively denoted by $v$, (3) a list of named predicates, the invariants, collectively denoted by $I(v)$, and (4) a collection of transitions (here called events). This is illustrated in Fig. 1. The invariant $I(v)$ yields the laws that the state variables $v$ must always fulfil. These laws are formalized by means of predicates expressed within the language of First Order Predicate Calculus with Equality extended by Set Theory.

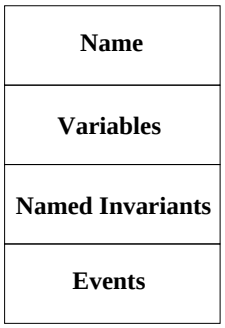



**Fig. 1.** A Model

An *event*, is made of three elements: (1) a name, (2) a list of named predicates, the guards, collectively denoted by $G(v)$, and (3) a generalized substitution denoted by $S(v)$. This is illustrated in Fig.

| **Name** |
| --- |
| **Named Guards** |
| **Generalized Substitution** |

**Fig. 2.** An Event

2. The guards $G(v)$ state the necessary conditions for the event to occur, and the generalized substitution $S(v)$ defines the state transition associated with the event. Among these events, a special one, called initialization, allows one to define an initial situation for a model: this event has no guard. For later convenience, an event E with guard $G(v)$ and generalized substitution $S(v)$ can be given the syntactic form shown in Fig. 3.

$$\mathsf{E} \;\widehat{=}\; \textbf{when}\;\; G(v)\;\; \textbf{then}\;\; S(v)\;\; \textbf{end}$$

**Fig. 3.** Syntactic form of an event

Events can be grouped to form, so-called, arrays of events. Besides the normal elements of an events (name, guards, and generalized substitution), an array of events has two more elements: (1) a list of local distinct indices, collectively denoted by $i$, and (2) a list of array conditions, collectively denoted by $C(i)$. This is illustrated in Fig. 4.

| **Name** |
| --- |
| **Indices** |
| **Array Conditions** |
| **Named Guards** |
| **Generalized Substitution** |

**Fig. 4.** An array of events

In Fig. 5, is shown the syntactic form of an array of events.

```
                array  i  where
                  C(i)
E   ≙         then
                  when  G(i, v)  then  S(i, v)  end
                end
```

**Fig. 5.** Syntactic form of an array of events

### 2.2 Generalized Substitutions.

We have three kinds of generalized substitutions for expressing the transition associated with an event: (1) the deterministic multiple substitution, (2) the empty substitution, and (3) the non-determinitistic multiple substitution. The shapes of these constructs are shown in Fig. 6

| **Kind** | **Generalized Substitution** |
|---|---|
| Deterministic | $x := E(v)$ |
| Empty | skip |
| Non-deterministic | **any** $t$ **where** $P(t, v)$ **then** $x := F(t, v)$ **end** |

**Fig. 6.** Kinds of Generalized Substitutions

In the deterministic and non-deterministic cases, $x$ denotes a list of variables of $v$ which are all distinct. In the deterministic case, $E(v)$ denotes a number of set-theoretic expressions corresponding to each of the variables in $x$. In the non-deterministic case, $t$ denotes a collection of distinct fresh variables which are local to the generalized substitution, $P(t, v)$ denotes a conjoined list of predicates, and $F(t, v)$ denotes a number of set-theoretic expressions corresponding to each of the variables in $x$. As can be seen, not all variables in $v$ are necessarily assigned in a substitution since $x$ does not necessarily cover $v$.

The variables that are placed on the left hand side of the assignment operator ":=" in a generalized substitution are called the *left variables* of that substitution. The ones occurring on the right hand side are called the *right variables*. Some state variables can be left and right variables at the same time in a given substitution. In the generalized substitution skip, the variables $v$ are all, by convention, left as well as right variables.

In the initialization event mentioned in section 2.1, all state variables are left variables only (thus the generalized substitution skip is not possible).

### 2.3 Generalized Substitutions Syntactic Facilities

In this section, we define a number of syntactic facilities allowing us to denote generalized substitutions in a way which is slightly different from the one used in the previous section. All such new forms however are defined in terms of the ones we introduced in the previous section. They are just proposed here as shorthands and can thus always be eliminated.

We first introduce two constructs by means of the operators $:|$ and $:\in$. They are shown in Fig. 7 under the form of two rewriting rules.

| New Construct | Rewritten |
|---|---|
| $x :\mid P(x_0, x, y)$ | **any** $z$ **where** $P(x, z, y)$ **then** $x := z$ **end** |

| New Construct | Rewritten |
|---|---|
| $x :\in S(v)$ | **any** $z$ **where** $z \in S(v)$ **then** $x := z$ **end** |

**Fig. 7.** Rewriting Rules for the $:|$ and $:\in$ operators

The first one is to be read "$x$ becomes such that the predicate $P(x_0, x, y)$ holds", where $x$ denotes some distinct variables of $v$, $y$ denotes those variables of $v$ that are distinct from $x$, and $x_0$ denotes the values of the variables $x$ before the substitution is applied. The second one is to be read "$x$ becomes a member of the set $S(v)$". In both rewritten generalized substitutions, $z$ denotes a number of distinct fresh variables. Each of them corresponds to the variables of $x$.

The next series of constructs allows one to define the different substitutions of an event in a separate manner: this is done by means of the parallel operator "$\|$". The definition of this operator takes the form of a number of rewriting rules corresponding to the combination of the two basic generalized substitutions presented in section 2.2 (skip is not concerned by these rewriting rules). These rewriting rules are shown in Fig. 8 and 9. In all cases, $x$ and $y$ denote distinct variables of $v$. In the last case of Fig. 9, the fresh variables in $t$ and $u$ must be distinct. Thanks to these rewriting rules, it is possible to completely eliminate the '$\|$' operator.

| Combination | Rewritten |
|---|---|
| $x := E(v) \quad \| \quad y := F(v)$ | $x, y := E(v), F(v)$ |
| **any** $t$ **where** $P(t, v)$ **then** $x := E(t, v)$ **end** $\quad \| \quad y := F(v)$ | **any** $t$ **where** $P(t, v)$ **then** $x, y := E(t, v), F(v)$ **end** |

**Fig. 8.** Rewriting Rules for the Parallel Operator (1)

| Combination | Rewritten |
|---|---|
| $x := E(v)$ $\parallel$ **any** $t$ **where** $P(t,v)$ **then** $y := F(t,v)$ **end** | **any** $t$ **where** $P(t,v)$ **then** $x, y := E(v), F(t,v)$ **end** |
| **any** $t$ **where** $P(t,v)$ **then** $x := E(t,v)$ **end** $\parallel$ **any** $u$ **where** $Q(u,v)$ **then** $y := F(u,v)$ **end** | **any** $t, u$ **where** $P(t,v) \wedge Q(u,v)$ **then** $x, y := E(t,v), F(u,v)$ **end** |

**Fig. 9.** Rewriting Rules for the Parallel Operator (2)

### 2.4 Before-After Predicates Associated with a Generalized Substitution.

The before-after predicate of a generalized substitution denotes the condition defining the *binary relation* associated with the corresponding transition. It is expressed in terms of the state variable values connected by this relation. By convention, the before values of the variables $v$ are also denoted by $v$ and the after values are denoted by $v'$. More generally, if $x$ denotes a number of state variables of the model, we collectively denote by $x$ and $x'$ their values before and after the transition. The before-after predicate is defined in Fig. 10 for the three basic generalized substitutions presented in section 2.2.

| Generalized Substitution | Before-after Predicate |
|---|---|
| $x := E(v)$ | $x' = E(v) \ \wedge \ y' = y$ |
| skip | $v' = v$ |
| **any** $t$ **where** $P(t,v)$ **then** $x := F(t,v)$ **end** | $\exists\, t \cdot (\, P(t,v) \ \wedge \ x' = F(t,v)\,) \ \wedge \ y' = y$ |

**Fig. 10.** Before-after Predicates

In the table of Fig. 10, the letter $y$ denotes the set of variables of $v$ which are distinct from those in $x$. As can be seen, such variables are not modified by the substitution, as shown by the equalities $y' = y$ in the before-after predicates. It is thus important to note that the before-after predicate of the substitution of an event is *not a universal property* of that substitution: it depends on the variables of the model where the event resides. The most obvious case is that of the empty substitution.

Note that since the generalized substitution of the initialization event has no right variables (see end of section 2.2), the before-after predicate of its generalized substitution is rather simply an after predicate.

### 2.5 Contexts

In the previous sections, we have considered that a discrete model was made of a number of variables, invariants, and events. There is a need for a secondary component besides the models envisaged so far, it is called a *context*. As we shall see in section 5, contexts will play a very important rôle in the generic instantiation mechanism. In fact, the contexts associated with a given model define the way this model is *parameterized* and can thus be instantiated.

A context is made of the following elements: (1) a name, (2) a list of distinct carrier sets, collectively denoted by $s$, (3) a list of distinct constants, collectively denoted by $c$, and (4) a list of named properties, collectively denoted by $P(s, c)$. This is illustrated in Fig. 11

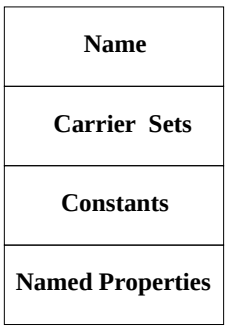



**Fig. 11.** Context

The carrier sets are just represented by their name. The different carrier sets of a context are completely independent. The only requirement concerning such sets is that they are supposed to be non-empty. The constants are defined, usually non-deterministically, by means of the properties $P(s, c)$, which are predicates.

Each model may reference a context. When it is the case, a model is said to "see" that context. When a model M sees a context C, then all carrier sets and constants defined in C can be used in M. In Fig.12, you can see the contents of models and contexts and their relationship.

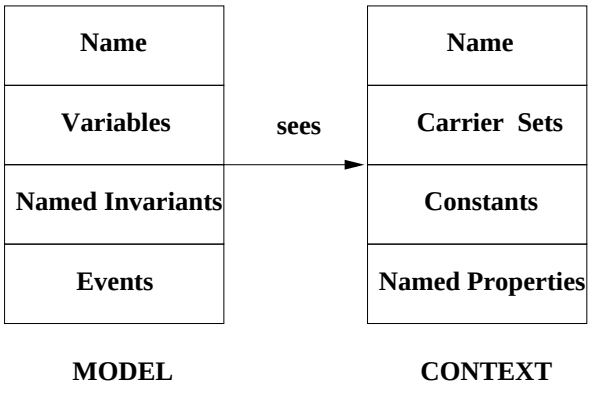



**Fig. 12.** Model and Context Relationship

### 2.6 Consistency Proofs for an Event System: Feasibility and Invariance Preservation

Let M be a model with variables $v$, seeing a context C with carrier sets $s$ and constants $c$. The properties of constants are denoted by $P(s, c)$ and the invariant by $I(s, c, v)$. Let E be an event of M with guards $G(s, c, v)$ and before-after predicate $R(s, c, v, v')$. When dealing with an array of events, the guards $G(s, c, v)$ are extended with corresponding array conditions.

We first have to express that, under the properties $P(s, c)$, the invariant $I(s, c, v)$, and the guard $G(s, c, v)$, the before-after predicate indeed yields at least one after value $v'$ defined by the before-after predicate $R(s, c, v, v')$. This is the feasibility statement FIS. We then express that the invariant is maintained. This is the invariant preservation statement INV. These two statements are shown in Fig. 13.

| | |
|---|---|
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ G(s,c,v) \ \Rightarrow \ \exists v' \cdot R(s,c,v,v')$ | FIS |
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ G(s,c,v) \ \wedge \ R(s,c,v,v') \ \Rightarrow \ I(s,c,v')$ | INV |

**Fig. 13.** Feasibility and Invariant Preservation Statements

In **Appendix 1**, we give the special forms of these laws for the various kinds of event we may have (deterministic or non-deterministic).

There is a special rule for the initialization event. Let $RI(s, c, v')$ denote the after predicate of the generalized substitution associated with this event. The two simplified statements to prove, INI_FIS and INI_INV, are given in Fig. 14.

| | |
|---|---|
| $P(s,c) \ \Rightarrow \ \exists \, v' \cdot RI(s,c,v')$ | INI_FIS |
| $P(s,c) \ \wedge \ RI(s,c,v') \ \Rightarrow \ I(s,c,v')$ | INI_INV |

**Fig. 14.** Feasibility and Invariant Preservation Statements for the Initialization

It is sometimes useful to state that the model which has been defined is deadlock free, that it can run for ever. This is very simply done by stating that the disjunction of the event guards always hold under the properties of the constant and the invariant. This is shown on Fig. 15 where $G_1(s, c, v)$, ..., $G_n(s, c, v)$ denote the guards of the events.

| | |
|---|---|
| $P(s,c) \ \wedge \ I(s,c,v) \ \Rightarrow \ G_1(s,c,v) \ \vee \ \ldots \ \vee \ G_n(s,c,v)$ | DLKF |

**Fig. 15.** Deadlock Freeness

# 3 Refinement

In this section we define the refinement of models and contexts. This is first described in section 3.1. In sections 3.2, 3.3, 3.4, and 3.5 we present in more detail the refinement of events and the rules to be followed in order to prove that a refinement is correct. Finally, in sections 3.6 and 3.7 we study a number of extra concepts related to refinements.

## 3.1 Model and Context Refinements

From a given model M, a new model N can be built and asserted to be a refinement of M. Model M will be said to be an *abstraction* of N, and model N will be said to be a *refinement* of M or a *concrete version* of it. Likewise, context C, seen by a model M, can be refined to a context D, which may be seen by N.

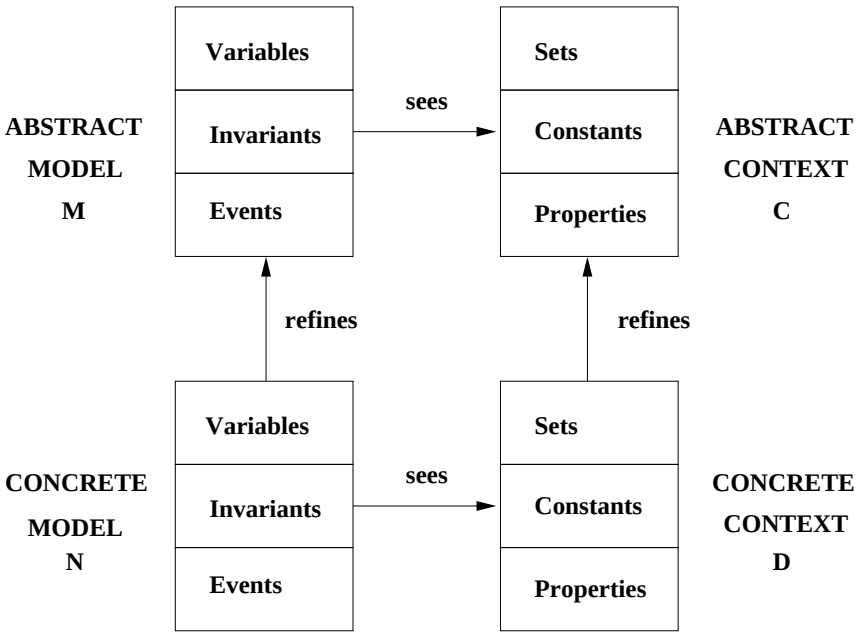



**Fig. 16.** Model and Context Refinements

This is represented in Fig. 16. Note that it is not necessary to refine context C when refining model M. In this restricted case, model N just sees context C as does its abstraction M. This is illustrated in Fig. 17.

The sets and constants of an abstract context are kept in its refinement. In other words, the refinement of a context just consists of adding new carrier sets and new constants to existing sets and constants. These are defined by means of new properties. In the case illustrated on Fig 16, model N, which sees context D, can thus use all carrier sets and constants defined in D as well as in C. From now on, to simplify matters, $s$ will denote the *accumulated* carrier sets, and $c$ will denote the *accumulated* constants seen from a model refinement.

The situation is not the same when refining models. The concrete model N (which supposedly sees concrete context D) has a collection of state variables $w$, which must be *completely distinct* (in first approximation) from the collection $v$ of variables in the abstraction M. Model N also has an invariant dealing with

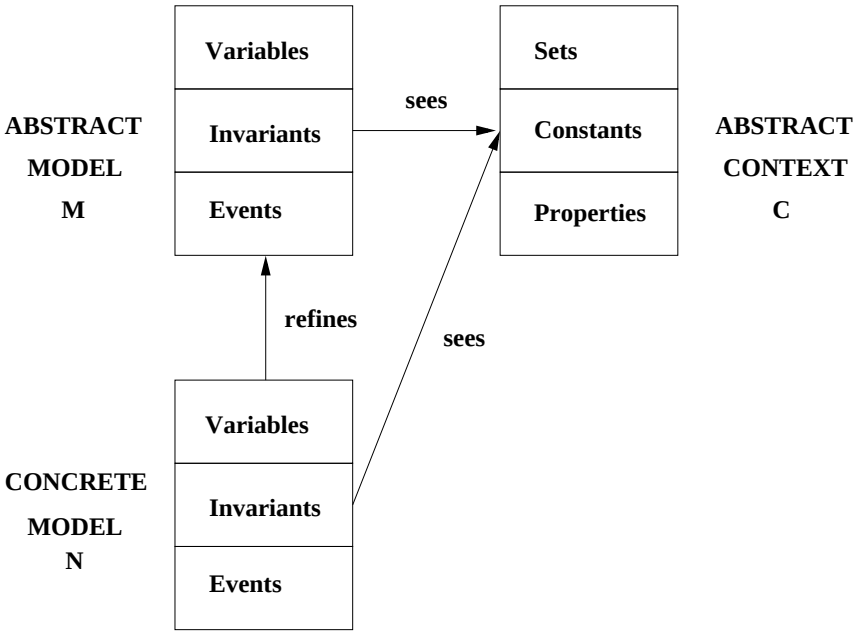



**Fig. 17.** Special Case of Model and Context Refinements

these variables $w$. But contrary to the case of abstract model M where the invariant exclusively depended on the variables $v$ of this model, this time it is possible to have the invariant of N, not only depending on variables $w$ of N, but also on the variables $v$ of its abstraction M. This is the reason why we collectively name this invariant of N the *gluing invariant* $J(s, c, v, w)$: it "glues" the state of the concrete model N to that of its abstraction M.

The development process we have seen so far was limited to two levels: an abstraction and its refinement. Of course, this process can be extended to more refinements as shown in Fig. 18. Note however that a gluing invariant links two successive models only. In other words, the variables mentioned in a gluing invariant are only those of the corresponding model and of its abstraction.

### 3.2 Refinement of Existing Events

The new model N has a number of events. Each event in the abstract model M has to be refined by one or several events in the concrete model N. This is illustrated in Fig. 19. It means that when proposing an event in a concrete model one must say explicitely which event it is supposed to refine (if any).

Suppose we have an abstract event with guard $G(s, c, v)$ and before-after predicate $R(s, c, v, v')$ and a refining concrete event with guard $H(s, c, w)$ and before-after predicate $S(s, c, w, w')$. The refinement statements to prove are shown in Fig. 20. The first law, FIS_REF, expresses that the refined event is feasible. The second and third laws, GRD_REF and INV_REF express the correct refinement of the concrete event with respect to the corresponding abstract one. In **Appendix 2** we give the various simplifed forms of these laws fort the various cases of event refinement.

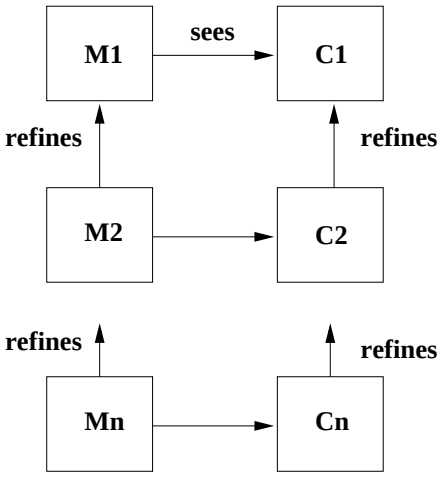



**Fig. 18.** Model and Context Refinements

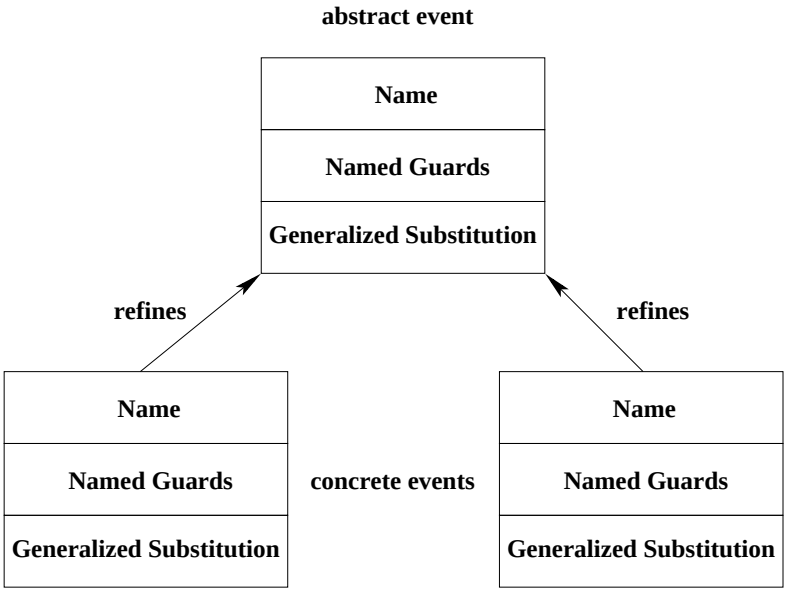



**Fig. 19.** An abstract event is refined by one or several concrete events

| | |
|---|---|
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,v,w) \ \wedge \ H(s,c,w)$<br>$\Rightarrow$<br>$\exists\, w' \cdot S(s,c,w,w')$ | FIS_REF |
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,v,w) \ \wedge \ H(s,c,w)$<br>$\Rightarrow$<br>$G(s,c,v)$ | GRD_REF |
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,v,w) \ \wedge \ H(s,c,w) \ \wedge \ S(s,c,w,w')$<br>$\Rightarrow$<br>$\exists\, v' \cdot (\, R(s,c,v,v') \ \wedge \ J(s,c,v',w')\,)$ | INV_REF |

**Fig. 20.** Refinement laws

### 3.3 Merging and Refining Existing Events

It is also possible for several abstract events to be merged before being refined to a single concrete event. This is shown in Fig. 21. For this to be possible however, the abstract merging events must fulfil a special constraint: all their generalized substitutions must be identical.

This process of merging and refining is made of two phases. The merging events are first implicitly transformed into a single abstract merged event. This event has a guard formed by taking the disjunction of the guards of the merging events. It has the same generalized substitution as the ones of the merging events. This is indicated in Fig. 22. Then this abstract merging event is refined as explained in section 3.2.

Note that the merging of two arrays of events require that they have tha same indices and the same array conditions.

### 3.4 Introducing New Events in a Refinement

New events can be introduced in a refinement. In this case, the refinement mechanism is slightly different from the one described in sections 3.2 and 3.3 for events already existing in the abstraction. This kind of refinement has two special constraints which are the following: (1) each new event refines an implicit skip event, and (2) the new events should *not together diverge* (run for ever) since then the abstract events could possibly never occur.

We now formalize these two constraints. Suppose we have an abstract model M seeing a context C as above. This model is refined to a more concrete model N seeing the refinement D of context C, again as above. In the refined model N, we supposedly have a new event with guard $H(s,c,w)$ and before-after predicate $S(s,c,w,w')$. The first constraint (refining skip) leads to refinement statements as indicated in Fig. 23. Note that these laws are just special cases of similar laws given in Fig. 20.

The second constraint (non-divergence of the new events), imposes to exhibit a variant $V(s,c,w)$, which is a well-founded structure (e.g. $\mathbb{N}, \leq$). And it is then necessary to prove that each new event decreases that *same* variant. This leads to a non-divergence statement as indicated in Fig. 24. Note that more generally, the variant could be any expression whcih is proved to be decreased by a well-founded relation.

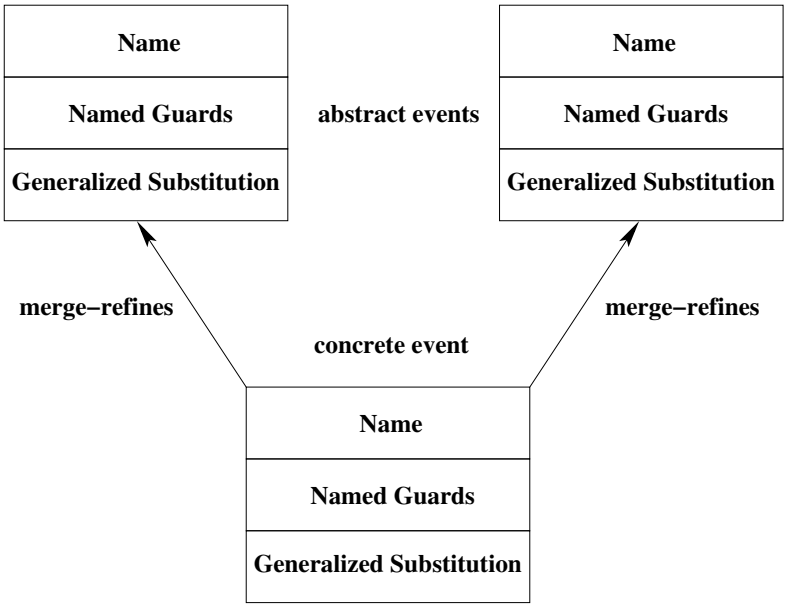



**Fig. 21.** Several abstract events are merged and then refined to a concrete event

| Abstract merging events | Abstract merged event |
|---|---|
| $\mathsf{E} \mathrel{\widehat{=}}$ **when** $G(v)$ **then** $S(v)$ **end**<br>$\mathsf{F} \mathrel{\widehat{=}}$ **when** $H(v)$ **then** $S(v)$ **end** | $\mathsf{EF} \mathrel{\widehat{=}}$ **when** $G(v) \lor H(v)$ **then** $S(v)$ **end** |

**Fig. 22.** Abstract merging of two events

| | |
|---|---|
| $P(s,c) \land I(s,c,v) \land J(s,c,v,w) \land H(s,c,w)$<br>$\Rightarrow$<br>$\exists w' \cdot S(s,c,w,w')$ | FIS_REF |
| $P(s,c) \land I(s,c,v) \land J(s,c,v,w) \land H(s,c,w) \land S(s,c,w,w')$<br>$\Rightarrow$<br>$J(s,c,v,w')$ | INV_REF |

**Fig. 23.** Refinement statement of a new event

| | |
|---|---|
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,v,w) \ \wedge \ H(s,c,w) \ \wedge \ S(s,c,w,w')$<br>$\Rightarrow$<br>$V(s,c,w) \in \mathbb{N} \ \wedge \ V(s,c,w') < V(s,c,w)$ | WFD_REF |

**Fig. 24.** Non-divergence statement

### 3.5 Relative Deadlock Freeness

The relative deadlock freeness is the property which says that a concrete model cannot deadlock more often than its abstraction. There are two kinds of relative deadlock freeness: the weak one and the strong one. For each abstract event $\mathsf{E_i}$, we can state one or the other.

The weak relative deadlock freeness expresses that the guard $G_i(s,c,v)$ of an abstract event $\mathsf{E}_i$ implies the disjunction of the concrete guards $H_1(s,c,w), \ldots, H_m(s,c,w)$ of the refined abstract events disjointed with the disjunction of the guards $N_1(s,c,w), \ldots, N_n(s,c,w)$, of the new events. This is stated in Fig. 25.

| | |
|---|---|
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,v,w) \ \wedge \ G_i(s,c,v)$<br>$\Rightarrow$<br>$H_1(s,c,w) \ \vee \ \ldots \ \vee \ H_m(s,c,w) \ \vee \ N_1(s,c,w) \ \vee \ \ldots \ \vee \ N_n(s,c,w)$ | W_DLK_$\mathsf{E}_i$ |

**Fig. 25.** Weak relative deadlock freeness

The strong relative deadlock freeness expresses that the guard $G_i(s,c,v)$ of an abstract event $\mathsf{E}_i$ implies the disjunction of the guard $H_i(s,c,w)$, of the event $\mathsf{F}_i$ refining $\mathsf{E}_i$, and of those guards, $N_1(s,c,w)$, $\ldots, N_n(s,c,w)$, of the new events. This is stated in Fig. 26. Note that when event $\mathsf{E_i}$ is refined by several

| | |
|---|---|
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,v,w) \ \wedge \ G_i(s,c,v)$<br>$\Rightarrow$<br>$H_i(s,c,w) \ \vee \ N_1(s,c,w) \ \vee \ \ldots \ \vee \ N_n(s,c,w)$ | S_DLK_$\mathsf{E}_i$ |

**Fig. 26.** Strong relative deadlock freeness

concrete events as explained in section 3.2, then $H_i(s,c,w)$ in Fig. 26 stands for the disjunction of the guards of these events. Arrays of events (see section 2.1) always follow strong relative deadlock freeness.

### 3.6 Anticipating New Events

Some of the new events introduced in a model $\mathsf{N}$ are said to be *anticipating events*. Such events are refining the implicit event skip as other non-anticipating new events do. But they can keep the variant introduced with $\mathsf{N}$ either smaller (as other new event do) or *unchanged*. This constraint of anticipating events must

hold in further refinements with regards to the introduced variants. At some refining stage however, say P, a previous anticipating event can become "new". Its only constraint then is to strictly decrease the variant introduced at this stage, but it is not required that it refines skip.

### 3.7 External Variables and External Events

In this section, we introduce external variables and external events. These features will play an important rôle in the mathematical justification of refinement and in the decomposition mechanism (section 4).

In each model M, the variables are partitioned in two categories: the *external variables*, $e$, and the internal variables, $i$. Let model N be a refinement of model M, and let $f$ be the external variables of N and $j$ its internal variables. The nature of the external variables $e$ of M just implies that these variables are *functionally dependent* of the external variables $f$ of N in the gluing invariant linking both models. This is expressed by means of the following gluing invariant where $h$ is supposed to be a function which does not depend on the internal variables $i$ or $j$:

$$J(s,c,i,j) \ \wedge \ e = h(s,c,f)$$

Besides external variables, there exist some *external events*. Such events only depend on the external variables. In other words, their guards and generalized substitutions do not use internal variables. External events are refined to other external events in the most general terms. Suppose we have an external event in model M with guard $G(s,c,e)$ and before-after predicate $R(s,c,e,e')$. It is then refined to an external event with guard $G(s,c,h(f))$ and before-after predicate $R(s,c,h(f),h(f'))$. The latter is indeed a refinement of the former since we can prove the three laws FIS_REF, FIS_GRD, and FIS_INV of Fig. 20:

$$P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,i,j) \ \wedge \ e = h(s,c,f) \ \wedge \ G(s,c,h(f)) \ \Rightarrow \ \exists f' \cdot R(s,c,h(s,c,f),h(s,c,f'))$$

$$P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,i,j) \ \wedge \ e = h(s,c,f) \ \wedge \ G(s,c,h(s,c,f)) \ \Rightarrow \ G(s,c,e)$$

$$P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,i,j) \ \wedge \ e = h(s,c,f) \ \wedge \ G(s,c,h(f)) \ \wedge R(s,c,h(s,c,f),h(s,c,f'))$$
$$\Rightarrow$$
$$\exists e' \cdot ( R(s,c,e,e') \ \wedge \ J(s,c,i,j) \ \wedge \ e' = h(s,c,f') )$$

The second and third laws are easily provable. The first one however, the feasibility law, imposes that we have the following extra law:

$$P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ G(s,c,e) \ \wedge \ R(s,c,e,e') \ \Rightarrow \ \exists f' \cdot (e' = h(s,c,f'))$$

Now the feasibility is easily provable since we supposedly already have the feasibility of event in M with guard $G(s,c,e)$ and before-after predicate $R(s,c,e,e')$ according to FIS on Fig. 13:

$$P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ G(s,c,e) \ \Rightarrow \ \exists e' \cdot R(s,c,e,e')$$

# 4 Decomposition

## 4.1 Partitioning the events and the Variables

At some point in a development, it is appropriate to decompose a model M into several sub-models. For the sake of simplicity, we suppose that we only have two sub-models, N and P. The decomposition is done by partitioning the events of M in two groups: those going into N and those going into P.

Likewise, we must partition the variables of M in two groups: those going into N and those going into P. However, this partitioning is in general not possible since we always have some variables that must be shared by both sub-models.

## 4.2 External Variables and Events

The shared variables mentioned in the previous section are present in both sub-models under the form *external variables*, which we have introduced in section 3.7. Together with the external variables, we must also introduce some *external events*, which simulate in each sub-model the way the external variables are handled in the other. This is illustrated in Fig. 27. As can be seen, variable $v2$ is external. We also have external events e3_ext in N and e2_ext in P.

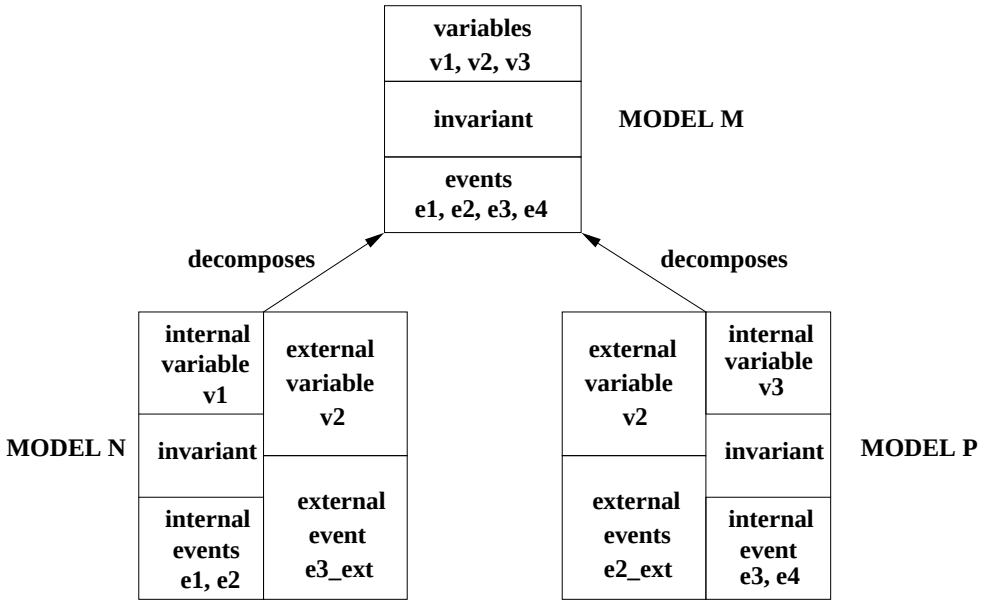



**Fig. 27.** Decomposition

## 4.3 Possible Recomposition

Once the decomposition is accomplished, the sub-models can be refined independently. For example, model N is refined to model NR with internal variables $w1$ and external variables $w2$. And model P is refined to model PR with internal variables $w3$ and external variables $w2$. There are some constraints which

must be observed however: the shared variables $v2$ which are external and common in both decomposed models must be *refined in the same way*. This is very simply accomplished by having the *same functional gluing invariant* $v2 = h(w2)$ in both sub-models.

It is then possible to re-compose models NR and PR to form model MR. This is done by conjoining the invariants of both models and removing the external events. But we now have to prove that the re-composed model MR is indeed a refinement of the original model M. In order to do so, it is sufficient to prove the following:

| External events e3_ext in N and e2_ext in P are refined to events e3 and e2 in M. |
|---|

Decomposition thus appears to be just an abstraction as indicated on Fig. 28.

## 5 Generic Instantiation

Generic instantiation is another proposal for solving the difficulties raised by the construction of large models. Suppose we have done an abstract development A with models M1 to Mn and corresponding contexts C1 to Cn as shown on Fig. 29.

This development is in fact parameterized by the carrier sets $s$ and the constants $c$ that have been accumulated in contexts C1 to Cn. This development is said to be *generic* with regards to such carrier sets and constants. Remember that such sets are completely independent of each other and have no properties except that they are supposed to be non-empty. The constants are defined by means of some properties $P(s, c)$, which stand here for all properties accumulated in contexts C1 to Cn. In fact, in all our proofs of this development, $s$ and $c$ appear as *free variables*. Moreover, the constants properties $P(s, c)$ appear as assumptions in all statements to be proved, which are thus of the following form as can be seen in proof obligations FIS and INV (Fig. 13), FIS_REF, FIS_REF, and INV_REF (Fig. 20), WFD_REF (Fig. 24), W_DLK_E$_i$ and S_DLK_E$_i$ (Fig. 25 and 26):

$$\underline{P(s, c)} \ \wedge \ A(s, c, \ldots) \ \Rightarrow \ B(s, c, \ldots)$$

Suppose now that in another development B, we reach a situation with model N seeing a certain context D (after some model and context refinements), as shown on Fig. 30.

The accumulated sets and constants in context D are denoted by $t$ and $d$ respectively. And the accumulated properties in context D are denoted by $Q(t, d)$. We might figure out at this point that a nice continuation of Development B would simply consist in *reusing* Development A with some *slight changes* consisting of instantiating sets $s$ and constants $c$ of Development A with expressions $S(t, d)$ and $C(t, d)$ depending on sets and constants $t$ and $d$ of Development B.

Let M1′, ... Mn′ be the models of Development A after performing the instantiations on the various invariants and events which can be found in M1 to Mn. The effective reuse is that shown in Fig. 31.

As can be seen, instantiated models M1′, ... Mn′ implicitly "see"context D. It remains, of course, to prove now that model M1′ refines model N. Once this is successfully done, we would like to resume Development B after Mn′. To do so, it is then necessary to prove that all feasibility, invariant, and refinement proofs performed in the Development A are still valid after the instantiation. Remember that all statements proved in the Development A were of the following form:

$$P(s, c) \ \wedge \ A(s, c, \ldots) \ \Rightarrow \ B(s, c, \ldots)$$

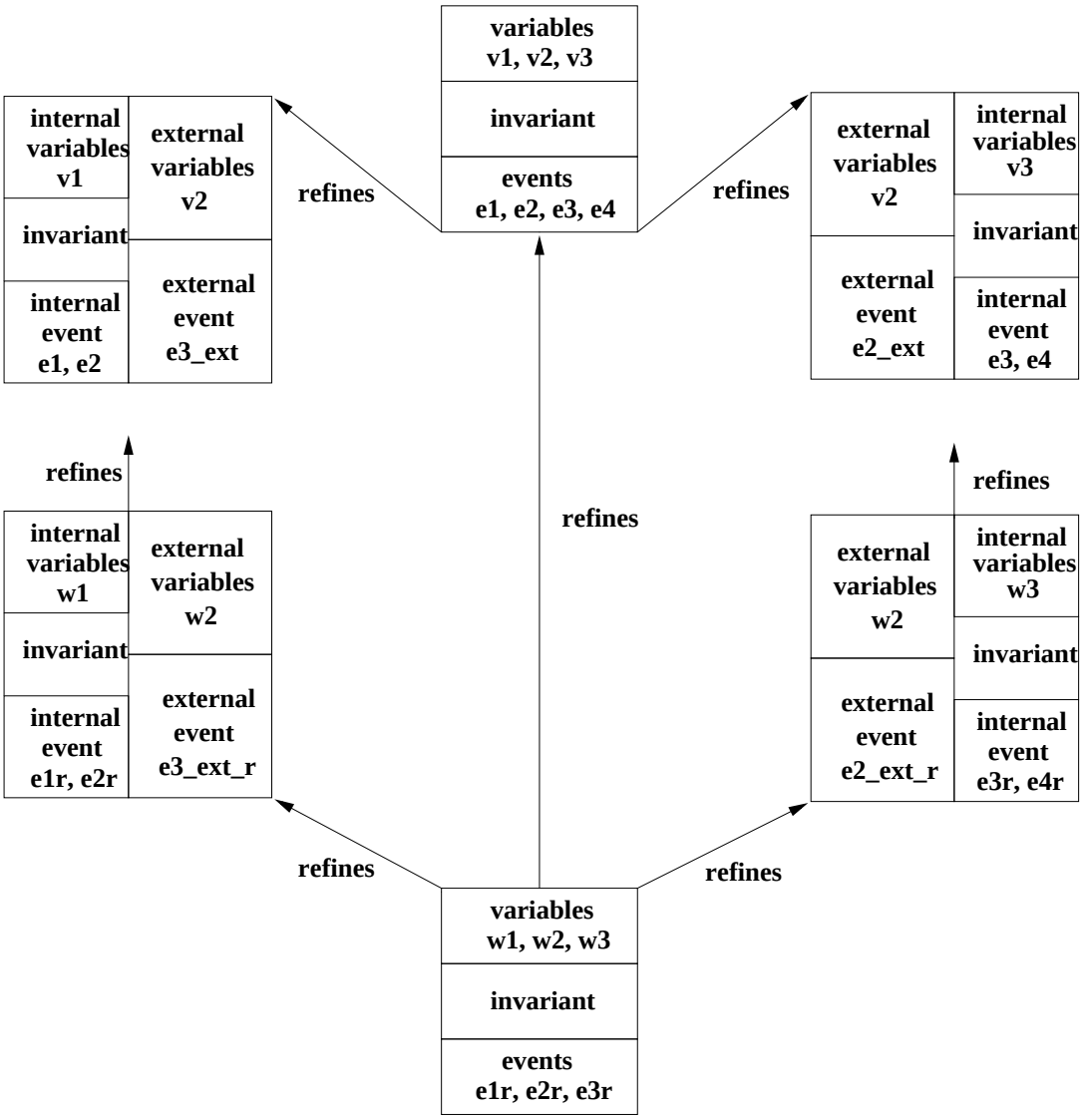



**Fig. 28.** Recomposition

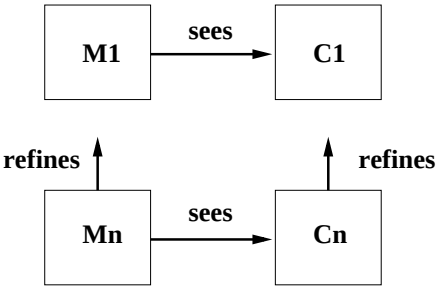



**Fig. 29.** Development A

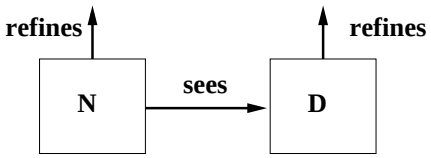



**Fig. 30.** Development B

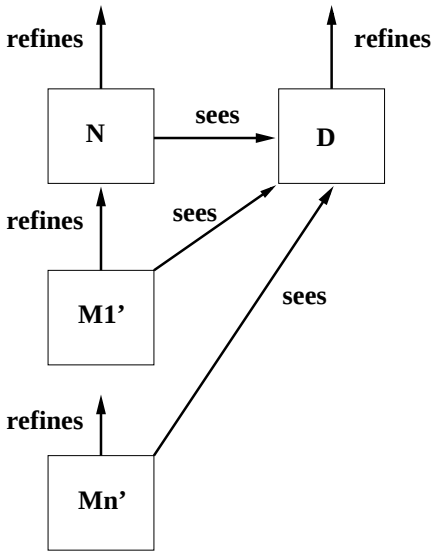



**Fig. 31.** Generic Instantiation of Development A as a continuation of Development B

As a consequence, they all look as follows after instantiation:

$$P(S(t,d),C(t,d)) \ \wedge \ A(S(t,d),C(t,d),\ldots) \ \Rightarrow \ B(S(t,d),C(t,d),\ldots)$$

But we have now to remove $P(S(t,d),C(t,d))$, since contexts C1 to Cn have disappeared as shown on Fig. 31, and replace it by the new set and constant properties $Q(t,d)$, namely:

$$Q(t,d) \ \wedge \ A(S(t,d),C(t,d),\ldots) \ \Rightarrow \ B(S(t,d),C(t,d),\ldots)$$

In order to prove this statement from the previous one, it is just sufficient that $Q(t,d)$ implies $P(S(t,d),C(t,d))$. In other words and quite intuitively, the instantiated properties of the constants of Development A should become mere theorems in Development B. This is indicated as follows:

| | |
|---|---|
| $Q(t,d) \ \Rightarrow \ P(S(t,d),C(t,d))$ | INS |

**Appendix 1: Special Cases of the Invariant Laws**

We instantiate the general forms of the laws INV and FIS of section 2.6, namely:

| | |
|---|---|
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ G(s,c,v) \ \Rightarrow \ \exists v' \cdot R(s,c,v,v')$ | FIS |
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ G(s,c,v) \ \wedge \ R(s,c,v,v') \ \Rightarrow \ I(s,c,v')$ | INV |

to the various shapes of the generalized substitutions involved in an event. We have two cases to consider (since skip trivially maintains an invariant).

(1) Given a deterministic event of the form:

| |
|---|
| **when** $G(s,c,v)$ **then** $x := E(s,c,v)$ **end** |

then law FIS is trivially true and law INV simplifies to the following:

| | |
|---|---|
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ G(s,c,v) \ \Rightarrow \ [x := E(s,c,v)]\, I(s,c,v)$ | INV_1 |

(2) Given a non-deterministic event of the form

| |
|---|
| **when** $G(s,c,v)$ **then**<br>  **any** $t$ **where** $P(t,s,c,v)$ **then** $x := E(t,s,c,v)$ **end**<br>**end** |

then law FIS and INV becomes the following:

| | |
|---|---|
| $I(s,c,v) \ \wedge \ G(s,c,v) \ \Rightarrow \ \exists t \cdot P(t,s,c,v)$ | FIS_2 |
| $I(s,s,v) \ \wedge \ G(s,c,v) \ \wedge \ P(t,s,c,v) \ \Rightarrow \ [x := E(t,s,c,v)]I(s,c,v)$ | INV_2 |

**Appendix 2: Special Cases of the Refinement Laws**

We instantiate the general forms of the refinement laws FIS_REF, GRD_REF, and INV_REF given in section 3.2, namely:

| | |
|---|---|
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,v,w) \ \wedge \ H(s,c,w)$<br>$\Rightarrow$<br>$\exists w' \cdot S(s,c,w,w')$ | FIS_REF |
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,v,w) \ \wedge \ H(s,c,w)$<br>$\Rightarrow$<br>$G(s,c,v)$ | GRD_REF |
| $P(s,c) \ \wedge \ I(s,c,v) \ \wedge \ J(s,c,v,w) \ \wedge \ H(s,c,w) \ \wedge \ S(s,c,w,w')$<br>$\Rightarrow$<br>$\exists v' \cdot ( R(s,c,v,v') \ \wedge \ J(s,c,v',w') )$ | INV_REF |

to the various cases of event refinements. In fact, we have four cases to consider. In what follows in order to simplify things, we omit to mention the carrier sets $s$ and the constants $c$.

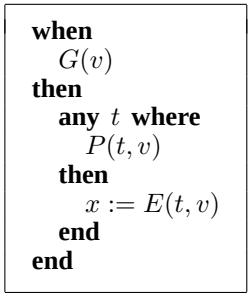
(1) Here is the first special case where the abstract and refined events are as follows :

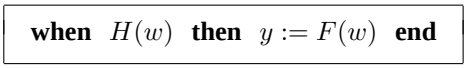
| | |
|---|---|
| **when** $G(v)$ **then** $x := E(v)$ **end** | **when** $H(w)$ **then** $y := F(w)$ **end** |

Then FIS_REF is trivially true and the other two laws are as follows:

| | |
|---|---|
| $I(v) \ \wedge \ J(v,w) \ \wedge \ H(w) \ \Rightarrow \ G(v)$ | GRD_REF_1 |
| $I(v) \ \wedge \ J(v,w) \ \wedge \ H(w) \ \Rightarrow \ [x,y := E(v),F(w)]J(v,w)$ | INV_REF_1 |

(2) The second special case of abstract and concrete events is as follows:



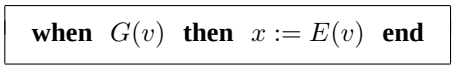

Again law FIS_REF is trivially true and the other two laws are as follows:

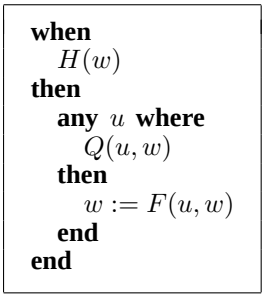
| | |
|---|---|
| $I(v) \wedge J(v,w) \wedge H(w) \Rightarrow G(v)$ | GRD_REF_2 |
| $I(v) \wedge J(v,w) \wedge H(w) \Rightarrow \exists t \cdot (\, P(t,v) \wedge [x, y := E(t,v), F(w)] J(v,w)\,)$ | INV_REF_2 |

Note that law INV_REF_2 can still be simplified provided the user gives some *witness* $W(w)$ for the existentially quantified variable $t$. This yields:

$$I(v) \wedge J(v,w) \wedge H(w) \Rightarrow P(W(w),v) \wedge [x, y := E(W(w),v), F(w)] J(v,w)$$

(3) The next special case of abstract and concrete events is the following:

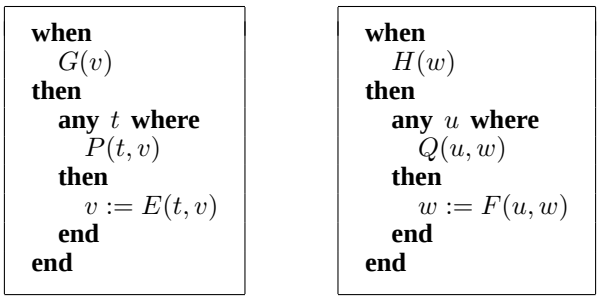



In that case, the three laws are simplified as follows:

| | |
|---|---|
| $I(v) \wedge J(v,w) \wedge H(w) \Rightarrow \exists u \cdot Q(u,w)$ | FIS_REF_3 |
| $I(v) \wedge J(v,w) \wedge H(w) \wedge Q(u,w) \Rightarrow G(v)$ | GRD_REF_3 |
| $I(v) \wedge J(v,w) \wedge H(w) \wedge Q(u,w) \Rightarrow [x, y := E(v), F(u,w)] J(v,w)$ | INV_REF_3 |

(4) The final special case of abstract and concrete events is then the following:




In that case, the three laws are simplified as follows:

| | |
|---|---|
| $I(v) \ \wedge \ J(v,w) \ \wedge \ H(w) \ \Rightarrow \ \exists u \cdot Q(u,w)$ | FIS_REF_4 |
| $I(v) \wedge J(v,w) \wedge H(w) \ \Rightarrow \ G(v)$ | GRD_REF_4 |
| $I(v) \wedge J(v,w) \ \wedge \ H(w) \ \wedge \ Q(u,w) \ \Rightarrow \ \exists t \cdot ( \, P(t,v) \wedge J(E(t,v), F(u,w)) \, )$ | INV_REF_4 |

Notice that one may provide some *witness* for the existentially quantified variables $t$. For example a witness $W(u,w)$ for $t$, would transform law INV_REF_4 into the following, which can be further decomposed:

$$I(v) \ \wedge \ J(v,w) \ \wedge \ H(w) \ \wedge \ Q(u,w) \ \Rightarrow \ P(W(u,w),v) \ \wedge \ J(E(W(u,w),v), F(u,w))$$

# (III) Event-B: Mathematical Model

J.-R. Abrial

April 2005

Version 1

# Event-B: Mathematical Model

## 1 Introduction

This document contains the mathematical justification of the laws which have been proposed in the companion document entitled *Event-B: Structure and Laws*. The laws of model and refinement consistencies are presented in section 2 and 3 under the form of a set-theoretic model and corresponding proofs. The laws of decomposition are then justified in section 4 under the forms of a proof performed within the First Order Predicate Calculus.

In this presentation to simplify matters, we suppose that we have models defined without contexts.

## 2 Initial Model Set-theoretic Representation

In this section our intention is to *formally justify* the invariant verification statements, namely FIS and INV, which were proposed in in section 2.6 of the document entitled *Event_B: Structure and Laws*. For this, we shall develop a set theoretic representation of the discrete models we have presented.

We suppose that the state variables $v$ are all together moving within a certain set $S$ involving the invariant $I(v)$. Each event can be represented by a certain binary relation $p$. The fact that the invariant $I(v)$ is preserved by event $p$ is simply formalized by saying that $p$ is a binary relation built on $S$:

$$p \subseteq S \times S$$

In order to link this set-theoretic representation to the previous verification statements, it suffices to formally define $S$ and $p$. It involves the invariant $I(v)$ for the set $S$, and the guard $G(v)$ and before-after predicate $R(v, v')$ for the relation $p$. This yields:

$$\begin{array}{lcl} S & = & \{\, v \mid I(v) \,\} \\ p & = & \{\, v \mapsto v' \mid I(v) \ \wedge \ G(v) \ \wedge \ R(v, v') \,\} \\ \mathsf{dom}\,(p) & = & \{\, v \mid I(v) \ \wedge \ G(v) \,\} \end{array}$$

The last equality states that $G(v)$ and $I(v)$ together denote the genuine domain of the relation $p$. But the domain of $p$ is defined to be the set

$$\{\, v \mid I(v) \ \wedge \ G(v) \ \wedge \ \exists\, v' \cdot R(v, v') \,\}$$

This leads to the following, which is exactly FIS:

| $I(v) \ \wedge \ G(v) \ \Rightarrow \ \exists v' \cdot R(v, v')$ | FIS |
|---|---|

And the translation of the predicate $p \subseteq S \times S$ yields exactly the desired result, INV, namely:

| $I(v) \ \wedge \ G(v) \ \wedge \ R(v, v') \ \Rightarrow \ I(v')$ | INV |
|---|---|

# 3 Refinement Set-theoretic Representation

As in the previous section for invariants, our intention is to formally *justify* in this section the refinement verification statements, namely FIS_REF, GRD_REF, INV_REF, which have been proposed in section 3.2 of the document entitled *Event_B: Structure and Laws*. For this, we shall extend the set theoretic representation of section 2.

## 3.1 Sets and Relations

We suppose, as above, that the abstract state variables are within the set $S$. But we now have to make a distinction between external and internal sets. So the set $S$ is defined to be the cartesian product of the sets $E$ (standing for external set) and $I$ (standing for internal set). The refined state variables are within a certain set $T$, which is also decomposed as the cartesian product of sets $F$ (refined external set) and $J$ (refined internal set). Let $p$ represent an abstract event binary relation and let $q$ be the corresponding refined event relation. We have then the following typing constraints:

$$p \ \in \ E \times I \ \leftrightarrow \ E \times I$$
$$q \ \in \ F \times J \ \leftrightarrow \ F \times J$$

Let $f$ and $g$ denote the functions projecting the set $E \times I$ on the set $E$ and the set $F \times J$ on the set $F$ respectively. Formally

$$f \ \in \ E \times I \rightarrow E$$
$$g \ \in \ F \times J \rightarrow F$$

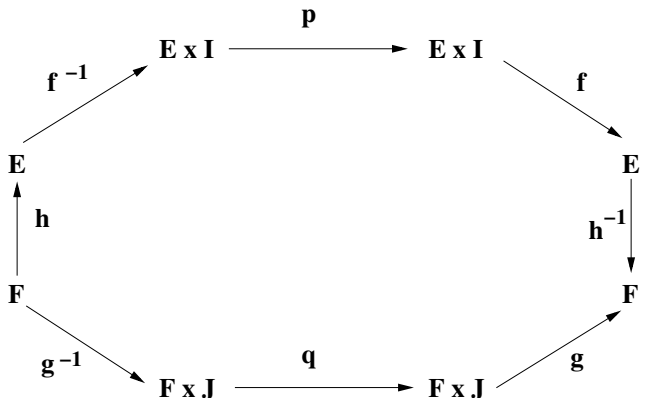
We have then the following:

$$\forall (e, i) \cdot ( \, e \in E \ \wedge \ i \in I \ \Rightarrow \ f(e \mapsto i \,) = e \,)$$
$$\forall (f, j) \cdot ( \, f \in F \ \wedge \ j \in J \ \Rightarrow \ g(f \mapsto j) = f \,)$$

The external sets $E$ and $F$ are related by a certain *total* function $h$, which is thus typed as follows:

$$h \in F \rightarrow E$$

All this can be illustrated in the following diagram:




### 3.2 Formal Definition of Refinement

In this section we present a formal definition of refinement, which is entirely based on the external sets. This will result in a kind of ultimate definition of refinement. In the next section we shall however derive some *sufficient refinement conditions* implying a formalization of the gluing invariant.

The previous diagram shows how one can link the external set $F$ to itself by navigating either through $h$, $f^{-1}$, $p$, $f$, and $h^{-1}$ in the abstraction or through $g^{-1}$, $q$, and $g$ in the refinement. These two compositions result in two binary relations built on $F$. Let us call them $\alpha$ and $\beta$ respectively. The definition of refinement follows: the event represented by the relation $p$ is refined by that represented by the relation $q$ if the relation $\beta$ is *included* in the relation $\alpha$. As can be seen, refinement is clearly defined *relative to the external sets*.

$$\underbrace{g^{-1} ; q ; g}_{\beta} \quad \subseteq \quad \underbrace{h ; f^{-1} ; p ; f ; h^{-1}}_{\alpha} \qquad \text{REF}$$

This means that every pair pertaining to the refined relation $\beta$ is also pertaining to the abstract relation $\alpha$. However, we have no equality between these relations because the refined relation $\beta$ might be more deterministic than its abstraction $\alpha$. We might have some loss of information between the refined model and its abstraction. What is important to note here is that no pair of external values in $\beta$ can be outside the abstraction $\alpha$: this constitutes the *essence of refinement*. If a pair of external values is linked through the refined event $q$, it must also be linked through the abstract event $p$. In other words, the refined event must not contradict the abstract one *from the point of view of the external sets*.

### 3.3 Sufficient Refinement Conditions

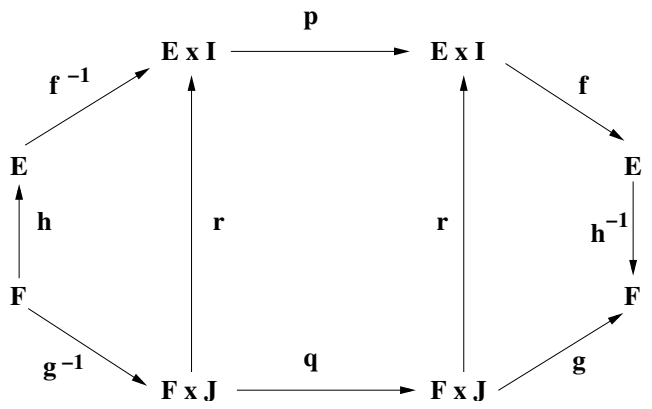
We are now going to define a sufficient refinement condition for refinement. Let $r$ be a *total* binary relation from the concrete set $F \times J$ to the abstract set $E \times I$. This relation formalizes the gluing invariant between the refined state and the abstract one. Formally

$$r \in F \times J \leftrightarrow\!\!\!\!\!\rightarrow E \times I$$

Note that the symbol "$\leftrightarrow\!\!\!\!\!\rightarrow$" is used to define the set of *total binary relations* from one set to another. The relation $r$ must be *compatible* with the function $h$ linking the external sets $F$ and $E$. In other words, if the pair $c \mapsto d$ is linked to the pair $a \mapsto b$ through $r$, then $c$ must be linked to $a$ through $h$, that is to say: $a = h(c)$. This can be formalised by means of the following condition:

| | |
|---|---|
| $r^{-1} ; g \subseteq f ; h^{-1}$ | C1 |

The introduction of the relation $r$ leads to the following diagram:




We now suppose that the following two extra conditions hold:

| | |
|---|---|
| $r^{-1} ; q \subseteq p ; r^{-1}$ | C2 |
| $g^{-1} \subseteq h ; f^{-1} ; r^{-1}$ | C3 |

It is then easy to prove that these conditions are *sufficient to ensure refinement*, namely condition REF above. It relies on the monotonicity of composition with regards to set inclusion and also on the associativity of composition:

$$
\begin{array}{lll}
 & \underline{g^{-1}\,;\,q\,;\,g} & \\
\subseteq & & \mathsf{C3} \\
 & h\,;\,f^{-1}\,;\,\underline{r^{-1}\,;\,q}\,;\,g & \\
\subseteq & & \mathsf{C2} \\
 & h\,;\,f^{-1}\,;\,p\,;\,\underline{r^{-1}\,;\,g} & \\
\subseteq & & \mathsf{C1} \\
 & h\,;\,f^{-1}\,;\,p\,;\,f\,;\,h^{-1} &
\end{array}
$$

But it happens that condition C3 can be deduced from condition C1 and from the totality of $r$:

$$
\begin{array}{lll}
 & r^{-1}\,;\,g \;\subseteq\; f\,;\,h^{-1} & \mathsf{C1} \\
\Rightarrow & & \text{Set Theory} \\
 & r\,;\,r^{-1}\,;\,g \;\subseteq\; r\,;\,f\,;\,h^{-1} & \\
\Rightarrow & & \mathsf{id}\,(T) \subseteq r\,;\,r^{-1} \quad \text{since} \quad r \in T \leftrightarrow\!\!\!\!\!\rightarrow S \\
 & g \;\subseteq\; r\,;\,f\,;\,h^{-1} & \\
\Leftrightarrow & & \text{Set Theory} \\
 & g^{-1} \;\subseteq\; h\,;\,f^{-1}\,;\,r^{-1} & \mathsf{C3}
\end{array}
$$

As a consequence, there only remains condition C2. In order to translate this condition and thus establish the verification statements FIS_REF, GRD_REF, and INV_REF, it suffices to link $E \times I$, $F \times J$, $p$, $q$, and $r$ with this new formulation. This yields:

$$
\begin{array}{|l|}
\hline
E \times I \;=\; \{\, v \mid I(v) \,\} \\
F \times J \;=\; \{\, w \mid \exists v \cdot (\, I(v) \;\wedge\; J(v,w) \,) \,\} \\
p \;=\; \{\, v \mapsto v' \mid I(v) \;\wedge\; G(v) \;\wedge\; R(v,v') \,\} \\
q \;=\; \{\, w \mapsto w' \mid \exists v \cdot (\, I(v) \;\wedge\; J(v,w) \,) \;\wedge\; H(w) \;\wedge\; S(w,w') \,\} \\
r \;=\; \{\, w \mapsto v \mid I(v) \;\wedge\; J(v,w) \,\} \\
\mathsf{dom}\,(q) \;=\; \{\, w \mid \exists v \cdot (\, I(v) \;\wedge\; J(v,w) \,) \;\wedge\; H(w) \,\} \\
\hline
\end{array}
$$

Note that the domain of the binary relation $r$ is $F \times J$. The binary relation $r$ is thus indeed a total relation as required. The domain of the binary relation $q$ is the set:

$$\{\, w \mid \exists v \cdot (\, I(v) \;\wedge\; J(v,w) \,) \;\wedge\; H(w) \;\wedge\; \exists w' \cdot S(w,w') \,\}$$

Thus our last constraint on the domain of $q$ leads to the following, which is exactly FIS_REF:

| | |
|---|---|
| $I(v) \;\wedge\; J(v,w) \;\wedge\; H(w) \;\Rightarrow\; \exists\, w' \cdot S(w,w')$ | FIS_REF |

The translation of condition C2, namely $r^{-1}\,;\,q \;\subseteq\; p\,;\,r^{-1}$, yields the following:

$$I(v) \ \wedge \ J(v,w) \ \wedge \ H(w) \ \wedge \ S(w,w') \ \Rightarrow \ G(v) \ \wedge \ \exists\, v' \cdot (\, R(v,v') \ \wedge \ J(v',w')\,)$$

According to condition FIS_REF it can be split as follows yielding exactly GRD_REF and INV_REF

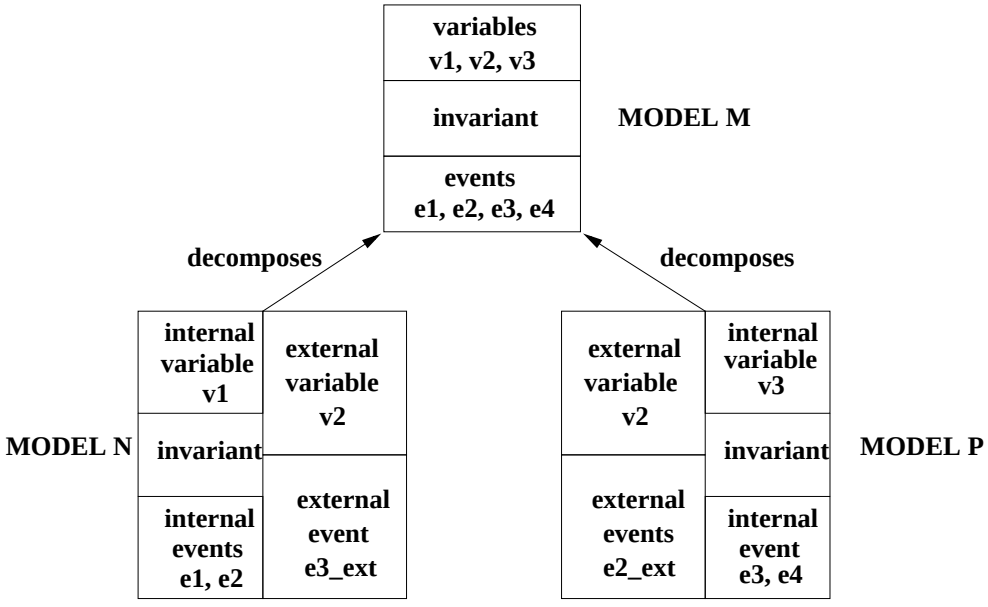
| | |
|---|---|
| $I(v) \ \wedge \ J(v,w) \ \wedge \ H(w) \ \Rightarrow \ G(v)$ | GRD_REF |
| $I(v) \ \wedge \ J(v,w) \ \wedge \ H(w) \ \wedge \ S(w,w') \ \Rightarrow \ \exists\, v' \cdot (\, R(v,v') \ \wedge \ J(v',w')\,)$ | INV_REF |

# 4 Decomposition

In this section we justify the decomposition presented in section 4 of the document entitled *Event_B: Structure and Laws*. We recommend the reader to read again this short section as we shall use in what follows the same convention as those used there.

## 4.1 Decomposing Model M into models N and P

We shall carry out the proof based on the example shown in Fig. 26 of the previous document which we reproduce here:




Suppose that the usage of the variables $v1$, $v2$ and $v3$ in model M is as follows:

| variable | events |
|---|---|
| $v1$ | e1, e2 |
| $v2$ | e2, e3 |
| $v3$ | e3, e4 |

| events | variables |
|---|---|
| e1 | $v1$ |
| e2 | $v1$, $v2$ |
| e3 | $v2$, $v3$ |
| e4 | $v3$ |

Model M is then decomposed into models N and P. Model N uses variable $v1$ as an internal variable and variable $v2$ as an external variable. It has events e1 and e2 plus an extra external events e3a (for e3 abstracted) dealing with variable $v2$ only. Event e3a is supposed to be refined by event e3. In other words, event e3a simulates in model N the behavior of event e3 in model P. This is summarized in the following table:

| model | internal variables | internal events | external variables | external events |
|---|---|---|---|---|
| N | $v1$ | e1, e2 | $v2$ | e3a |

Similarly, model P uses variable $v3$ as an internal variable and variable $v2$ as an external variable. It has events e3 and e4 plus an extra external events e2a (for e2 abstracted) dealing with variable $v2$ only. Event e2a is supposed to be refined by event e2. In other words, event e2a simulates in model P the behavior of event e2 in model N. This is summarized in the following table:

| model | internal variables | internal events | external variables | external events |
|---|---|---|---|---|
| P | $v3$ | e3, e4 | $v2$ | e2a |

It can easily be seen that models N and P are both refined by model M. This is so because events e1 and e2 of N are clearly refined by events e1 and e2 of M (they are the same); event e3a of N is refined *by construction* by event e3 of M; finally event e4 of M clearly refines skip in N since it deals with variables $v3$ which does not exist in N. And similarly for P. More precisely, let the guards and before-after predicates of the four events be the following in model M:

| events | guards in M | before-after predicates in M |
|---|---|---|
| e1 | $G_1(v1)$ | $E_1(v1, v1')$ |
| e2 | $G_2(v1, v2)$ | $E_2(v1, v2, v1', v2')$ |
| e3 | $G_3(v2, v3)$ | $E_3(v2, v2', v3, v3')$ |
| e4 | $G_4(v3)$ | $E_4(v3, v3')$ |

And they are the following in N and P

| events | guards in N | BA predicates in N | events | guards in P | BA predicates in P |
|---|---|---|---|---|---|
| e1 | $G_1(v1)$ | $E_1(v1, v1')$ | | | |
| e2 | $G_2(v1, v2)$ | $E_2(v1, v2, v1', v2')$ | e2a | $G_{2a}(v2)$ | $E_{2a}(v2, v2')$ |
| e3a | $G_{3a}(v2)$ | $E_{3a}(v2, v2')$ | e3 | $G_3(v2, v3)$ | $E_3(v2, v3, v2', v3')$ |
| | | | e4 | $G_4(v3)$ | $E_4(v3, v3')$ |

The condition expressing that event e2 is a refinement of event e2a is the following:

$$G_2(v1, v2) \ \wedge \ E_2(v1, v2, v1', v2') \ \Rightarrow \ G_{2a}(v2) \ \wedge \ E_{2a}(v2, v2')$$

### 4.2 Refining Models N and P to Models NR and PR

Suppose that we now refine N to NR. Model NR has variables $w1$ and $w2$ together with the gluing invariant $J(v1, w1, w2) \wedge v2{=}h(w2)$. Model NR has events e1r and e2r which are supposed to be refinements of e1 and e2 respectively, and also event e3ar, which is a refinement of event e3a. This can be summarized in the following table:

| model | int. variables | int. events | ext. variables | ext. events | gluing invariant |
|---|---|---|---|---|---|
| NR | $w1$ | e1r, e2r | $w2$ | e3ar | $J(v1, w1, w2) \wedge v2 = h(w2)$ |

Similarly, we refine P to PR. Model PR has variables $w3$ and $w2$ together with the gluing invariant $K(v3, w3, w2) \wedge v2{=}h(w2)$. Model PR has events e3r and e4r which are supposed to be refinements of e3 and e4 respectively, and also event e2ar, which is a refinement of event e2a. Notice that both gluing invariants $J$ and $K$ also depend on the external variable $w2$. This can be summarized in the following table:

| model | int. variables | int. events | ext. variables | ext. events | gluing invariant |
|---|---|---|---|---|---|
| PR | $w3$ | e3r, e4r | $w2$ | e2ar | $K(v3, w3, w2) \wedge v2 = h(w2)$ |

The guards and before-after predicates in NR and PR are as follows

| events | guards in NR | BA pred. in NR |
|---|---|---|
| e1r | $G_{1r}(w1)$ | $E_{1r}(w1, w1')$ |
| e2r | $G_{2r}(w1, w2)$ | $E_{2r}(w1, w2, w1', w2')$ |
| e3ar | $G_{3ar}(w2)$ | $E_{3ar}(w2, w2')$ |
| | | |

| events | guards in PR | BA pred. in PR |
|---|---|---|
| | | |
| e2ar | $G_{2ar}(w2)$ | $E_{2ar}(w2, w2')$ |
| e3r | $G_{3r}(w2, w3)$ | $E_{3r}(w2, w3, w2', w3')$ |
| e4r | $G_{4r}(w3)$ | $E_{4r}(w3, w3')$ |

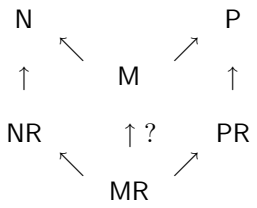
Let us now construct a model MR as follows. The state of MR is made of the three variables $w1$, $w2$ and $w3$. The invariant of MR is $J(v1, w1w2) \ \wedge \ K(v3, w3, w2) \ \wedge \ v2 = h(w2)$. The events of MR are e1r, e2r, e3r and e4r. Notice that e2ar and e3ar have been thrown away. This can be summarized in the following table:

| model | variables | events | gluing invariant |
|---|---|---|---|
| MR | $w1, v2, w3$ | e1r, e2r, e3r, e4r | $J(w1, w1, w2) \ \wedge \ K(v3, w3, w2) \ \wedge \ v2 = h(w2)$ |

The guards and before-after predicates in MR are as follows:

| **events** | **guards in** MR | **before-after predicates in** MR |
|---|---|---|
| e1r | $G_{1r}(w1)$ | $E_{1r}(w1, w1')$ |
| e2r | $G_{2r}(w1, w2)$ | $E_{2r}(w1, w2, w1', w2')$ |
| e3r | $G_{3r}(w2, w3)$ | $E_{3r}(w2, w3, w2', w3')$ |
| e4r | $G_{4r}(w3)$ | $E_{4r}(w3, w3')$ |

Clearly NR and PR are refined by MR, but it is not obvious that M is refined by MR, this is precisely what we have to prove. The situation is illustrated in the following diagram, where the arrows indicate a refinement relationship:




In what follows we shall prove that, provided e1r and e2r are refinements of e1 and e2 respectively in NR, then they also are correct refinements of e1 and e2 in MR. Similar proofs can be conducted for the other events of MR.

### 4.3 Event e1r is a Refinement of e1 in MR

The correct refinement condition of e1 to e1r within NR is the following:

$$\begin{array}{l} J(v1, w1, w2) \ \wedge \ v2 = h(w2) \ \wedge \ G_{1r}(w1) \ \wedge \ E_{1r}(w1, w1') \\ \Rightarrow \\ G_1(v1) \ \wedge \ \exists\, v1' \cdot (\, E_1(v1, v1') \ \wedge \ J(v1', w1', w2) \ \wedge \ v2 = h(w2)\,) \end{array}$$

Under this hypothesis, the following correct refinement condition of e1 to e1r within MR clearly holds:

$$\begin{array}{l} J(v1, w1, w2) \ \wedge \ \underline{K(v3, w3, w2)} \ \wedge v2 = h(w2) \ \wedge \ G_{1r}(w1) \ \wedge \ E_{1r}(w1, w1') \\ \Rightarrow \\ G_1(v1) \ \wedge \ \exists\, v1' \cdot (\, E_1(v1, v1') \ \wedge \ J(v1', w1', w2) \ \wedge \ \underline{K(v3, w3, w2)} \ \wedge \ v2 = h(w2)\,) \end{array}$$

As can be seen, condition $K(v3, w3, w2)$ can be extracted from the existential quantification in the consequent of this implication (this is so because $K(v3, w3, w2)$ does not contain any reference to the quantified variable $v1'$). It is then easily discharged because it is already present in the antecedent of the implication.

### 4.4 Event e2r is a Refinement of e2 in MR

The situation is a bit different in the case of the event e2: this is because this event modifies variable $v2$. Next is the correct refinement condition of e2 to e2r within NR:

$$\begin{array}{l} J(v1, w1, w2) \ \wedge v2 = h(w2) \ \wedge \ G_{2r}(w1, w2) \ \wedge \ E_{2r}(w1, w2, w1', w2') \\ \Rightarrow \\ G_2(v1, v2) \ \wedge \ \exists\, (v1', v2') \cdot (\, E_2(v1, v2, v1', v2') \ \wedge \ J(v1', w1', w2') \ \wedge v2' = h(w2')\,) \end{array}$$

This can be simplified as follows:

$$\begin{array}{l} J(v1, w1, w2) \ \wedge \ G_{2r}(w1, w2) \ \wedge \ E_{2r}(w1, w2, w1', w2') \\ \Rightarrow \\ G_2(v1, h(w2)) \ \wedge \ \exists\, v1' \cdot (\, E_2(v1, h(w2), v1', h(w2')) \ \wedge \ J(v1', w1', w2')\,) \end{array}$$

Under this hypothesis, the following correct refinement condition of e2 to e2r within MR must hold:

$$\begin{array}{l} J(v1, w1, w2) \ \wedge \ K(v3, w3, w2) \ \wedge \ v2 = h(w2) \ \wedge \ G_{2r}(w1, w2) \ \wedge \ E_{2r}(w1, w2, w1', w2') \\ \Rightarrow \\ G_2(v1, v2) \ \wedge \ \exists\, (v1', v2') \cdot (\, E_2(v1, v2, v1', v2') \ \wedge \ J(v1', w1', w2') \ \wedge \ K(v3, w3, w2') \ \wedge \ v2' = h(w2')\,) \end{array}$$

This can be simplified as follows:

$$\begin{array}{l} J(v1, w1, w2) \ \wedge \ \underline{K(v3, w3, w2)} \ \wedge \ G_{2r}(w1, w2) \ \wedge \ E_{2r}(w1, w2, w1', w2') \\ \Rightarrow \\ G_2(v1, h(w2)) \ \wedge \ \exists\, v1' \cdot (\, E_2(v1, h(w2), v1', h(w2')) \ \wedge \ J(v1', w1', w2') \ \wedge \ \underline{K(v3, w3, w2')}\,) \end{array}$$

As above with $K(v3, w3, w2)$ in section 4.3, the condition $K(v3, w3, w2')$ can be extracted from the existential quantification in the consequent of this implication. But this time the situation is different from the previous one in section 4.3 as we still have the condition $K(v3, w3, w2)$ in the antecedent, not $K(v3, w3, w2')$, so that the proof is not trivial. Again, the presence of $w2'$ in the consequent is due to the fact that $v2$ is modified by e2 and $w2$ by e2r. Fortunately, we have not yet exploited the fact that event e2a of model P is refined within model N by event e2. The condition was stated at the end of section 4.1:

$$\forall\, (v2, v2') \cdot (\, G_2(v1, v2) \ \wedge \ E_2(v1, v2, v1', v2') \ \Rightarrow \ G_{2a}(v2) \ \wedge \ E_{2a}(v2, v2')\,)$$

But we also know that external event e2ar is the *most general event* refining e2a under the gluing invariant $v2 = h(w2)$. As a consequence we have:

$$\begin{array}{lcl} G_{2ar}(w2) & \Leftrightarrow & G_{2a}(h(w2)) \\ E_{2ar}(w2, w2') & \Leftrightarrow & E_{2a}(h(w2), h(w2')) \end{array}$$

From this, we deduce

$$G_2(v1, h(w2)) \ \wedge \ E_2(v1, h(w2), v1', h(w2')) \ \Rightarrow \ G_{2ar}(w2) \ \wedge \ E_{2ar}(w, w2')$$

Finally, we also have not exploited the fact that event e2a of P is refined to e2ar in PR. This yields:

$$\begin{array}{l} K(v3, w3, w2) \ \wedge \ v2 = h(w2) \ \wedge \ G_{2ar}(w2) \ \wedge \ E_{2ar}(w2, w2') \\ \Rightarrow \\ G_{2a}(v2) \ \wedge \ \exists v2' \cdot ( \, E_{2a}(v2, v2') \ \wedge \ K(v3, w3, w2') \ \wedge \ v2' = h(w2') \, ) \end{array}$$

This can be simplified to the following:

$$K(v3, w3, w2) \ \wedge \ G_{2ar}(w2) \ \wedge \ E_{2ar}(w2, w2') \ \Rightarrow \ K(v3, w3, w2')$$

Putting all these conditions together yields the following to prove, which now holds "trivially":

$$\begin{array}{l} K(v3, w3, w2) \ \wedge \ G_{2ar}(w2) \ \wedge \ E_{2ar}(w2, w2') \ \Rightarrow \ \underline{K(v3, w3, w2')} \\ \\ G_2(v1, h(w2)) \ \wedge \ E_2(v1, h(w2), v1', h(w2')) \ \Rightarrow \ G_{2ar}(w2) \ \wedge \ E_{2ar}(w2, w2') \\ \\ J(v1, w1, w2) \ \wedge \ G_{2r}(w1, w2) \ \wedge \ E_{2r}(w1, w2, w1', w2') \Rightarrow \\ \qquad\qquad G_2(v1, h(w2)) \ \wedge \ \exists \, v1' \cdot ( \, E_2(v1, h(w2), v1', h(w2')) \ \wedge \ J(v1', w1', w2') \, ) \\ \\ J(v1, w1, w2) \ \wedge \ K(v3, w3, w2) \ \wedge \ G_{2r}(w1, w2) \ \wedge \ E_{2r}(w1, w2, w1', w2') \\ \\ \Rightarrow \\ \\ G_2(v1, h(w2)) \ \wedge \ \exists \, v1' \cdot ( \, E_2(v1, h(w2), v1', h(w2')) \ \wedge \ J(v1', w1', w2') \ \wedge \ \underline{K(v3, w3, w2')} \, ) \end{array}$$

# (IV) Event_B: Examples

J.-R. Abrial

April 2005

Version 3

# Event_B: Examples

## 1 Introduction

In this document, we provide some examples to illustrate what has been presented in the three other companion documents. In the first of this examples in section 2 a small reactive system, whose goal is to control cars on a bridge, is presented. The second example in section 4 shows how one can decompose a system. And in the third example in section 5 the usage of generic instantiation is presented.

## 2 Cars on a Bridge

### 2.1 Requirements

The system we are going to build is a piece of software connected to some equipment. Its goal is to control cars on a narrow bridge. This bridge is supposed to link the mainland to a small island.

| | |
|---|---|
| The system is controlling cars on a bridge connecting the mainland to an island | FUN-1 |

This controller is equipped with two traffic lights.

| | |
|---|---|
| The system is equipped with two traffic lights with two colors: green and red | EQP-1 |

One of the traffic lights is situated on the mainland and the other one on the island. Both are close to the bridge.

| | |
|---|---|
| The traffic lights control the entrance to the bridge at both ends of it | EQP-2 |

Drivers are supposed to obey the traffic light by not passing when a traffic light is red.

| | |
|---|---|
| Cars are not supposed to pass on a red traffic light, only on a green one | EQP-3 |

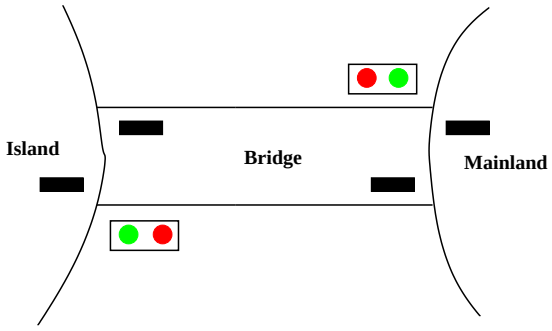
There are also some car sensors situated at both ends of the bridge.

| | |
|---|---|
| The system is equipped with four sensors with two states: on or off | EQP-4 |

These sensors are supposed to detect the presence of cars intending to enter or leave the bridge. There are four such sensors. Two of them are situated on the bridge and the other two are situated on the mainland and on the island respectively.

| | |
|---|---|
| The sensors are used to detect the presence of car entering or leaving the bridge | EQP-5 |

The pieces of equipment which have been described are illustrated on the following figure:




This system has two main constraints: the number of cars on the bridge and island is limited,

| | |
|---|---|
| The number of cars on bridge and island is limited | FUN-2 |

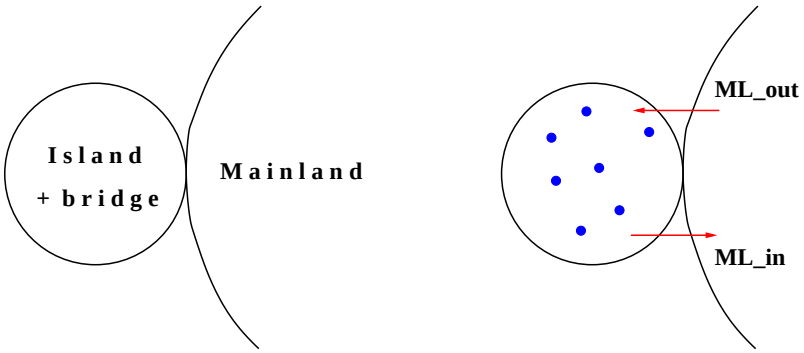
and the bridge is one way.

| | |
|---|---|
| The bridge is one way or the other, not both at the same time | FUN-3 |

### 2.2 Initial Model: Limiting the Number of Cars

The first model we are going to construct is very simple. We do not consider at all the various pieces of equipment, namely the traffic lights and sensors. Such equipment will be introduced in subsequent refinements. Likewise, we do not even consider the bridge, only a *compound* made of the bridge and the island together.

As a useful analogy, we suppose to see the situation from very high in the sky. Although we cannot see the bridge, we suppose however that we can "see" the cars in the island-bridge and observe the two transitions, ML_out and ML_in, corresponding to cars entering and leaving the island-bridge compound. All this is illustrated on the following figures:




**Formalizing the state.** The state is first made of a simple context containing a constant $d$ which is a natural number denoting the maximum number of cars allowed to be in the island-bridge at the same time. This is formalized by the property named prp0_1.

The state is also made of a variable $n$ denoting the actual number of cars in the island-bridge at a given moment. Two invariants named inv0_1 and inv0_2 are used to define the variable $n$. Invariant inv0_1 says that $n$ is a natural number. The *first basic requirement* of our system, namely FUN_2, is taken into account at this stage by stating in inv0_2 that the number $n$ of cars in the compound is always smaller than or equal to the maximum number $d$. Here is the formal state:

| | | |
|---|---|---|
| **constants:** $d$<br>**variables:** $n$ | prp0_1 : $d \in \mathbb{N}$ | inv0_1 : $n \in \mathbb{N}$<br>inv0_2 : $n \leq d$ |

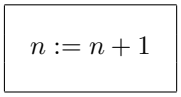
**Events.** At this stage we can observe two transitions corresponding to cars entering the island-bridge compound or leaving it. Here is an illustration of the situation just before and just after an occurrence of the first event, ML_out. As can be seen, the number of cars in the compound is incremented as a result of this event.

**Before** **After**

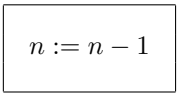
Likewise, here is the situation just before and just after an occurrence of the second event, ML_in. As can be seen, the number of cars in the compound is decremented as a result of this event.

**Before** **After**

These two events can then be defined in a simple way as follows:



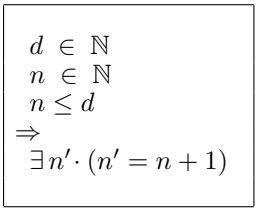

ML_out

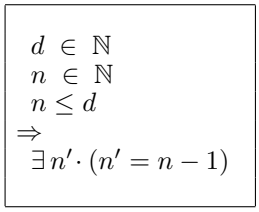



ML_in

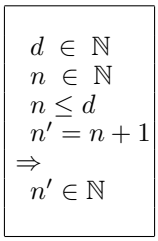
The before-after predicates corresponding to these event actions are straightforward, namely $n' = n + 1$ for ML_out, and $n' = n - 1$ for ML_in.

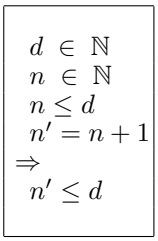
**Proving Invariant Preservation.** Rule FIS applied to both events leads to the following, which holds trivially:

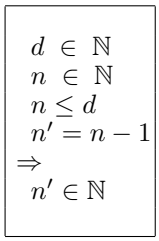


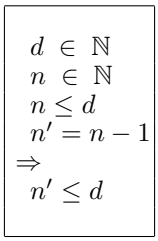




Invariant preservation rule INV applied to both events leads to the following statements:

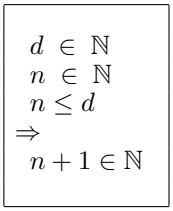


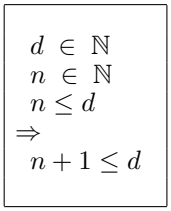



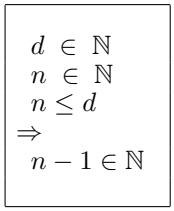



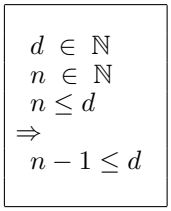




The variable $n'$ can be eliminated by replacing it by its value, yielding:




We notice that in the second case, $n + 1 \leq d$ cannot be proved when $n$ is already equal to $d$. And in the third case, $n - 1 \in \mathbb{N}$ cannot be proved when $n$ is already equal to 0. This is so because the proposed events ML_out and ML_in are too primitive.

**Improving the two Events** We have to add *guards* to our events. They denote the necessary conditions for these events to be enabled. For event ML_out to be enabled, it is required that $n$ be strictly smaller than $d$. And for event ML_in to be enabled, it is required that $n$ be strictly positive. All this is indicated in the following new versions of these events:

ML_out
**when**
$n < d$
**then**
$n := n + 1$
**end**

ML_in
**when**
$0 < n$
**then**
$n := n - 1$
**end**

The statements to prove by applying rule INV are modified accordingly and are now easily provable:

$$d \in \mathbb{N}$$
$$n \in \mathbb{N}$$
$$n \leq d$$
$$n < d$$
$$\Rightarrow$$
$$n + 1 \in \mathbb{N}$$

$$d \in \mathbb{N}$$
$$n \in \mathbb{N}$$
$$n \leq d$$
$$n < d$$
$$\Rightarrow$$
$$n + 1 \leq d$$

$$d \in \mathbb{N}$$
$$n \in \mathbb{N}$$
$$n \leq d$$
$$0 < n$$
$$\Rightarrow$$
$$n - 1 \in \mathbb{N}$$

$$d \in \mathbb{N}$$
$$n \in \mathbb{N}$$
$$n \leq d$$
$$0 < n$$
$$\Rightarrow$$
$$n - 1 \leq d$$

**Proving Deadlock Freeness** Since our events are now guarded, it means that our system might deadlock when both guard are together false. Clearly, we want to avoid this happening. We have thus to prove rule DLKF stating that one of the two guards is always true. In other words, cars can always either enter the compound or leave it. This is to be proved under the property of the constant and under the invariant:

$$d \in \mathbb{N}$$
$$n \in \mathbb{N}$$
$$n \leq d$$
$$\Rightarrow$$
$$n < d \ \lor \ 0 < n$$

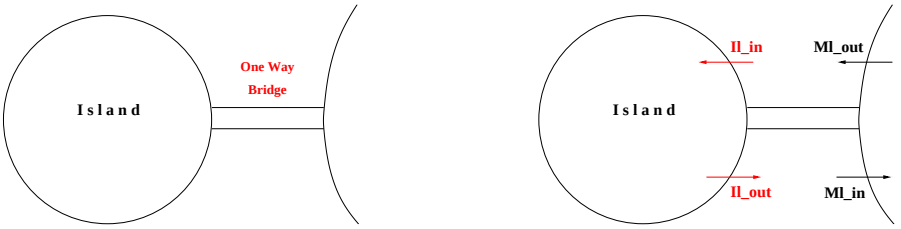
But we now discover that this statement cannot be proved when $d$ is zero, which is quite obvious since then no car can ever enter the compound nor, a fortiori, leave it. We have thus to add the following property, named prp0_2, which was obviously forgotten:

prp0_2 : $0 < d$

**Conclusion of the initial model** As we have seen, the proofs (or rather the failures of the proofs) allowed us to discover that our events were too primitive and also that one property was missing in the context for the constant $d$.

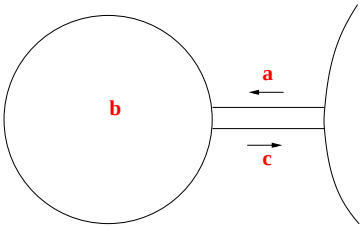
### 2.3 First Refinement: Introducing the One Way Bridge

In this first refinement, we introduce the bridge. This means that we are able to observe our system more accurately. Together with this more accurate observation, we can also see more events, namely cars entering and leaving the island. These events are called IL_in and IL_out. Note that events ML_out and ML_in which were present in the initial model still exist in this refinement: they now correspond to cars leaving the mainland and entering the bridge or leaving the bridge and entering the mainland. All this is illustrated in the following figures:




**Refining the state.** The state which was defined by the constant $d$ and variable $n$ in the initial model now becomes more accurate. The constant $d$ remains, but the variable $n$ is now replaced by three new variables. This is because now we can see cars on the bridge and on the island, something which we could not distinguish in the previous abstraction. Moreover, we can see where cars on the bridge are going: either towards the island or towards the mainland.

For these reasons, the state is now represented by means of three variables $a$, $b$, and $c$. Variable $a$ denotes the number of cars on the bridge and going to the island, variable $b$ denotes the number of cars on the island, and variable $c$ denotes the number of cars on the bridge and going to the mainland. This is illustrated on the following figure:




Formally the refined state is represented by a number of new invariants. First, variables $a$, $b$, and $c$ are all natural numbers. This is stated in invariant inv1_1, inv1_2, and inv1_3. Then we express in the, so-called, gluing invariant, that the sum of these variables is equal to the previous abstract variable $n$ which now disappears. This is expressed in invariant inv1_4. And finally, we state that the bridge is one way, this is our basic requirement FUN-3, by saying that $a$ or $c$ is 0. Clearly they cannot be both positive since the bridge is one way. Note that they can be both 0 however. This is expressed in invariant inv1_5. Here is the formalization of these invariants:

| | |
|---|---|
| **constants:** | $d$ |
| **variables:** | $a, b, c$ |

| | |
|---|---|
| inv1_1 : | $a \in \mathbb{N}$ |
| inv1_2 : | $b \in \mathbb{N}$ |
| inv1_3 : | $c \in \mathbb{N}$ |

| | |
|---|---|
| inv1_4 : | $a + b + c = n$ ; |
| inv1_5 : | $a = 0 \quad \vee \quad c = 0$ |

**Refining the Abstract Events.** The two abstract events ML_out and ML_in have now to be refined as they are not dealing with variable $n$ any more but with variables $a$, $b$, and $c$. Here is the proposed refinement of event ML_out, which is presented together with its abstraction.

abstract_ML_out
**when**
$n < d$
**then**
$n := n + 1$
**end**

concrete_ML_out
**when**
$a + b < d$
$c = 0$
**then**
$a := a + 1$
**end**

Likewise, here is the proposed refined version of event ML_in, which is also presented together with its abstraction.

abstract_ML_in
**when**
$0 < n$
**then**
$n := n - 1$
**end**

concrete_ML_in
**when**
$0 < c$
**then**
$c := c - 1$
**end**

**Proving that the Refinement of Abstract Events are Correct.** Applying Rule FIS_REF to the refined event holds trivially. Likewise, applying rule GRD_REF to both refined events leads to the following, which holds trivially:

$d \in \mathbb{N}$
$0 < d$
$n \in \mathbb{N}$
$n \leq d$
$a \in \mathbb{N}$
$b \in \mathbb{N}$
$c \in \mathbb{N}$
$a + b + c = n$
$a = 0 \ \lor \ c = 0$
$a + b < d$
$c = 0$
$\Rightarrow$
$n < d$

$d \in \mathbb{N}$
$0 < d$
$n \in \mathbb{N}$
$n \leq d$
$a \in \mathbb{N}$
$b \in \mathbb{N}$
$c \in \mathbb{N}$
$a + b + c = n$
$a = 0 \ \lor \ c = 0$
$0 < c$
$\Rightarrow$
$0 < n$

Applying rule INV_REF to both events leads to the following after some simplifications. Both statements hold trivially.

$d \in \mathbb{N}$
$0 < d$
$n \in \mathbb{N}$
$n \leq d$
$a \in \mathbb{N}$
$b \in \mathbb{N}$
$c \in \mathbb{N}$
$a + b + c = n$
$a = 0 \ \lor \ c = 0$
$a + b < d$
$c = 0$
$\Rightarrow$
$a + 1 \in \mathbb{N}$
$a + 1 + b + c = n + 1$
$a + 1 = 0 \ \lor \ c = 0$

$d \in \mathbb{N}$
$0 < d$
$n \in \mathbb{N}$
$n \leq d$
$a \in \mathbb{N}$
$b \in \mathbb{N}$
$c \in \mathbb{N}$
$a + b + c = n$
$a = 0 \ \lor \ c = 0$
$0 < c$
$\Rightarrow$
$c - 1 \in \mathbb{N}$
$a + b + c - 1 = n - 1$
$a = 0 \ \lor \ c - 1 = 0$

**Introducing New Events.** We now have to introduce some new events corresponding to cars entering and leaving the island. Next are the proposed new events. A can be seen , such events indeed refine skip as they only modify variables $a$, $b$ and $c$ in such a way that the abstract variable $n$ remains constant according to invariant inv1_4.

IL_in
**when**
$0 < a$
**then**
$a, b := a - 1, b + 1$
**end**

IL_out
**when**
$0 < b$
$a = 0$
**then**
$b, c := b - 1, c + 1$
**end**

**Proving that the New Events are Correct** We leave it as an exercise to the reader to state and prove that these events refine skip. We now have to prove that new events do not diverge. For this, we have to exhibit a variant and prove that is decreased by both new events. The proposed variant is the following:

variant_1 : $2 * a + b$

Applying rule WFD_REF leads to the following obvious statements to prove:

$$
\begin{array}{l}
d \in \mathbb{N} \\
0 < d \\
n \in \mathbb{N} \\
n \leq d \\
a \in \mathbb{N} \\
b \in \mathbb{N} \\
c \in \mathbb{N} \\
a + b + c = n \\
a = 0 \ \lor \ c = 0 \\
0 < a \\
\Rightarrow \\
2 * (a - 1) + b + 1 < 2 * a + b
\end{array}
$$

$$
\begin{array}{l}
d \in \mathbb{N} \\
0 < d \\
n \in \mathbb{N} \\
n \leq d \\
a \in \mathbb{N} \\
b \in \mathbb{N} \\
c \in \mathbb{N} \\
a + b + c = n \\
a = 0 \ \lor \ c = 0 \\
0 < b \\
a = 0 \\
\Rightarrow \\
2 * a + (b - 1) < 2 * a + b
\end{array}
$$

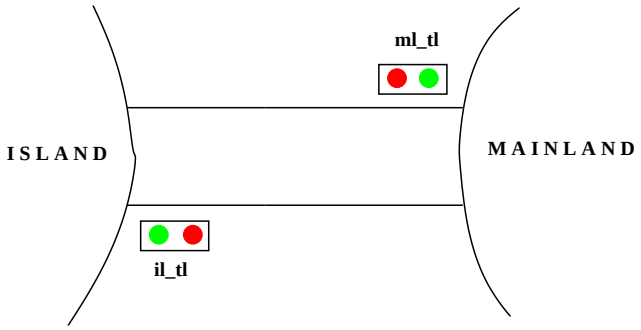
**No Deadlock.** Finally, we have to prove that the refined events and new events do not together deadlock. This is so because the abstraction did not deadlock. Rule W_DLK leads to the following to prove (it is simplified because we already proved that the abstraction did not deadlock):

$$
\begin{array}{l}
d \in \mathbb{N} \\
0 < d \\
n \in \mathbb{N} \\
n \leq d \\
a \in \mathbb{N} \\
b \in \mathbb{N} \\
c \in \mathbb{N} \\
a + b + c = n \\
a = 0 \ \lor \ c = 0 \\
\Rightarrow \\
(a + b < d \ \land \ c = 0) \ \lor \\
c > 0 \ \lor \\
a > 0 \ \lor \\
(b > 0 \ \land \ a = 0)
\end{array}
$$

### 2.4 Second Refinement: Introducing the Traffic Lights

At this point, the situation is a bit magic. It seems that car drivers can count cars and thus decide to enter into the bridge from the mainland (event ML_out) or from the island (event IL_out). In reality, as we know, the drivers follows the indication of the traffic lights, they clearly do not the count of the cars, which is impossible.

This refinement then consists in introducing first the two traffic lights, named ml_tl and il_tl, then the corresponding invariants, and finally some new events that are able to change the colors of the traffic lights. The next figure illustrates the new physical situation which can be observed:




**Refining the State** Two new variables are introduced, $ml\_tl$ (for mainland traffic light) and $il\_tl$ (for island traffic light). These variables take values 0 (for red) or 1 (for green): this is formalized in invariants named inv2_1 and inv2_2. Since drivers are allowed to pass when traffic lights are green (1), we better ensure by two invariants named inv2_3 and inv2_4 that when $ml\_tl$ is green then the guard of events ML_out holds, and that when $il\_tl$ is green then the guard of events IL_out holds. Here is the refined state:

**constants** : $d$

**variables** : $a, b, c,$
$ml\_tl, il\_tl$

inv2_1 : $ml\_tl \in \{0, 1\}$

inv2_2 : $il\_tl \in \{0, 1\}$

inv2_3 : $ml\_tl = 1 \Rightarrow a + b < d \wedge c = 0$

inv2_4 : $il\_tl = 1 \Rightarrow 0 < b \wedge a = 0$

**Refining Abstract Events** Events ML_out and IL_out are now refined by changing their guards to the test of the green value of the corresponding traffic lights. This is exactly what car drivers are doing. This is here where we implicitly assume that drivers obey the traffic lights, as indicated by requirements EQP-3. Note that events IL_in and ML_in are not modified in this refinement. Here is the new version of event ML_out presented together with its abstraction:

abstract_ML_out
**when**
$c = 0$
$a + b < d$
**then**
$a := a + 1$
**end**

concrete_ML_out
**when**
$ml\_tl = 1$
**then**
$a := a + 1$
**end**

And here is the new version of event IL_out presented together with its abstraction:

abstract_IL_out
**when**
$a = 0$
$0 < b$
**then**
$b, c := b - 1, c + 1$
**end**

concrete_IL_out
**when**
$il\_tl = 1$
**then**
$b, c := b - 1, c + 1$
**end**

**Introducing New Events** We have to introduce two new events to turn the value of the traffic lights color to green when they are red and when the conditions are appropriate. The appropriate conditions, once again, are exactly the guards of the abstract events ML_out and IL_out. Here are the proposed new events:

ML_tl_green
**when**
$ml\_tl = 0$
$a + b < d$
$c = 0$
**then**
$ml\_tl := 1$
**end**

IL_tl_green
**when**
$il\_tl = 0$
$0 < b$
$a = 0$
**then**
$il\_tl := 1$
**end**

**Proving that the Events are Correct.** Proving that the concrete events correctly refine their abstraction is easily done and left to the reader. But we have some problems with proving that they maintain the new invariants. For example, here is the statement to be proved (after some simplification) concerning the preservation of invariant inv2_4 by event ML_out:

$$
\begin{array}{l}
a \in \mathbb{N} \\
il\_tl = 1 \;\Rightarrow\; a = 0 \;\wedge\; 0 < b \\
ml\_tl = 1 \\
\Rightarrow \\
il\_tl = 1 \;\Rightarrow\; \underline{a + 1} = 0 \;\wedge\; 0 < b
\end{array}
$$

This statement cannot be proved when $il\_tl$ is green (1). Likewise the following statement concerning the preservation of invariant inv2_3 by event IL_out cannot be proved when $ml\_tl$ is green (1):

$$
\begin{array}{l}
c \in \mathbb{N} \\
ml\_tl = 1 \;\Rightarrow\; c = 0 \;\wedge\; a + b < d \\
il\_tl = 1 \\
\Rightarrow \\
ml\_tl = 1 \;\Rightarrow\; \underline{c + 1} = 0 \;\wedge\; a + b - 1 < d
\end{array}
$$

What these failures show is that both lights cannot be green at the same time, on obvious fact, which we have forgotten to state. We thus now introduce it as an extra invariant:

inv2_5 : $\quad ml\_tl = 0 \ \lor \ il\_tl = 0$

But this new invariant has to be preserved and this is clearly not the case with the proposed new events ML_tl_green and IL_tl_green unless we correct them by turning to red the other traffic light, yielding:

ML_tl_green
**when**
$ml\_tl = 0$
$a + b < d$
$c = 0$
**then**
$ml\_tl := 1$
$\underline{il\_tl := 0}$
**end**

IL_tl_green
**when**
$il\_tl = 0$
$0 < b$
$a = 0$
**then**
$il\_tl := 1$
$\underline{ml\_tl := 0}$
**end**

When trying to prove the preservation of invariant inv2_3 by event ML_out, we are again in trouble. Here is the corresponding (simplified) statement to prove:

$$\begin{array}{l} ml\_tl = 1 \ \Rightarrow \ c = 0 \ \land \ a + b < d \\ ml\_tl = 1 \\ \Rightarrow \\ ml\_tl = 1 \ \Rightarrow \ c = 0 \ \land \ \underline{a + 1} + b < d \end{array}$$

As can be seen, this statement cannot be proved when $a + 1 + b$ is equal to $d$ unless ml_tl is set to red (0). In fact, when $a + 1 + b$ is equal to $d$, it means that the entering car is the last one allowed to enter at this stage because more cars would violate requirement FUN_3, which says that there are no more than $d$ cars in the island and bridge. This indicates that event ML_out has to be split into two events (both refining their abstraction however) as follows:

ML_out_1
**when**
$ml\_tl = 1$
$\underline{a + b + 1 \neq d}$
**then**
$a := a + 1$
**end**

ML_out_2
**when**
$ml\_tl = 1$
$\underline{a + b + 1 = d}$
**then**
$a := a + 1$
$\underline{ml\_tl := 0}$
**end**

Likewise, invariant inv2_4 cannot be maintained by event IL_out when $b$ is equal to 1. In this case the last car is leaving the island. As a consequence the island traffic light has to turn red. Here is the simplified statement which has to be proved:

$$\begin{array}{l} il\_tl = 1 \ \Rightarrow \ a = 0 \ \land \ 0 < b \\ il\_tl = 1 \\ \Rightarrow \\ il\_tl = 1 \ \Rightarrow \ a = 0 \ \land \ 0 < \underline{b - 1} \end{array}$$

As for event ML_out, we have to split event IL_out as follows:

IL_out_1
**when**
$il\_tl = 1$
$b \neq 1$
**then**
$b, c := b - 1, c + 1$
**end**

IL_out_2
**when**
$il\_tl = 1$
$b = 1$
**then**
$b, c := b - 1, c + 1$
$il\_tl := 0$
**end**

**No divergence of new events** We have now to prove that the new events cannot diverge for ever. For this, we must exhibit a certain variant that must be decreased by the new events. In fact, it turns out to be impossible. For instance, when $a$ and $c$ are both 0, meaning that there is no car on the bridge in either direction then the traffic lights could freely change color for ever as one can figure out by looking at the new events ML_tl_green and IL_tl_green:

ML_tl_green
**when**
$ml\_tl = 0$
$a + b < d$
$c = 0$
**then**
$ml\_tl := 1$
$il\_tl := 0$
**end**

IL_tl_green
**when**
$il\_tl = 0$
$0 < b$
$a = 0$
**then**
$il\_tl := 1$
$ml\_tl := 0$
**end**

What could then happen is that the light colors are changing so rapidly that the drivers can never pass. We have to make the color changing in a more disciplined way, that is only when some car has passed in the other direction. For this we introduce two more variables $ml\_pass$ and $il\_pass$. Each of them can take two values 0 and 1. When $ml\_pass$ is equal to 1 it means that one car at least has passed on the bridge going to the island since mainland traffic light last turned green, and similarly when $il\_pass$ is equal to 1. These variables are formalized in the following invariants:

inv2_6 : ml_pass $\in \{0, 1\}$

inv2_7 : il_pass $\in \{0, 1\}$

We must now modify events ML_out_1, ML_out_2, IL_out_1, and IL_out_2 to make $ml\_pass$ or $il\_pass$ to 1 since a car has passed in the proper direction.

ML_out_1
**when**
$ml\_tl = 1$
$a + b + 1 \neq d$
**then**
$a := a + 1$
$ml\_pass := 1$
**end**

ML_out_2
**when**
$ml\_tl = 1$
$a + b + 1 = d$
**then**
$a := a + 1$
$ml\_tl := 0$
$ml\_pass := 1$
**end**

IL_out_1
**when**
$il\_tl = 1$
$b \neq 1$
**then**
$b := b - 1$
$c := c + 1$
$il\_pass := 1$
**end**

IL_out_2
**when**
$il\_tl = 1$
$b = 1$
**then**
$b := b - 1$
$c := c + 1$
$il\_tl := 0$
$il\_pass := 1$
**end**

But we must also modify event ML_tl_green and IL_tl_green to reset $ml\_pass$ and $il\_pass$ and also add in their guards the conditions $il\_pass = 1$ and $ml\_pass = 1$ respectively in order to be sure that indeed a car has passed in the other direction. This yields the following:

ML_tl_green
**when**
$ml\_tl = 0$
$a + b < d$
$c = 0$
$il\_pass = 1$
**then**
$ml\_tl := 1$
$il\_tl := 0$
$ml\_pass := 0$
**end**

IL_tl_green
**when**
$il\_tl = 0$
$0 < b$
$a = 0$
$ml\_pass = 1$
**then**
$il\_tl := 1$
$ml\_tl := 0$
$il\_pass := 0$
**end**

Having done all that, we can now state what is to be proved in order to guarantee that there is no divergence of the new events. The variant we can exhibit is the following:

variant_2 : $ml\_pass + il\_pass$

And the statements to be proved are the following:

$ml\_tl = 0$
$a + b < d$
$c = 0$
$il\_pass = 1$
$\Rightarrow$
$il\_pass < ml\_pass + il\_pass$

$il\_tl = 0$
$b > 0$
$a = 0$
$ml\_pass = 1$
$\Rightarrow$
$ml\_pass < ml\_pass + il\_pass$

At this point we figure out that it cannot be proved unless $ml\_pass = 1$ in the first case and $il\_pass = 1$ in the second one. We have two solutions: either to strengthen the guards of events ML_tl_green and IL_tl_green (adding to them the extra guards $ml\_pass = 1$ and $il\_pass = 1$ respectively) or adding some extra invariants. The first solution seems to be the more economical one, yielding:

ML_tl_green
**when**
$ml\_tl = 0$
$a + b < d$
$c = 0$
$il\_pass = 1$
$\underline{ml\_pass = 1}$
**then**
$ml\_tl := 1$
$il\_tl := 0$
$ml\_pass := 0$
**end**

IL_tl_green
**when**
$il\_tl = 0$
$0 < b$
$a = 0$
$ml\_pass = 1$
$\underline{il\_pass = 1}$
**then**
$il\_tl := 1$
$ml\_tl := 0$
$il\_pass := 0$
**end**

**No Deadlock** It remains now to prove that we have no deadlock. The statement to prove is then the disjunction of the various guards with some simplified assumption (we do not need all invariants):

$$
\begin{array}{l}
d \in \mathbb{N} \\
0 < d \\
ml\_tl \in \{0,1\} \\
il\_tl \in \{0,1\} \\
ml\_pass \in \{0,1\} \\
il\_pass \in \{0,1\} \\
a \in \mathbb{N} \\
b \in \mathbb{N} \\
c \in \mathbb{N} \\
\Rightarrow \\
(ml\_tl = 0 \ \wedge \ a + b < d \ \wedge \ c = 0 \ \wedge \ ml\_pass = 1 \ \wedge \ il\_pass = 1) \ \vee \\
(il\_tl = 0 \ \wedge \ a = 0 \ \wedge \ b > 0 \ \wedge \ ml\_pass = 1 \ \wedge \ il\_pass = 1) \ \vee \\
ml\_tl = 1 \ \vee \\
il\_tl = 1 \ \vee \\
a > 0 \ \vee \\
c > 0
\end{array}
$$

This statement can be simplified to the following:

$$
\begin{array}{l}
d \in \mathbb{N} \\
0 < d \\
b \in \mathbb{N} \\
ml\_tl = 0 \\
il\_tl = 0 \\
\Rightarrow \\
(b < d \ \wedge \ ml\_pass = 1 \ \wedge \ il\_pass = 1) \ \vee \\
(b > 0 \ \wedge \ ml\_pass = 1 \ \wedge \ il\_pass = 1)
\end{array}
$$

Unfortunately, this statement cannot be proved unless we add the following two extra invariants:

inv2_8 : $ml\_tl = 0 \Rightarrow ml\_pass = 1$

inv2_9 : $il\_tl = 0 \Rightarrow il\_pass = 1$

As a consequence, we can now remove the guards $ml\_pass = 1$ and $il\_pass = 1$ in events ML_tl_green and IL_tl_green since they are implied by the guards $ml\_tl = 0$ and $il\_tl = 0$. We figure out that our choice of strengthening the guard of these events was the wrong one. We should have added the two invariants. It remains for us to prove that the two new invariants inv2_8 and inv2_9 are indeed preserved by all events. We leave this as an exercise to the reader.

**Conclusion of the Second Refinement.** During this refinement, we have seen again how the proofs (or rather the failures of the proofs) have helped us correct our mistake or enlarge our model. If fact, we discovered 4 errors, we introduced several extra invariants, we corrected four events, and we introduced two more variables.

## 3 Third Refinement: Introducing Car Sensors

Let us consider the events ML_out_1, ML_out_2, IL_out_1, IL_out_2, ML_in, and IL_in:

ML_out_1
**when**
$ml\_tl = 1$
$a + b + 1 \neq d$
**then**
$a := a + 1$
$ml\_pass := 1$
**end**

ML_out_2
**when**
$ml\_tl = 1$
$a + b + 1 = d$
**then**
$a := a + 1$
$ml\_tl := 0$
$ml\_pass := 1$
**end**

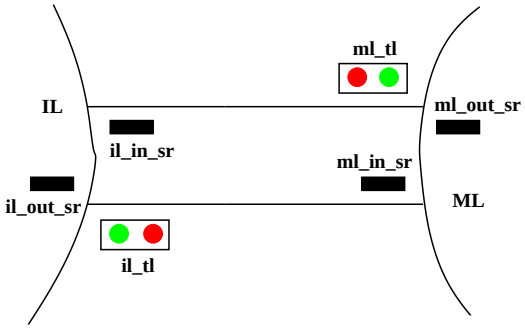
IL_out_1
**when**
$il\_tl = 1$
$b \neq 1$
**then**
$b := b - 1$
$c := c + 1$
$il\_pass := 1$
**end**

IL_out_2
**when**
$il\_tl = 1$
$b = 1$
**then**
$b := b - 1$
$c := c + 1$
$il\_tl := 0$
$il\_pass := 1$
**end**

ML_in
**when**
$0 < c$
**then**
$c := c - 1$
**end**

IL_in
**when**
$0 < a$
**then**
$a := a - 1$
$b := b + 1$
**end**

We can observe these events happening in the real world, but it is now important to have the controller being aware of them. For this, we introduce in this refinement some sensors able to communicate these transitions to the controllers, namely the passing of cars from the mainland to the bridge and vice-versa and from the bridge to the island and vice-versa. For doing this, we put four such sensors at both ends of the bridge as indicated on the following figure:




A sensor can be in one of two states: either on or off. It is on when a car is "on" it, off otherwise.

As a consequence, we shall enlarge our state with four variables corresponding to each sensor state: $ml\_out\_sr$, $ml\_in\_sr$, $il\_out\_sr$, and $il\_in\_sr$. They are defiined in invariants **inv3_1** to **inv3_4**:

| | |
|---|---|
| **constants** : | $d$ |
| **variables** : | $a, b, c,$ |
| | $ml\_tl, il\_tl$ |
| | $ml\_pass, il\_pass$ |
| | $ml\_out\_sr, ml\_in\_sr,$ |
| | $il\_out\_sr, il\_in\_sr$ |

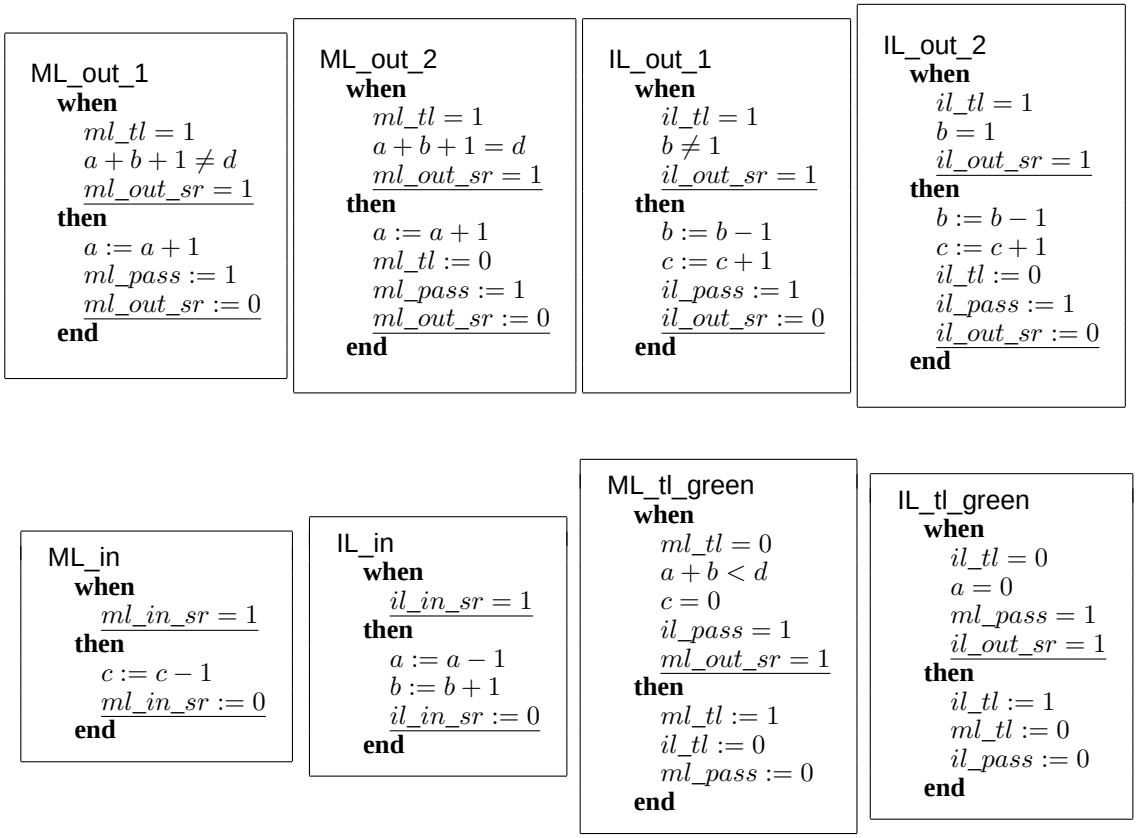
| | |
|---|---|
| **inv3_1** : | $ml\_out\_sr \in \{0, 1\}$ |
| **inv3_2** : | $ml\_in\_sr \in \{0, 1\}$ |
| **inv3_3** : | $il\_out\_sr \in \{0, 1\}$ |
| **inv3_4** : | $il\_in\_sr \in \{0, 1\}$ |

We also clearly have the new invariants stating that when the state $il\_in\_sr$ is 1, then $a$ is positive. In other words, there is at least one car on the bridge, namely the one that sits on the sensor $il\_in\_sr$. We have similar invariants for $il\_out\_sr$ and $ml\_in\_sr$, yielding:

| | |
|---|---|
| **inv3_5** : | $il\_in\_sr = 1 \Rightarrow a > 0$ |
| **inv3_6** : | $il\_out\_sr = 1 \Rightarrow b > 0$ |
| **inv3_7** : | $ml\_in\_sr = 1 \Rightarrow c > 0$ |

### 3.1 Refining abstract events.

It is now easy to proceed with the refinement of abstract events. This is done in a straightforward fashion as follows:



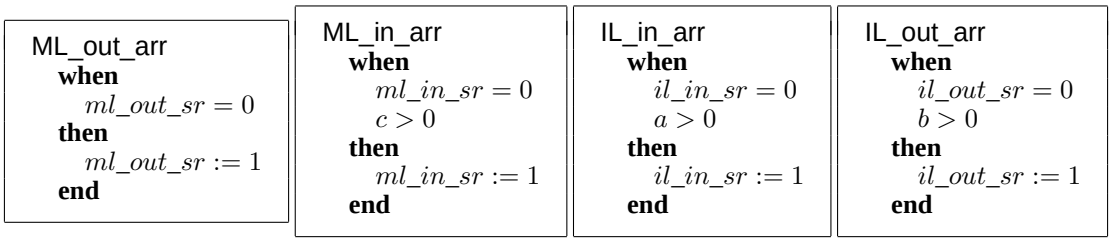

As can be seen, we have added the guards $ml\_out\_sr = 1$ and $il\_out\_sr = 1$ to the events ML_tl_green and IL_tl_green since there is no point in turning a traffic light to green when there is no car willing to pass. Note that this is a *design decision* as opposed to a requirement.

### 3.2 Correct Refinement

Proving that the previous events refine their abstract counterparts is left as an exercise to the reader.

### 3.3 Adding New Events

We now add four new events corresponding to cars arriving on the various sensors:



### 3.4 Refinement of new events.

The new events clearly refine skip as they do not work with old variables. We leave it to the reader to prove that they preserve the new invariants.

### 3.5 No divergence of new events.

We have to exhibit a variant which is decremented by all new events. Here it is:

**variant_3:** $4 - (ml\_out\_sr + ml\_in\_sr + il\_out\_sr + il\_in\_sr)$

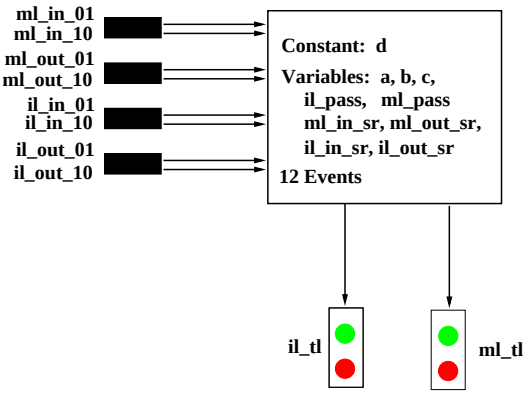
We leave it as an exercise to the reader to prove that this variant is decreased by the new events.

### 3.6 No Deadlock

Again, we leave it to the reader to prove that this third refinement does not deadlock.

### 3.7 Conclusion of the Third Refinement

The final structure of the system is shown on the following figure:



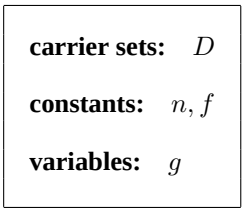

As can be seen, we now have one constant, nine variables, eight input wires, two output wires and twelve events.

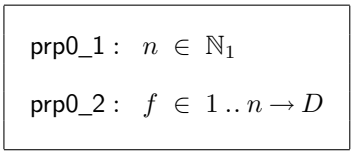
## 4 Example with Decomposition: The Two-phase Handshake Protocol

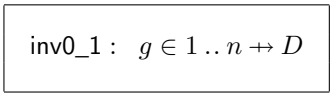
Our next example is a presentation of the very classical two-phase handshake protocol. This protocol is supposed to transfer a file from one agent, the Sender, to another one, the Receiver. These agents are supposed to reside on *different sites*, so that the transfer is not made by a simple copy of the file, it is rather realized gradually by two distinct programs exchanging various kinds of messages on a network. Such programs are working with different contexts and on different machines: the overall protocol is indeed a *distributed program*.

### 4.1 Protocol Initial Specification

What we are going to develop here is *not* directly the distributed program in question. We are rather going to construct a *model of its distributed execution*. In the context of this *model*, the file to transfer is formalized by means of a constant total function $f$ from the interval 1 to $n$ to some set $D$ (where $n$ is a constant positive natural number). The file $f$ is supposed to "reside" at the Sender's site. At the end of this protocol execution, we want the file $f$ to be copied without loss nor duplication on the Receiver's site.




The very global transfer action of the protocol can be abstracted by means of a *single* event called trm as follows:

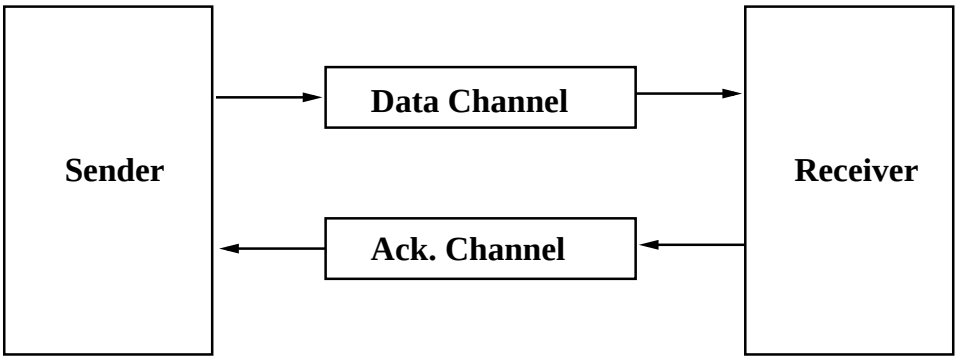

This event does not "exist" by itself. In other words, it is not part of the protocol: it is just a *time snapshot* that we would like hopefully to *observe*. In the "reality", the transfer of the file $f$ is not done in one shot, it is made gradually. But, at this very initial stage of our approach, we are not interested in this. In other words, *as an abstraction*, and regardless of what will happen in the details of the distributed execution of the protocol, its final action must result in the possibility to observe that the file $f$ has indeed been copied in the file $g$.

It should be noted that, at this point, we are not comitted with any particular protocol: this specification is thus, in a sense, the most general one corresponding to a given class, namely that of file transfers. Some more sophisticated specifications could have been proposed, in which the file might have only been partially transfered.

### 4.2 Protocol First Refinement

We are now going to *refine* the file transfer done in one shot by the previous *abstract* event trm acting "magically" on the Receiver's side. For this, we have a number of *concrete* events corresponding to the various *phases* of the protocol. They are aiming at transfering the file *piece by piece*. Of course, the abstract event trm should not disappear: it will have a concrete counterpart in which the same observation as in the abstraction must be possible.

These phases are informally behaving as follows: the Sender has a local counter, $s$, which records the "index" of the next datum to be sent to the Receiver (initially, $s$ is set to 1). When a transmission does occur, the data item $d$, which is equal to $f(s)$, is sent to the Receiver, the counter $s$ is incremented, and the new value of $s$ is also sent together with $d$ to the Receiver (event snd). Notice that the Sender does not immediately send the next item. It waits until it receives an *acknowledgement* from the Receiver. This acknowledgement, as we shall see, will also take the form of a counter.




The Receiver is also supposed to have its own local counter, $r$, initially set to 1. When receiving a pair "index-datum", the Receiver compares the received counter with $r$ and accepts the datum if the counter it receives is different from $r$ (event rcv). In this case, $r$ is incremented and then sent back as an acknowledgment. When the Sender receives a number $r$ which is equal to its own counter $s$, it consider that the acknowledgement is effective and proceeds with the next item, and so on.

The Sender and the Receiver are thus connected by means of two channels as indicated in the previous figure: the data channel and the acknowledgement channel. The state variables are declared as follows:

**carrier sets:** $D$

**constants:** $n, f$

**variables:** $g, s, r, d$

inv1_1 : $s \in 1 \mathinner{..} n+1$

inv1_2 : $r \in 1 \mathinner{..} n+1$

inv1_3 : $s \in r \mathinner{..} r+1$

inv1_4 : $g = 1 \mathinner{..} r-1 \lhd f$

Invariants inv1_1 and inv1_2 correspond to the declarations of $s$ and $r$. Invariant inv1_3 states that the counter $s$ is at most one more than the counter $r$. This invariant is the key of the protocol, since it will allow us to replace in a subsequent refinement $s$ and $r$ by their parities. Invariant inv1_4 states that the result file corresponds exactly to the $r-1$ first elements of the original file $f$. It remains now for us to formalize the channels. For the moment (in this refinement) the data channel contains the counter $s$ of the Sender and also some data item $d$. As the counter $s$ has already been formalized,we only have to define the invariants corresponding to $d$, formally:

inv1_5 : $d \in D$

inv1_6 : $s \neq r \;\Rightarrow\; d = f(s-1)$

Invariant inv1_6 states that the transmitted data $d$ is exactly the $(s-1)$th element of the input file $f$ when $s$ is different from $r$. The Acknowledgment channel just contains the counter $r$ of the Receiver. Next are the various events. They encode the informal behavior of the protocol as described above:

snd $\widehat{=}$
**when**
$s = r$
$s \neq n + 1$
**then**
$d := f(s)$
$s := s + 1$
**end**

rcv $\widehat{=}$
**when**
$s \neq r$
**then**
$g(r) := d$
$r := r + 1$
**end**

trm $\widehat{=}$
**when**
$s = r$
$s = n + 1$
**then**
skip
**end**

### 4.3 Protocol Second Refinement

In this refinement, we shall give the final implementation of the two-phase handshake protocol. The idea is to observe that it is not necessary to transmit the entire counters $s$ and $r$ through the data and acknowledgment channels. This is so for three reasons: (1) the only tests made on both sites are equality tests ($s = r$ or $s \neq r$), as can be seen in the events defined at the end of the previous section), (2) the only modifications of the counters are simple increments (again, this can be seen in the events defined in the previous section), and (3) the difference between $s$ and $r$ is at most 1 (look at invariant inv1_3). As a consequence, these equality tests can be performed on the *parities* of these pointers only. These are thus the quantities we are going to tranfer between the sites. Here are a few obvious definitions concerning the parities of natural numbers

prp2_1: $parity \in \mathbb{N} \to \{0, 1\}$

prp2_2: $parity(0) = 0$

prp2_3: $\forall x \cdot (x \in \mathbb{N} \Rightarrow parity(x + 1) = 1 - parity(x))$

It is then easy to prove the following result, which we are going to exploit. It says that the comparison of two natural numbers is identical to the comparison of their parities when the difference between these two numbers is at most one.

thm2_1: $\forall x, y \cdot (x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x \in y \mathinner{..} y + 1 \wedge parity(x) = parity(y) \Rightarrow x = y)$

We introduce two new variables $p$ and $q$ defined to be the parities of $s$ and $r$ respectively:

**carrier sets:** $D$

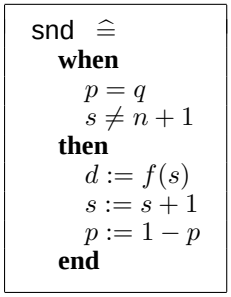
**constants:** $n, f, parity$

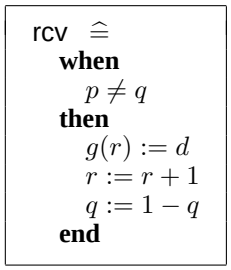
**variables:** $g, s, r, d, p, q$

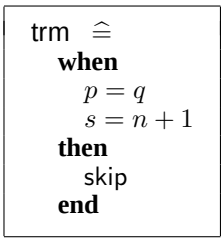
inv2_1 : $p = parity(s)$

inv2_2 : $q = parity(r)$

The refined events are as follows:




It can be seen that each counter $s$ or $r$ is now used on one site only. So the only data transmitted from one site to the other are $d$ and $p$ from the Sender to the Receiver and $q$ from the Receiver to the Sender. We can also observe that the event trm does not do anything any more (although it refines its abstraction). In fact, the transfer is now entirely made by the two other events. The validation of this little development requires 19 proofs among which 4 had to be done interactively (easily).

### 4.4 Decomposition

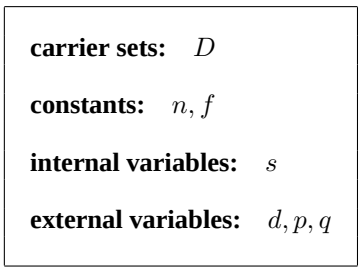
Before decomposing, let us rename the two events snd and rcv as follows:

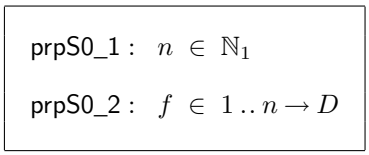
$$\text{snd} \quad \rightsquigarrow \quad \text{send\_data}$$

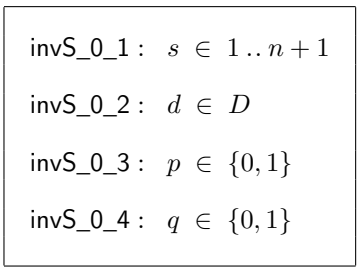
$$\text{rcv} \quad \rightsquigarrow \quad \text{send\_ack}$$

Our last refinement is now decomposed into two models called: Sender and Receiver.

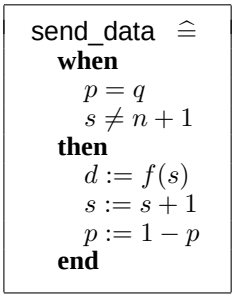
The Sender has internal variable $s$ and external variable $d$, $p$, and $q$.

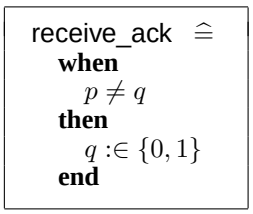


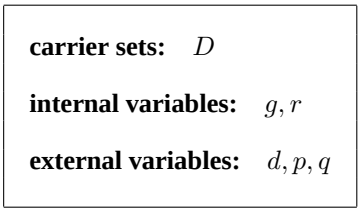



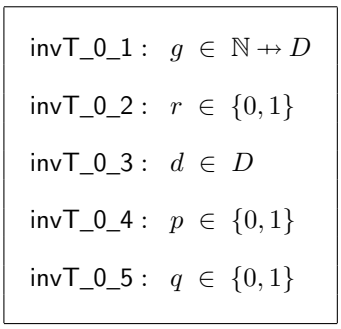

The internal event of Sender is send_data and its external event is called receive_ack.



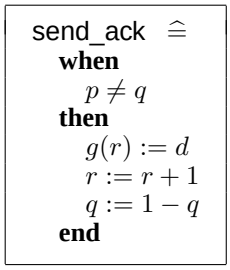



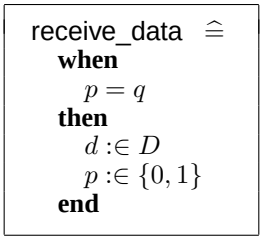
The Receiver has internal variable $g$ and $r$ and external variable $d$, $p$, and $q$.



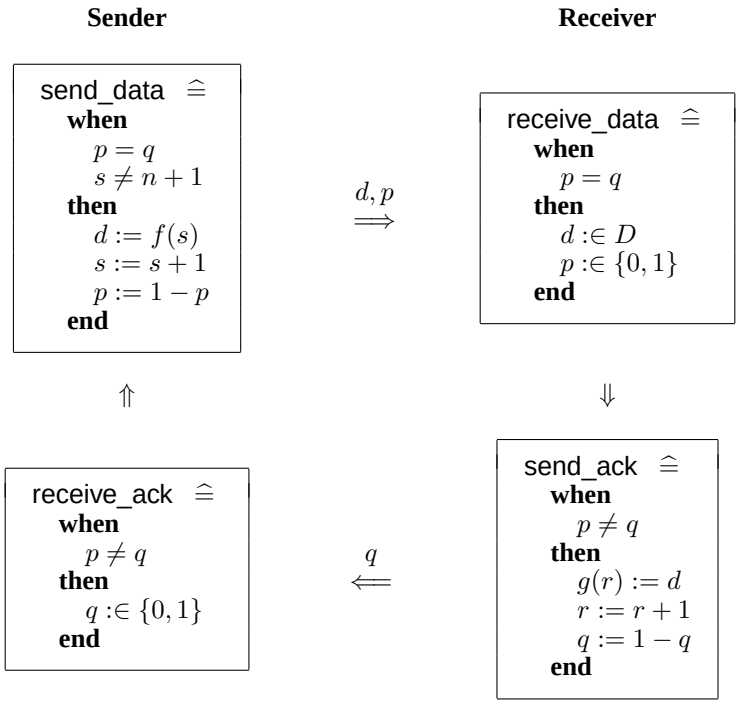




The internal event of Receiver is send_ack and its external event is called receive_data.




Putting the two models next to each other shows the communication between them.



In order to prove that the decomposition is correct, we have then to prove that the external event receive_ack of Sender is an abstraction of the internal event send_ack of Receiver. And symetrically, we have to prove that external event receive_data of Receiver is an abstraction of internal event send_data of Sender. Such proofs are easy.

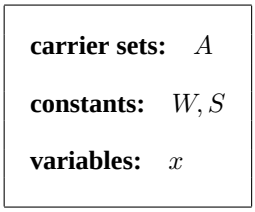
# 5 Example with Instantiation

In this example, we are going to perform three developments. The second will use an instantiation of the first one. Then we shall refine the result of this instantiation. And the third development will use an instantiation of the second one.

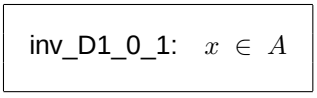
## 5.1 First Development

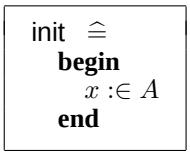
### Initial Model of First Development

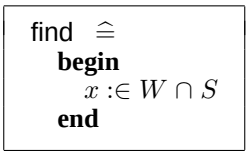
We are given a carrier set $A$ and two constant subsets $W$ and $S$ of it. Moreover, the intersection of $W$ and $S$ is supposed to be non-empty. We are asked to provide any element $x$ of $W \cap S$. Here is the formalization of this problem.





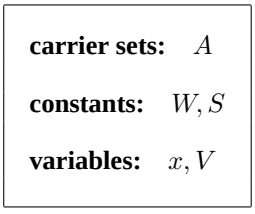


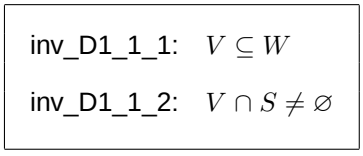




### First Refinement of First Development

In this refinement, we introduce a second variable $V$. It is a subset of $W$ and its intersection with $S$ is supposed to be non-empty. $V$ is equal to $W$ initially.



We introduce a new event remove. Given two distinct elements $a$ and $b$ of $V$ such that either $a$ is in $S$ or $b$ is not in $S$, then the action of remove consists in removing $b$ from $V$. Intuitively, if $a$ is in $S$, then removing $b$ form $V$ does not break the invariant $V \cap S \neq \varnothing$ ($a$ still is in $V \cap S$). And if $b$ is not in $S$, we can remove it from $V$ whatever the status of $A$. We leave the complete formal proof of this refinement as an exercise to the reader.

init $\widehat{=}$
**begin**
$x :\in A$
$V := W$
**end**

remove $\widehat{=}$
**any** $a, b$ **where**
$a \in V$
$b \in V$
$a \neq b$
$(a \in S \ \vee \ b \notin S)$
**then**
$V := V \setminus \{b\}$
**end**

find $\widehat{=}$
**any** $y$ **where**
$y \in V$
$V = \{y\}$
**then**
$x := y$
**end**

### 5.2 Second Development

#### Initial Model of Second Development

We are given a carrier set $B$ and two constants $r$ and $U$. The constant $r$ is supposed to be a binary relation built on $B$. This relation is a *well order*: it is a partial order (properties prp_D2_0_2 to prp_D2_0_4), such that any non-empty subset $V$ of $B$ has a smallest element with respect to $r$ (property prp_D2_0_5). The constant $U$ is a non-empty subset of $B$. We are asked to find the smallest element $x$ of $U$.

**carrier sets:** $B$
**constants:** $r, U$
**variables:** $x$

prp_D2_0_1: $r \in B \leftrightarrow B$
prp_D2_0_2: $\mathrm{id}(B) \subseteq r$
prp_D2_0_3: $r \cap r^{-1} \subseteq \mathrm{id}(B)$
prp_D2_0_4: $r ; r \subseteq r$
prp_D2_0_5: $\forall V \cdot \begin{pmatrix} V \subseteq B \\ V \neq \varnothing \\ \Rightarrow \\ \exists y \cdot ( y \in V \ \wedge \ V \subseteq r[\{y\}] ) \end{pmatrix}$
prp_D2_0_6: $U \subseteq B$
prp_D2_0_7: $U \neq \varnothing$
inv_D2_0_1: $x \in B$

init $\widehat{=}$
**begin**
$x :\in B$
**end**

find $\widehat{=}$
**begin**
$x :| ( x \in U \ \wedge \ U \subseteq r[\{x\}] )$
**end**

**First Refinement of Second Development**

We instantiate the first development by giving values to the parameters $A$, $W$, and $S$ in it. These values are defined as follows in terms of the carrier set $B$ and constants $r$ and $U$ of the second development:

$$A = B$$
$$W = U$$
$$S = \{ z \mid z \in B \wedge U \subseteq r[\{z\}] \}$$

And we now have to prove the three properties prp_D1_0_1, prp_D1_0_2, prp_D1_0_3 after instantiating them as above, namely

$$U \subseteq B$$
$$\{ z \mid z \in B \wedge U \subseteq r[\{z\}] \} \subseteq B$$
$$U \cap \{ z \mid z \in B \wedge U \subseteq r[\{z\}] \} \neq \varnothing$$

The first one is exactly prp_D2_0_6, the second is obvious, and the third is a direct consequence of prp_D2_0_5 by instantiating $V$ with $U$. We now have to prove that event find in the second development is refined by instantiated event find in the first development. This yields the following which trivially holds:

$$x \in U \cap \{ z \mid z \in B \wedge U \subseteq r[\{z\}] \} \Rightarrow x \in U \wedge U \subseteq r[\{x\}]$$

As a consequence, we get *for free* the instantiated refinement of the first development:

**carrier sets:** $B$

**constants:** $r, U$

**variables:** $x, V$

inv_D2_1_1: $V \subseteq U$

inv_D2_1_2: $\{ z \mid z \in B \wedge U \subseteq r[\{z\}] \} \cap V \neq \varnothing$

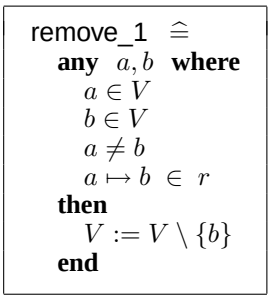
init $\widehat{=}$
**begin**
$x :\in B$
$V := U$
**end**

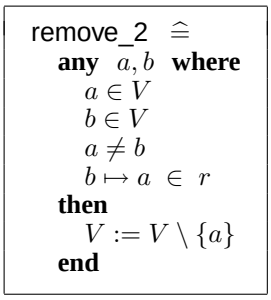
remove $\widehat{=}$
**any** $a, b$ **where**
$a \in V$
$b \in V$
$a \neq b$
$U \subseteq r[\{a\}] \vee U \not\subseteq r[\{b\}]$
**then**
$V := V \setminus \{b\}$
**end**

find $\widehat{=}$
**any** $y$ **where**
$y \in V$
$V = \{y\}$
**then**
$x := y$
**end**

**Second Refinement of Second Development**

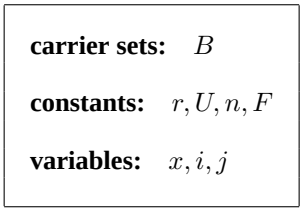
We now refine remove by strengthening its guard. We also split remove in two events as follows:

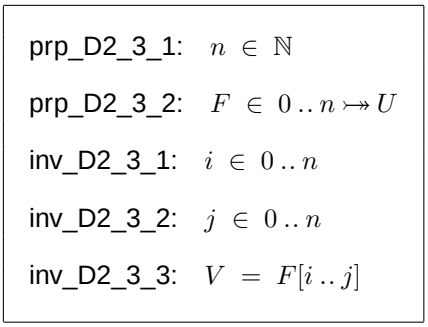


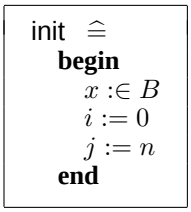



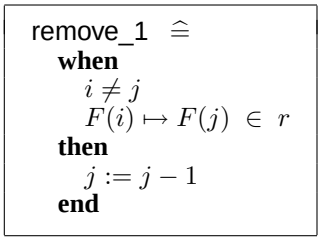

**Third Refinement of Second Development**

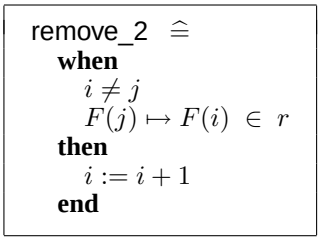
As an extra data-refinement, we introduce a constant $n$, which is a natural number, and a bijective function $F$ from the interval $0 \mathinner{..} n$ to the set $U$. We replace the variable $V$ by two natural numbers $i$, initialised to 0, and $j$, initialised to $n$, both belonging to the interval $0 \mathinner{..} n$. The gluing invariant inv_D2_3_3 stipulates that $V$ is the image of the interval $i \mathinner{..} j$ under $F$.

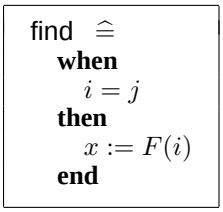


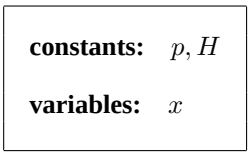



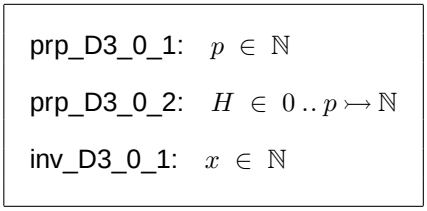



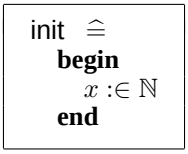

## 5.3 Third Development

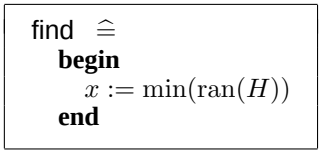
**Initial Model of Third Development**

In this development, we are given two constants: a natural number $p$ and an injective function $H$ from the interval $0 \mathinner{..} p$ to the set of natural numbers. And we are asked to find the minimum $x$ of the range of $H$.



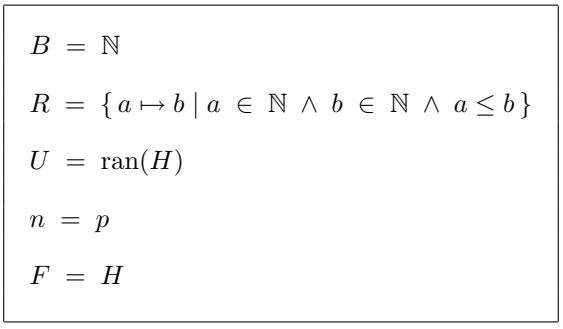




**First Refinement of Third Development**

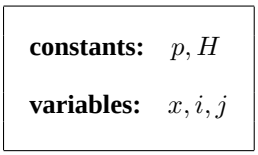
We instantiate the second development giving values to the parameters $B$, $r$, $U$,$n$, and $F$ in it. These values are defined as follows in terms of the constants $p$ and $H$ of the third development:

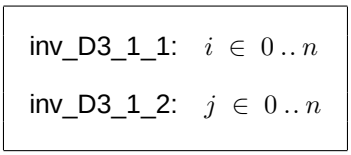



And we have to prove the seven properties prp_D2_0_1 to prp_D2_0_7 and the two proerties prp_D2_3_1 and prp_D2_3_2 after instantiating them. This can be done easily. We now have to prove that the instantiated event find of the initial model of second development is a refinement of event find of the initial model of the third development. This yields the following, which holds trivially:

$$x \in \mathrm{ran}(H) \wedge \forall y \cdot ( y \in \mathrm{ran}(H) \Rightarrow x \leq y ) \Rightarrow x = \min(\mathrm{ran}(H))$$

As a consequence, we obtain *for free* the instantiation of the third refinement of the second development, that is



```
init ≙
  begin
    x :∈ ℕ
    i := 0
    j := n
  end
```

```
remove_1 ≙
  when
    i ≠ j
    H(i) ≤ H(j)
  then
    j := j − 1
  end
```

```
remove_2 ≙
  when
    i ≠ j
    H(j) ≤ H(i)
  then
    i := i + 1
  end
```

```
find ≙
  when
    i = j
  then
    x := H(i)
  end
```

From these events, we can easily construct the following piece of code:

```
i, j := 0, n;
while i ≠ j do
  if H(i) ≤ H(j) then
    j := j − 1
  else
    i := i + 1
  end
end;
x := H(i)
```

# (V) Event-B: Mathematical Language

J.-R. Abrial

April 2005

Version 3

# Event-B: Mathematical Language

## 1 Introduction

This document contains the definition of the *Mathematical Language* we are going to use in Event-B. It is made of four sections introducing successively the Proposition Language (section 3), the Predicate Language (section 4), the Set-theoretic Language (section 5), and the Arithmetic Language (section 6). Each of these languages will be presented as an extension of the previous one. Before introducing these languages however, we shall give a brief summary of the Sequent Calculus (section 2).

## 2 Sequent Calculus

### 2.1 Definitions

(1) A *sequent* is a generic name for "something we want to prove". For the moment, this is just an informally defined notion, which we shall refine later. In what follows we shall use identifiers such as $S1$, $S2$, etc. to denote sequents. The important thing to note at this point is that we can associate a *proof* with a sequent. For the moment, we do not know what a proof is however. It will only be defined at the end of this section.

(2) An *inference rule* is a device used to construct proofs of sequents. It is made of two parts: the *antecedent* part and the *consequent* part. The antecedent denotes a finite set of sequents while the consequent denotes a single sequent. The inference rule, named **r** with antecendent $A$ and consequent $C$ is usually written as follows:

$$\mathbf{r} \quad \frac{A}{C}$$

It is to be read:

> Rule **r** yields a proof of sequent $C$ as soon as we have proofs of each sequent of $A$

Note that the antecedent $A$ might be empty. In this case, the inference rule, named say **x**, is written as follows:

$$\mathbf{x} \quad \frac{}{C}$$

And it is to be read:

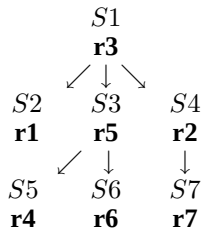
Rule **x** yields a proof of sequent $C$

(3) A *Theory* is a set of inference rules.

(4) It is now possible to give the definition of the *proof of a sequent* within a theory $\mathcal{T}$. It is simply a finite tree with certain contraints. The nodes of such a tree have two components: a sequent $s$ and a rule $r$ of the theory $\mathcal{T}$. Here are the constraints for each node of the form $s \mapsto r$: the consequent of the rule $r$ is $s$, and the sons of this node are nodes whose sequents are exactly all the sequents of the antecedent of rule $r$. As a consequence, the leaves of the tree contain rules with no antecedent. Moreover, the top node of the tree contains the sequent we want to prove. As an example, let $\mathcal{T}$ be the following theory:

**r1** $\frac{}{S2}$ **r2** $\frac{S7}{S4}$ **r3** $\frac{S2 \;\; S3 \;\; S4}{S1}$ **r4** $\frac{}{S5}$ **r5** $\frac{S5 \;\; S6}{S3}$ **r6** $\frac{}{S6}$ **r7** $\frac{}{S7}$

Here is a proof of the sequent $S1$:




As can be seen, the root of the tree has sequent $S1$ which we want to prove. And it is easy to check that each node, say node $S3 \mapsto$ **r5**, is indeed such that the consequent of its rule **r5** is the sequent $S3$. Moreover, we can check that the sequents of the son nodes of node $S3 \mapsto$ **r5**, namely, $S5$ and $S6$, are exactly the sequents forming the antecedents of rule **r5**. This tree can be given another more vertical form, which is the following, and which we shall adopt in the sequel:

| | |
|---|---|
| $S1$ | **r3** |
| $\quad S2$ | **r1** |
| $\quad S3$ | **r5** |
| $\quad\quad S5$ | **r4** |
| $\quad\quad S6$ | **r6** |
| $\quad S4$ | **r2** |
| $\quad\quad S7$ | **r7** |

### 2.2 Sequents for a Mathematical Language

We now refine our notion of sequent in order to define the way we shall make proofs with our Mathematical Language. Such a language contains constructs called *Predicates*. For the moment this is all that we know about our Mathematical Language. Within this framework, a sequent has the following form:

$\mathsf{H} \vdash G$

where $\mathsf{H}$ is a finite set of predicates called the *hypotheses*, and where G is a predicate called the *goal*. Note that the finite set of hypotheses $\mathsf{H}$ might be empty. This sequent is to be read as follows:

| Under the hypotheses of the set $\mathsf{H}$, prove the goal $G$ |
|---|

This is the sort of sequent we want to prove. It is also the sort of sequent we shall have in the theories associated with our Mathematical Language.

### 2.3 Initial Theory

We now have enough elements at our disposal to define the first rules of our proving theory. Note again that we still don't know what a predicate is. We just know that predicates are constructs we shall be able to define within our future Mathematical Language. We start with three basic rules which we first state informally. They are called **HYP**, **MON**, and **CUT**.

**HYP**: If the goal $P$ of a sequent belongs to the set of hypotheses $\mathsf{H}$ of this sequent, then it is proved.

**MON**: Once a sequent is proved, any sequent with the same goal and more hypotheses is also proved.

**CUT**: If you succeed in proving a predicate $P$ under a set of hypotheses $\mathsf{H}$, then $P$ can be added to the set of hypotheses $\mathsf{H}$ for proving a goal $Q$.

These rules can be encoded as follows:

$$\textbf{HYP}\ \frac{}{\mathsf{H},\ P\ \vdash\ P} \qquad \textbf{MON}\ \frac{\mathsf{H}\ \vdash\ Q}{\mathsf{H},\ P\ \vdash\ Q} \qquad \textbf{CUT}\ \frac{\mathsf{H}\ \vdash\ P \qquad \mathsf{H},\ P\ \vdash\ Q}{\mathsf{H}\ \vdash\ Q}$$

The previous theory can be given a more convenient tabular form, which we shall adopt in what follows. We name it $\mathcal{T}0$:

| | Antecedents | Consequent |
|---|---|---|
| **HYP** | | $\mathsf{H},\ P\ \vdash\ P$ |
| **MON** | $\mathsf{H}\ \vdash\ Q$ | $\mathsf{H},\ P\ \vdash\ Q$ |
| **CUT** | $\mathsf{H}\ \vdash\ P$<br>$\mathsf{H},\ P\ \vdash\ Q$ | $\mathsf{H}\ \vdash\ Q$ |

$\mathcal{T}0$

Note that in the previous rules, the letter $\mathsf{H}$, $P$ and $Q$ are, so-called, *meta-variables*. The letter $\mathsf{H}$ is a meta-variable standing for a finite set of predicates, whereas the letter $P$ and $Q$ are meta-variables standing for predicates. Clearly then, each of the previous "rules" stands for more than just one rule: it is better to call it a *rule schema* or a generic rule. This will always be the case in what follows.

# 3 The Proposition Language

In this section we present a first simple version of our Mathematical Language, it is called the Proposition Language. It will be later refined to more complete versions.

## 3.1 Syntax

Our first version is built around three constructs called *conjunction*, *implication*, and *negation*. Given two predicates $P$ and $Q$, we can construct their conjunction $P \land Q$ and their implication $P \Rightarrow Q$. And given a predicate $P$, we can construct its negation $\neg P$. This can be formalized by means of the following syntax:

$$\begin{array}{|l} predicate ::= \neg\, predicate \\ \qquad\qquad\quad predicate \land predicate \\ \qquad\qquad\quad predicate \Rightarrow predicate \end{array} \qquad \mathcal{SY}1$$

This syntax is clearly ambiguous, but we do not care about it at this stage. This will be studied and solved in another companion document named *Event-B: Syntax of the Mathematical Language*. In what follows, in order to avoid ambiguities, we shall provide as many pairs of parentheses as needed.

## 3.2 Enlarging the Initial Theory

The initial theory $\mathcal{T}0$ given in section 2.3 is enlarged with the following inference rules forming the Theory $\mathcal{T}1$:

| | Antecedents | Consequent |
|---|---|---|
| **R1** | $\mathsf{H} \vdash P$<br>$\mathsf{H} \vdash Q$ | $\mathsf{H} \vdash P \land Q$ |
| **R2** | $\mathsf{H} \vdash P \land Q$ | $\mathsf{H} \vdash P$ |
| **R3** | $\mathsf{H} \vdash P \land Q$ | $\mathsf{H} \vdash Q$ |
| **R4** | $\mathsf{H}, P \vdash Q$ | $\mathsf{H} \vdash P \Rightarrow Q$ |
| **R5** | $\mathsf{H} \vdash P \Rightarrow Q$ | $\mathsf{H}, P \vdash Q$ |
| **R6** | $\mathsf{H}, \neg Q \vdash P$<br>$\mathsf{H}, \neg Q \vdash \neg P$ | $\mathsf{H} \vdash Q$ |
| **R7** | $\mathsf{H}, Q \vdash P$<br>$\mathsf{H}, Q \vdash \neg P$ | $\mathsf{H} \vdash \neg Q$ |

$\mathcal{T}1$

As can be seen, rules **R1**, **R2**, and **R3** are used to eliminate or introduce the $\wedge$ operator. Rules **R4** and **R5** are used to eliminate or introduce the $\Rightarrow$ operator. And rules **R6** and **R7** are used to do proofs by contradiction.

### 3.3 Replacing the Previous Theory

The previous Theory $\mathcal{T}1$ is very natural and clearly in full accordance with our intuitive understanding of conjunction, implication and negation. But it suffers a very important drawback: it is not very convenient to use it to construct practical proofs because it offers too many possibilities.

We shall then propose another Theory called $\mathcal{S}1$. Theory $\mathcal{S}1$ can be constructed systematically from Theory $\mathcal{T}1$ but we shall not do this construction here. Theory $\mathcal{S}1$ is certainly far less natural than Theory $\mathcal{T}1$, but it offers the great advantage over Theory $\mathcal{T}1$ to be almost deterministic. Almost only, but, at the end of this section, we shall indicate a certain way of using it which makes it completely deterministic: it means that at each step of the proof tree construction we shall have only one possibility of choosing an applicable inference rule or no possibility at all in case of failure.

In fact, Theory $\mathcal{S}1$ defines a, so-called, *proof procedure* for the simple Proposition Language so far defined. It can be easily mechanized. But before doing this, we have to extend our proposition Language with one predicate: $\perp$. This predicate is just used in this theory. In other words, users of the Mathematical Language are not allowed to use it. Here is Theory $\mathcal{S}1$:

| | Antecedents | Consequent |
|---|---|---|
| **INI** | $\mathsf{H} \vdash \neg R \Rightarrow \perp$ | $\mathsf{H} \vdash R$ |
| **AXM** | | $\mathsf{H},\ P,\ \neg P \vdash R$ |
| **AND1** | $\mathsf{H} \vdash \neg Q \Rightarrow R$<br>$\mathsf{H} \vdash \neg P \Rightarrow R$ | $\mathsf{H} \vdash \neg (P \wedge Q) \Rightarrow R$ |
| **AND2** | $\mathsf{H} \vdash P \Rightarrow (Q \Rightarrow R)$ | $\mathsf{H} \vdash (P \wedge Q) \Rightarrow R$ |
| **IMP1** | $\mathsf{H} \vdash P \Rightarrow (\neg Q \Rightarrow R)$ | $\mathsf{H} \vdash \neg (P \Rightarrow Q) \Rightarrow R$ |
| **IMP2** | $\mathsf{H} \vdash Q \Rightarrow R$<br>$\mathsf{H} \vdash \neg P \Rightarrow R$ | $\mathsf{H} \vdash (P \Rightarrow Q) \Rightarrow R$ |
| **NEG** | $\mathsf{H} \vdash P \Rightarrow R$ | $\mathsf{H} \vdash \neg \neg P \Rightarrow R$ |
| **DED** | $\mathsf{H},\ P \vdash R$ | $\mathsf{H} \vdash P \Rightarrow R$ |

$\mathcal{S}1$

This theory has to be used with a, so-called, tactic telling us in which order the rules have to be applied. Here is the tactic, where RULES is one of **AXM**, **IMP1**, **IMP2**, **AND1**, **AND2**, **NEG**:

$$\boxed{\textbf{INI}\ ;\ (\text{RULES}^{\star}\ ;\ \textbf{DED})^{\star}}$$

This tactic has to be read as follows: First use rule **INI** once. Then start the following process: use any rule in RULES as long as it is possible, then use **DED** once, and finally re-start this process.

### 3.4 Example

Here is an example of using the previous proof procedure. It is used to prove the following sequent:

$$\vdash\ (A\ \Rightarrow\ B)\ \Rightarrow\ ((A\ \wedge\ C)\ \Rightarrow\ (B\ \wedge\ C))$$

**Proof**

$$\begin{array}{lr}
\vdash\ (A\ \Rightarrow\ B)\ \Rightarrow\ ((A\ \wedge\ C)\ \Rightarrow\ (B\ \wedge\ C)) & \textbf{INI}\\
\quad\vdash\ \neg((A\ \Rightarrow\ B)\ \Rightarrow\ ((A\ \wedge\ C)\ \Rightarrow\ (B\ \wedge\ C)))\Rightarrow\ \bot & \textbf{IMP1}\\
\quad\quad\vdash\ (A\ \Rightarrow\ B)\ \Rightarrow\ (\neg((A\ \wedge\ C)\ \Rightarrow\ (B\ \wedge\ C))\Rightarrow\ \bot) & \textbf{IMP2}\\
\quad\quad\quad\vdash\ B\ \Rightarrow\ (\neg((A\ \wedge\ C)\ \Rightarrow\ (B\ \wedge\ C))\Rightarrow\ \bot) & \textbf{DED}\\
\quad\quad\quad\quad B\ \vdash\ \neg((A\ \wedge\ C)\ \Rightarrow\ (B\ \wedge\ C))\Rightarrow\ \bot & \textbf{IMP1}\\
\quad\quad\quad\quad\quad B\ \vdash\ (A\ \wedge\ C)\ \Rightarrow\ (\neg(B\ \wedge\ C)\Rightarrow\ \bot) & \textbf{AND2}\\
\quad\quad\quad\quad\quad\quad B\ \vdash\ A\ \Rightarrow\ (C\ \Rightarrow\ (\neg(B\ \wedge\ C)\Rightarrow\ \bot)) & \textbf{DED}\\
\quad\quad\quad\quad\quad\quad\quad B, A\ \vdash\ C\ \Rightarrow\ (\neg(B\ \wedge\ C)\Rightarrow\ \bot) & \textbf{DED}\\
\quad\quad\quad\quad\quad\quad\quad\quad B, A, C\ \vdash\ \neg(B\ \wedge\ C)\Rightarrow\ \bot & \textbf{AND1}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad B, A, C\ \vdash\ \neg C\Rightarrow\ \bot & \textbf{DED}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad B, A, C, \neg C\ \vdash\ \bot & \textbf{AXM}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad B, A, C\ \vdash\ \neg B\Rightarrow\ \bot & \textbf{DED}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad B, A, C, \neg B\ \vdash\ \bot & \textbf{AXM}\\
\quad\quad\quad\vdash\ \neg A\ \Rightarrow\ (\neg((A\ \wedge\ C)\ \Rightarrow\ (B\ \wedge\ C))\Rightarrow\ \bot) & \textbf{DED}\\
\quad\quad\quad\quad\neg A\ \vdash\ \neg((A\ \wedge\ C)\ \Rightarrow\ (B\ \wedge\ C))\Rightarrow\ \bot & \textbf{IMP1}\\
\quad\quad\quad\quad\quad\neg A\ \vdash\ (A\ \wedge\ C)\ \Rightarrow\ (\neg(B\ \wedge\ C)\Rightarrow\ \bot) & \textbf{AND2}\\
\quad\quad\quad\quad\quad\quad\neg A\ \vdash\ A\ \Rightarrow\ (C\ \Rightarrow\ (\neg(B\ \wedge\ C)\Rightarrow\ \bot)) & \textbf{DED}\\
\quad\quad\quad\quad\quad\quad\quad\neg A, A\ \vdash\ C\ \Rightarrow\ (\neg(B\ \wedge\ C)\Rightarrow\ \bot) & \textbf{AXM}
\end{array}$$

As can be seen, this proof procedure must be mechanized: there is no point in doing such a proof manually.

### 3.5 Methodology

The method we are going to use to build our Mathematical Language will be very systematic. It consists of subsequently augmenting our syntax, and correlatively augmenting our available inference rules. At each step of the construction, we shall have a *natural* Theory $\mathcal{T}_i$ and a corresponding *practical* Theory $\mathcal{S}_j$ which is derived from $\mathcal{T}_i$.

We shall use two different approaches for extending our language. Either the extension corresponds to a simple facility. In other words, the new construct can entirely be defined in terms of existing ones. In that case, we shall only augment the current practical Theory $\mathcal{S}_j$. Or the new construct is definitely new and not related to any previous construct. In that case, we shall proceed differently. We first augment the current natural Theory $\mathcal{T}_i$ and then derive from it a corresponding augmentation of the current practical Theory $\mathcal{S}_j$

### 3.6 Extending the Proposition Language

The Proposition Language is now extended by adding two more constructs called *disjunction* and *equivalence*. Given two predicates $P$ and $Q$, we can construct their disjunction $P \vee Q$ and their equivalence $P \Leftrightarrow Q$. We also add one predicate: $\top$. As a consequence, our syntax is now the following, where we have underlined the new constructs:

$$
\begin{array}{rl}
predicate ::= & \bot \\
& \underline{\top} \\
& \neg\, predicate \\
& predicate \wedge predicate \\
& \underline{predicate \vee predicate} \\
& predicate \Rightarrow predicate \\
& \underline{predicate \Leftrightarrow predicate}
\end{array}
\qquad \mathcal{SY}2
$$

Such extensions are defined in terms of previous ones by mere rewriting rules:

| Predicate | Rewritten |
|---|---|
| $\top$ | $\neg\ \bot$ |
| $P\ \vee\ Q$ | $\neg P \Rightarrow Q$ |
| $P\ \Leftrightarrow\ Q$ | $(P\ \Rightarrow\ Q)\ \wedge\ (Q\ \Rightarrow\ P)$ |

Theory $\mathcal{S}1$ can be extended with the following rules which can be established easily after applying the previous rewriting rules:

| | Antecedents | Consequent |
|---|---|---|
| **OR1** | $\mathsf{H} \vdash \neg P \Rightarrow (\neg Q \Rightarrow R)$ | $\mathsf{H} \vdash \neg (P \vee Q) \Rightarrow R$ |
| **OR2** | $\mathsf{H} \vdash Q \Rightarrow R$<br>$\mathsf{H} \vdash P \Rightarrow R$ | $\mathsf{H} \vdash (P \vee Q) \Rightarrow R$ |
| **EQV1** | $\mathsf{H} \vdash P \Rightarrow (\neg Q \Rightarrow R)$<br>$\mathsf{H} \vdash \neg P \Rightarrow (Q \Rightarrow R)$ | $\mathsf{H} \vdash \neg (P \Leftrightarrow Q) \Rightarrow R$ |
| **EQV2** | $\mathsf{H} \vdash P \Rightarrow (Q \Rightarrow R)$<br>$\mathsf{H} \vdash \neg P \Rightarrow (\neg Q \Rightarrow R)$ | $\mathsf{H} \vdash (P \Leftrightarrow Q) \Rightarrow R$ |

$\mathcal{S}2$

# 4 The Predicate Language

In this section, we introduce the Predicate Language. The syntax is extended with a number of new kinds of predicates and also with the introduction of two new syntactic categories called *expression* and *variable*. A variable is a simple identifier. Given a list of variables $x$ made of distinct identifiers and a predicate $P$, the construct $\forall x \cdot P$ is called a *universally quantified predicate*. Given a list of variables $x$ made of distinct identifiers, a list of expressions $E$ of the same size as that of $x$, and a predicate $P$, the construct $[x := E]P$ is called a *substituted predicate*. An expression is either a variable, a *substituted expression* formed in exactly the same way as a substituted predicate, or else a *paired expression* $E \mapsto F$, where $E$ and $F$ are two expressions. Note that substituted predicates and expressions are given here in order to be able to define the inference rules of quantified predicates in section 4.5. In fact, such constructs will never be used by users of the Mathematical Language. Here is this new syntax:

$$
\begin{array}{lcl}
predicate & ::= & \bot \\
 & & \top \\
 & & \neg\, predicate \\
 & & predicate \ \wedge \ predicate \\
 & & predicate \ \vee \ predicate \\
 & & predicate \ \Rightarrow \ predicate \\
 & & predicate \ \Leftrightarrow \ predicate \\
 & & \forall var\_list \cdot predicate \\
 & & \underline{[var\_list := exp\_list]\, predicate} \\
 & & \\
\underline{expression} & ::= & variable \\
 & & [var\_list := exp\_list]\, expression \\
 & & expression \mapsto expression \\
 & & \\
\underline{variable} & ::= & identifier
\end{array}
\qquad \mathcal{SY}3
$$

Note that we have not defined any syntactic structures for the two syntactic categories $var\_list$ and $exp\_list$. They simply respectively denote finite sequences of variables and finite sequences of expressions.

## 4.1 Predicates and Expressions

It might be useful at this point to clearly identify the distinction between a predicate and an expression. A predicate $P$ is a piece of formal text which can be *proved* when embedded within a sequent as in:

$$\vdash \ P$$

A predicate does not denote anything. This is not the case of an expression which always denotes an *object*. An expression cannot be "proved". Hence predicate and expression are incompatible. Note that for the moment the possible expressions we can define are quite limited. This will be considerably extended in the Set-theoretic Language defined in section 5.

## 4.2 Substitutions and Quantified Predicates

A construct such as $x := E$, embedded into the predicate $[x := E]P$, where $x$ is a list of variables, $E$ is a list of expressions of the same size as $x$, and $P$ is a predicate, is called a *substitution*. The construct

$[x := E]P$ is a transformation of the predicate $P$ obtained by replacing in $P$ the *free occurrences* of the variables of the list $x$ by the corresponding expression of the list $E$. In section 4.3, we shall formally define what we mean by a free occurrence of a variable in a predicate or expression. And in section 4.4 we define rules to be applied in order to systematically perform such a transformation. Similar substitutions can be used in an expression.

A predicate such as $\forall x \cdot P$, where $x$ is a list of variables made of distinct identifiers and $P$ is a predicate, is called a *universally quantified predicate*. The predicate $P$ is the *scope* of the variables in $x$ in this quantified predicate. The variables of $x$ are said to be *bound* in $P$. Other variables having occurrences in $P$ which are not bound are said to be *free*. Informally speaking for the moment, saying that such a predicate is proved means that all predicates of the form $[x := E]P$ are then also proved. This will be formalized by two inference rules in section 4.5.

### 4.3 Free and Bound Variable Occurrences

The non-freeness of a list of variables in a formula can be calculated by means of a number of rules. These rules are defined on the structure of our Predicate Language. More precisely, the rules give meaning to the condition $l$ $\underline{\text{nfin}}$ $K$, where $l$ is a list of variables and $K$ is a predicate or an expression. A construct such as $l$ $\underline{\text{nfin}}$ $K$ is to be read "variables of the list $l$ are not free (bound) in $K$".

In the following table, $x$ and $y$ are meta-variables standing for a variable, $l$ is a meta-variable standing for a list of variables, $P$ and $Q$ are meta-variables standing for a predicate, $E$ and $F$ are meta-variables standing for an expression, L is a meta-variable standing for a list of expressions, and $K$ is a meta-variable standing for a predicate or an expression.

| | Non-freeness | Condition |
|---|---|---|
| **NF1** | $x$ $\underline{\text{nfin}}$ $y$ | $x$ and $y$ are distinct identifiers |
| **NF2** | $(l, y)$ $\underline{\text{nfin}}$ $P$ | $l$ $\underline{\text{nfin}}$ $P$ and $y$ $\underline{\text{nfin}}$ $P$ |
| **NF3** | $(l, y)$ $\underline{\text{nfin}}$ $E$ | $l$ $\underline{\text{nfin}}$ $E$ and $y$ $\underline{\text{nfin}}$ $E$ |
| **NF4** | $x$ $\underline{\text{nfin}}$ $(P \wedge Q)$ | $x$ $\underline{\text{nfin}}$ $P$ and $x$ $\underline{\text{nfin}}$ $Q$ |
| **NF5** | $x$ $\underline{\text{nfin}}$ $(P \Rightarrow Q)$ | $x$ $\underline{\text{nfin}}$ $P$ and $x$ $\underline{\text{nfin}}$ $Q$ |
| **NF6** | $x$ $\underline{\text{nfin}}$ $\neg P$ | $x$ $\underline{\text{nfin}}$ $P$ |
| **NF7** | $x$ $\underline{\text{nfin}}$ $\forall x \cdot P$ | |
| **NF8** | $x$ $\underline{\text{nfin}}$ $\forall y \cdot P$ | $x$ $\underline{\text{nfin}}$ $y$ and $x$ $\underline{\text{nfin}}$ $P$ |

$\mathcal{NF}1$

| | Non-freeness | Condition |
|---|---|---|
| **NF9** | $x$ nfin $(\forall\, l, y \cdot P)$ | $x$ nfin $(\forall l \cdot \forall y \cdot P)$ |
| **NF10** | $x$ nfin $[x := E]K$ | $x$ nfin $E$ |
| **NF11** | $x$ nfin $[y := E]K$ | $x$ nfin $y$ and $x$ nfin $E$ and $x$ nfin $K$ |
| **NF12** | $x$ nfin $[l, x := L, E]K$ | $x$ nfin$[l := L]K$ and $x$ nfin $E$ |
| **NF13** | $x$ nfin $[l, y := L, E]K$ | $x$ nfin $y$ and $x$ nfin$[l := L]K$ and $x$ nfin $E$ |
| **NF4** | $x$ nfin $l, y$ | x nfin $l$ and $x$ nfin $y$ |
| **NF15** | $x$ nfin $L, E$ | x nfin $L$ and $x$ nfin $E$ |
| **NF16** | $x$ nfin $(E \mapsto F)$ | $x$ nfin $E$ and $x$ nfin $F$ |

$\mathcal{NF}1$ (cont'd)

### 4.4 Substitution Rules

Substituted predicates or expressions can be calculated by means of the following rules defined on the structure of the Predicate Language. In this table, we use the same meta-variable conventions as in the previous table.

| | Substitution | Definition |
|---|---|---|
| **SUB1** | $[x := E]\, x$ | $E$ |
| **SUB2** | $[x := E]\, y$ | $y$ if $x$ nfin $y$ |
| **SUB3** | $[x := E]\, (P \wedge Q)$ | $[x := E]\, P \ \wedge\ [x := E]\, Q$ |
| **SUB4** | $[x := E]\, (P \Rightarrow Q)$ | $[x := E]\, P \ \Rightarrow\ [x := E]\, Q$ |
| **SUB5** | $[x := E]\, \neg\, P$ | $\neg\, [x := E]\, P$ |

$\mathcal{SUB}1$

| | Substitution | Definition |
|---|---|---|
| **SUB6** | $[x := E] \forall x \cdot P$ | $\forall x \cdot P$ |
| **SUB7** | $[x := E] \forall y \cdot P$ | $\forall y \cdot [x := E] P$ if $y$ nfin $x$ and $y$ nfin $E$ |
| **SUB8** | $[x := E] \forall l, y \cdot P$ | $[x := E] \forall l \cdot \forall y \cdot P$ |
| **SUB9** | $[x := E] (F \mapsto G)$ | $[x := E] F \mapsto [x := E] G$ |
| **SUB10** | $[l, x := L, E] F$ | $[z := E] [l := L] [x := z] F$ if $z$ nfin $(l, x)$ and $z$ nfin $(L, E, F)$ |

$\mathcal{SUB}1$ (cont'd)

The application of these rules makes it possible to completely eliminate substitutions. Note that it is always possible to transform a quantified predicate such as $\forall x \cdot P$ into the following equivalent one $\forall y \cdot [x := y]P$ provided $y$ is not free in $P$. This transformation is called a *change of variables*.

### 4.5 Universally Quantified Predicate Inference Rules

We now enlarge the initial theory $\mathcal{T}1$ of section 3.2 with the following specific rules for universally quantified predicates:

| | Antecedents | Consequent |
|---|---|---|
| **R8** | $x$ nfin $Q$ for each $Q$ in H<br>H $\vdash$ $P$ | H $\vdash$ $\forall x \cdot P$ |
| **R9** | H $\vdash$ $\forall x \cdot P$ | H $\vdash$ $[x := E] P$ |

$\mathcal{T}2$

### 4.6 Replacing the Previous Theory Extension

As for the case of the Proposition Language in section 3.3, we can replace the extension of previous section by this one which is more convenient to use for mechanizing the proof construction of our Predicate Language. Here is this extension of the practical Theory $\mathcal{S}2$:

| | Antecedents | Consequent |
|---|---|---|
| **ALL1** | if $x$ nfin $R$ and $x$ nfin H<br>H $\vdash \neg P \Rightarrow R$ | H $\vdash \neg(\forall x \cdot P) \Rightarrow R$ |
| **ALL2** | H $\vdash (\forall l, y \cdot P) \Rightarrow R$ | H $\vdash (\forall l \cdot \forall y \cdot P) \Rightarrow R$ |
| **INS** | H, $\forall x \cdot P \vdash [x := E]\, P \Rightarrow \bot$ | H, $\forall x \cdot P \vdash \bot$ |

$\mathcal{S}3$

This theory has to be used with the following tactic, where RULES is one of **AXM**, **IMP1**, **IMP2**, **AND1**, **AND2**, **NEG**, **OR1**, **OR2**, **ALL1**, and **ALL2**:

**INI** ; ((RULES$^\star$ ; **DED**)$^\star$;**INS**)$^\star$

### 4.7 Extending the Predicate Language: Existential Quantification

The Predicate Language is now extended by introducing *existential quantification* of predicates. Given a predicate $P$ and a list of variables $x$, made of distinct identifiers, we can construct the predicate $\exists\, x \cdot P$. The new syntax is thus now as follows:

$$
\begin{array}{lcl}
predicate & ::= & \bot \\
 & & \top \\
 & & \neg\, predicate \\
 & & predicate \wedge predicate \\
 & & predicate \vee predicate \\
 & & predicate \Rightarrow predicate \\
 & & predicate \Leftrightarrow predicate \\
 & & \forall var\_list \cdot predicate \\
 & & \exists var\_list \cdot predicate \\
 & & [var\_list := exp\_list]\, predicate \\
 & & \\
expression & ::= & variable \\
 & & [var\_list := exp\_list]\, expression \\
 & & expression \mapsto expression \\
 & & \\
variable & ::= & identifier
\end{array}
$$

$\mathcal{SY}4$

This extension is defined as follows by a rewriting rule in terms of the universally quantified predicate:

| Predicate | Rewritten |
|---|---|
| $\exists x \cdot P$ | $\neg \forall x \cdot \neg P$ |

The previous practical Theory $\mathcal{S}3$ can then be extended as follows after applying the previous rewriting rule:

| | Antecedents | Consequent |
|---|---|---|
| **XST1** | $\mathsf{H} \vdash (\forall x \cdot \neg P) \Rightarrow R$ | $\mathsf{H} \vdash \neg (\exists x \cdot P) \Rightarrow R$ |
| **XST2** | if $x$ nfin $R$ and $x$ nfin H<br>$\mathsf{H} \vdash P \Rightarrow R$ | $\mathsf{H} \vdash (\exists x \cdot P) \Rightarrow R$ |

$\mathcal{S}4$

### 4.8 Extending the Predicate Language: Equality

The Predicate Language is once again extended by adding a new predicate, the *equality predicate*. Given two expressions $E$ and $F$, we define the following construct $E = F$. Here is the extension of our syntax:

$$
\begin{array}{lcl}
predicate & ::= & \bot \\
 & & \top \\
 & & \neg\, predicate \\
 & & predicate \wedge predicate \\
 & & predicate \vee predicate \\
 & & predicate \Rightarrow predicate \\
 & & predicate \Leftrightarrow predicate \\
 & & \forall var\_list \cdot predicate \\
 & & \exists var\_list \cdot predicate \\
 & & [var\_list := exp\_list]\, predicate \\
 & & \underline{expression = expression} \\
 & & \\
expression & ::= & variable \\
 & & [var\_list := exp\_list]\, expression \\
 & & expression \mapsto expression \\
 & & \\
variable & ::= & identifier
\end{array}
$$

$\mathcal{SY}5$

We extend in a similar manner the rules of non-freeness as follows:

| | Non-freeness | Condition |
|---|---|---|
| **NF17** | $x$ nfin $(E = F)$ | $x$ nfin $E$ and $x$ nfin $F$ |

$\mathcal{NF}2$

We also extend the substitution rules as follows:

| | Substitution | Definition |
|---|---|---|
| **SUB11** | $[x := C\,](D = E)$ | $[x := C\,]D \;=\; [x := C\,]E$ |

$\mathcal{SUB}2$

The natural Theory $\mathcal{T}2$ is extended with two inference rules for proving equality predicates:

| | Antecedents | Consequent |
|---|---|---|
| **R10** | $\mathsf{H} \;\vdash\; E = F$<br>$\mathsf{H} \;\vdash\; [x := E]P$ | $\mathsf{H} \;\vdash\; [x := F]P$ |
| **R11** | | $\mathsf{H} \;\vdash\; E = E$ |

$\mathcal{T}3$

Finally, we extend accordingly our practical Theory $\mathcal{S}4$:

| | Antecedents | Consequent |
|---|---|---|
| **EQL1** | | $\mathsf{H} \;\vdash\; \neg\,(E = E) \Rightarrow R$ |
| **LBZ1** | $\mathsf{H},\, E = F,\, [x := E]P \;\vdash\; [x := F]P \Rightarrow\; \bot$ | $\mathsf{H},\, E = F,\, [x := E]P \;\vdash\; \bot$ |
| **LBZ2** | $\mathsf{H},\, E = F,\, [x := F]P \;\vdash\; [x := E]P \Rightarrow\; \bot$ | $\mathsf{H},\, E = F,\, [x := F]P \;\vdash\; \bot$ |

$\mathcal{S}5$

## 5 The Set-theoretic Language

Our next language, the Set-theoretic Language is now presented as an extension to the previous Predicate Language.

### 5.1 Syntax

We introduce another predicate the *membership predicate* and a new syntactic category: *set*. Given an expression $E$ and a set $s$, the construct $E \in s$ is a membership predicate. We also enlarge the expression syntactic category by adding that a set is an expression. Finally, we introduce the sets constructs. Given two sets $s$ and $t$, the construct $s \times t$ is a set called the *Cartesian product* of $s$ and $t$. Given a set $s$, the construct $\mathbb{P}(s)$ is a set called the *power set of* $s$. Finally, given a list of variables $x$ with distinct identifiers, a predicate $P$, and an expression $E$, the construct $\{x \cdot P \mid E\}$ is called a *set defined in comprehension*. Here is our new syntax:

$$
\begin{array}{lcl}
predicate & ::= & \bot \\
 & & \top \\
 & & \neg\, predicate \\
 & & predicate \ \wedge \ predicate \\
 & & predicate \ \vee \ predicate \\
 & & predicate \ \Rightarrow \ predicate \\
 & & predicate \ \Leftrightarrow \ predicate \\
 & & \forall\, var\_list \cdot predicate \\
 & & \exists\, var\_list \cdot predicate \\
 & & [var\_list := exp\_list]\, predicate \\
 & & expression \ = \ expression \\
 & & \underline{expression \ \in \ set} \\
 & & \\
expression & ::= & variable \\
 & & [var\_list := exp\_list]\, expression \\
 & & expression \mapsto expression \\
 & & \underline{set} \\
 & & \\
variable & ::= & identifier \\
 & & \\
\underline{set} & ::= & set \times set \\
 & & \mathbb{P}(set) \\
 & & \{\, var\_list \cdot predicate \mid expression \,\} \\
 & & variable
\end{array}
\qquad \mathcal{SY}6
$$

### 5.2 Non-freeness and Substitution Rules

We first enlarge the non-freeness set of rules by adding the following ones:

| | Non-freeness | Condition |
|---|---|---|
| **NF18** | $x$ nfin $(s \times t)$ | $x$ nfin $s$ and $x$ nfin $t$ |
| **NF19** | $x$ nfin $\mathbb{P}(s)$ | $x$ nfin $s$ |
| **NF20** | $x$ nfin $\{l \cdot P \mid E\}$ | $x$ nfin $(\forall l \cdot P \ \Rightarrow \ E = E)$ |

$\mathcal{NF}3$

Likewise, we add the following rules to the substitution ones:

| | Substitution | Definition |
|---|---|---|
| **SUB12** | $[\,x := E\,]\,(s \times t)$ | $[\,x := E\,]s \;\times\; [\,x := E\,]t$ |
| **SUB13** | $[\,x := E\,]\;\mathbb{P}(s)$ | $\mathbb{P}([\,x := E\,]s)$ |
| **SUB14** | $[\,x := E\,]\,\{x \cdot P \mid F\}$ | $\{\,x \cdot P \mid F\,\}$ |
| **SUB15** | $[\,x := E\,]\,\{y \cdot P \mid F\}$ | $\{\,y \cdot [\,x := E\,]P \mid [\,x := E\,]F\,\}$ if $y$ nfin $x$ and $y$ nfin $E$ |
| **SUB16** | $[\,x := E\,]\,\{l, x \cdot P \mid F\}$ | $\{l, x \cdot P \mid F\}$ |
| **SUB17** | $[\,x := E\,]\,\{l, y \cdot P \mid F\}$ | $\{\,l, y \cdot [x := E]P \mid [\,x := E\,]\,F\,\}$ if $x$ nfin $y$ and $l$ nfin $x$ and $l, y$ nfin $E$ |

$\mathcal{SUB}3$ (cont'd)

### 5.3 Inference Rules

The inference rules for the set-theoretic Language are given under the form of equivalences to various set memberships. They are all defined in terms of rewriting rules. Note that the first of this rule defines equality for sets. It is called the extensionality axiom.

| Operator | Predicate | Rewritten |
|---|---|---|
| Set equality | $s = t$ | $s \in \mathbb{P}(t) \;\wedge\; t \in \mathbb{P}(s)$ |
| Cartesian product | $E \mapsto F \in s \times t$ | $E \in s \;\wedge\; F \in t$ |
| Power set | $E \in \mathbb{P}(s)$ | $\forall x \cdot (\,x \in E \;\Rightarrow\; x \in s\,)$ if $x$ nfin $E$ and $x$ nfin $s$ |
| Set comprehension | $E \in \{\,x \cdot P \mid F\,\}$ | $\exists x \cdot (\,P \;\wedge\; E = F\,)$ if $x$ nfin $E$ |

$\mathcal{SET}1$

As a special case, set comprehension can sometimes be written $\{\,F \mid P\,\}$, which can be read as follows: "the set of objects with shape $F$ when $P$ holds". However, as you can see, the list of variables $x$ has

now disappeared. In fact, these variables are then *implicitly determined* as being all the free variables in $F$. When we want that $x$ represent only *some*, but not all, of these free variables we cannot use this shorthand.

A more special case is one where the expression $F$ is exactly $x$, that is $\{ x \cdot P \mid x \}$. As a shorthand, this can be written $\{ x \mid P \}$, which is very common in informally written mathematics. But again, notice that, contrary to intuition, the list of variables $x$ has disappeared. Again, it will be determined as the free variables of expression $x$. And then $E \in \{ x \mid P \}$ becomes $[x := E]P$.

From now on, all extensions of the Set-theoretic Language will take the form of "simple facilities", as explained in section 3.5. And most of them are extensions of the *set* syntactic category. As a consequence, the new syntax will be presented differently. The new constructs will be presented under the form of rewriting rules. And since most of the new construct are sets, the rewriting rules will transform some set membership predicates into simpler ones.

### 5.4 Elementary Set Operators

In this section, we introduce the classical set operators: inclusion, union, intersection, difference, extension, and the empty set.

$$
\begin{array}{lll}
predicate & ::= & \ldots \\
 & & set \subseteq set \\
\ldots & & \\
set & ::= & \ldots \\
 & & set \cup set \\
 & & set \cap set \\
 & & set \setminus set \\
 & & \{exp\_list\} \\
 & & \varnothing
\end{array}
\qquad \mathcal{SY}7
$$

| Operator | Predicate | Rewritten |
|---|---|---|
| Inclusion | $S \subseteq T$ | $S \in \mathbb{P}(T)$ |
| Union | $E \in S \cup T$ | $E \in S \ \lor \ E \in T$ |
| Intersection | $E \in S \cap T$ | $E \in S \ \land \ E \in T$ |
| Difference | $E \in S \setminus T$ | $E \in S \ \land \ E \notin T$ |
| Set extension | $E \in \{a, \ldots, b\}$ | $E = a \ \lor \ \ldots \ \lor \ E = b$ |
| Empty set | $E \in \varnothing$ | $\bot$ |

$\mathcal{SET}2$

### 5.5 Generalization of Elementary Set Operators

The next series of operators consists in generalizing union and intersection to sets of sets. This takes the forms either of an operator acting on a set or of a quantifier.

$$
\begin{array}{|l|}
\hline
\ldots \\
\\
set \ ::= \ \ldots \\
\qquad \mathrm{union}(set) \\
\qquad \bigcup var\_list \cdot (predicate \mid set) \\
\qquad \mathrm{inter}(set) \\
\qquad \bigcap var\_list \cdot (predicate \mid set) \\
\hline
\end{array}
\qquad \mathcal{SY}8
$$

| Operator | Predicate | Rewritten |
|---|---|---|
| Generalized union | $E \in \mathsf{union}\,(S)$ | $\exists s \cdot (\, s \in S \ \wedge \ E \in s \,)$ if $s$ $\underline{\mathsf{nfin}}$ $S$ and $s$ $\underline{\mathsf{nfin}}$ $E$ |
| Quantified union | $E \in \bigcup x \cdot (\, P \mid T \,)$ | $\exists x \cdot (\, P \ \wedge \ E \in T \,)$ if $x$ $\underline{\mathsf{nfin}}$ $E$ |
| Generalized intersection | $E \in \mathrm{inter}\,(S)$ | $\forall s \cdot (\, s \in S \ \Rightarrow \ E \in s \,)$ if $s$ $\underline{\mathsf{nfin}}$ $S$ and $s$ $\underline{\mathsf{nfin}}$ $E$ |
| Quantified intersection | $E \in \bigcap x \cdot (\, P \mid T \,)$ | $\forall x \cdot (\, P \ \Rightarrow \ E \in T \,)$ if $x$ $\underline{\mathsf{nfin}}$ $E$ |

$\mathcal{SET}3$

The last two rewriting rules require that the set $\mathrm{inter}(S)$ and $\bigcap \ x \cdot (\, P \mid T \,)$ be *well defined*. This is defined in the following table:

| Set construction | Well-definedness condition |
|---|---|
| $\mathrm{inter}\,(S)$ | $S \neq \varnothing$ |
| $\bigcap \ x \cdot (\, P \mid T \,)$ | $\{x \cdot \ P \mid T \,\} \neq \varnothing$ |

$\mathcal{WF}1$

Note that well-definedness is studied in full details in the document entitled *Event-B: Syntax of Mathematical Language*.

### 5.6 Binary Relation Operators

We now define a first series of binary relation operators: the set of binary relations built on two sets, the domain and range of a binary relation, and then various sets of binary relations.

$$
\begin{array}{|l|}
\hline
\ldots \\
\\
set \ ::= \ \ldots \\
\qquad\quad set \leftrightarrow set \\
\qquad\quad \mathrm{dom}(set) \\
\qquad\quad \mathrm{ran}(set) \\
\qquad\quad set \nleftrightarrow set \\
\qquad\quad set \twoheadleftarrow set \\
\qquad\quad set \nleftrightarrow\!\!\!\!\twoheadrightarrow set \\
\hline
\end{array}
\qquad \mathcal{SY}9
$$

| Operator | Predicate | Rewritten |
|---|---|---|
| Set of binary relations | $r \in S \leftrightarrow T$ | $r \subseteq S \times T$ |
| Domain | $E \in \mathrm{dom}\,(r)$ | $\exists y \cdot (E \mapsto y \ \in \ r)$ if $y$ $\underline{\mathsf{nfin}}$ $E$ and $y$ $\underline{\mathsf{nfin}}$ $r$ |
| Range | $F \in \mathrm{ran}\,(r)$ | $\exists x \cdot (x \mapsto F \ \in \ r)$ if $x$ $\underline{\mathsf{nfin}}$ $F$ and $x$ $\underline{\mathsf{nfin}}$ $r$ |
| Set of total relations | $r \in S \nleftrightarrow T$ | $r \in S \leftrightarrow T \ \wedge \ \mathrm{dom}\,(r) = S$ |
| Set of surjective relations | $r \in S \twoheadleftarrow T$ | $r \in S \leftrightarrow T \ \wedge \ \mathrm{ran}\,(r) = T$ |
| Set of total and surjective relations | $r \in S \nleftrightarrow\!\!\!\!\twoheadrightarrow T$ | $r \in S \nleftrightarrow T \ \wedge \ r \in S \twoheadleftarrow T$ |

$\mathcal{SET}4$

The next series of binary relations operators define the converse of a relation, various relation restrictions and the image of a set under a relation.

$$
\begin{array}{l}
\ldots \\
\\
set \ ::= \ \ldots \\
\qquad\quad set^{-1} \\
\qquad\quad set \lhd set \\
\qquad\quad set \rhd set \\
\qquad\quad set \lhd\!\!\!\cdot \ set \\
\qquad\quad set \rhd\!\!\!\cdot \ set \\
\qquad\quad set[set]
\end{array}
\qquad \mathcal{SY}10
$$

| Operator | Predicate | Rewritten |
|---|---|---|
| Converse | $E \mapsto F \in r^{-1}$ | $F \mapsto E \in r$ |
| Domain restriction | $E \mapsto F \ \in \ S \lhd r$ | $E \in S \ \ \wedge \ \ E \mapsto F \in r$ |
| Range restriction | $E \mapsto F \ \in \ r \rhd T$ | $E \mapsto F \in r \ \ \wedge \ \ F \in T$ |
| Domain subtraction | $E \mapsto F \ \in \ S \lhd\!\!\!\cdot \ r$ | $E \notin S \ \ \wedge \ \ E \mapsto F \in r$ |
| Range subtraction | $E \mapsto F \ \in \ r \rhd\!\!\!\cdot \ T$ | $E \mapsto F \in r \ \ \wedge \ \ F \notin T$ |
| Image | $F \in r[w]$ | $\exists x \cdot ( \ x \in w \ \ \wedge \ \ x \mapsto F \in r \ )$ if $x$ <u>nfin</u> $F$ and $x$ <u>nfin</u> $r$ and $x$ <u>nfin</u> $w$ |

$\mathcal{SET}5$

Our next series of operators defines the composition of two binary relations, the overriding of a relation by another one, and the direct and parallel products of two relations.

$$
\begin{array}{l}
\ldots \\
\\
set \ ::= \ \ldots \\
\qquad\quad set \ ; \ set \\
\qquad\quad set \circ set \\
\qquad\quad set \lhd\!\!\!\cdot \ set \\
\qquad\quad set \otimes set \\
\qquad\quad set \parallel set
\end{array}
\qquad \mathcal{SY}11
$$

| Operator | Predicate | Rewritten |
|---|---|---|
| Forward composition | $E \mapsto F \in p \,;\, q$ | $\exists x \cdot (\, E \mapsto x \in p \;\; \land \;\; x \mapsto F \in q \,)$ if $x$ nfin $E$ and $x$ nfin $F$ and $x$ nfin $p$ and $x$ nfin $q$ |
| Backward composition | $E \mapsto F \in q \circ p$ | $E \mapsto F \in p \,;\, q$ |
| Overriding | $E \mapsto F \in p \lhd\!\!+ q$ | $E \mapsto F \in (\mathrm{dom}\,(q) \lhd p) \;\cup\; q$ |
| Direct product | $E \mapsto (F \mapsto G) \in p \otimes q$ | $E \mapsto F \in p \;\; \land \;\; E \mapsto G \in q$ |
| Parallel product | $(E \mapsto F) \mapsto (G \mapsto H) \in p \parallel q$ | $E \mapsto G \in p \;\; \land \;\; F \mapsto H \in q$ |

$\mathcal{SET}6$

### 5.7 Functional Operators

In this section we define various function operators: the sets of partial and total functions, partial and total injections, partial and total surjections, and bijections. We also introduce the two projection functions as well as the identity function.

$$
\begin{array}{l}
\ldots \\
\\
set \;::=\; \ldots \\
\qquad\quad set \pfun set \\
\qquad\quad set \fun set \\
\qquad\quad set \pinj set \\
\qquad\quad set \inj set \\
\qquad\quad set \psur set \\
\qquad\quad set \surj set \\
\qquad\quad set \bij set \\
\qquad\quad \mathrm{prj}_1(set) \\
\qquad\quad \mathrm{prj}_2(set) \\
\qquad\quad \mathrm{id}(set)
\end{array}
$$

$\mathcal{SY}12$

| Operator | Predicate | Rewritten |
|---|---|---|
| Set of partial functions | $f \in S \pfun T$ | $f \in S \rel T \ \land \ (f^{-1} ; f) = \mathrm{id}\,(\mathrm{ran}\,(f))$ |
| Set of total functions | $f \in S \fun T$ | $f \in S \pfun T \ \land \ S = \mathrm{dom}\,(f)$ |
| Set of partial injections | $f \in S \pinj T$ | $f \in S \pfun T \ \land \ f^{-1} \in T \pfun S$ |
| Set of total injections | $f \in S \inj T$ | $f \in S \fun T \ \land \ f^{-1} \in T \pfun S$ |
| Set of partial surjections | $f \in S \psur T$ | $f \in S \pfun T \ \land \ T = \mathrm{ran}\,(f)$ |
| Set of total surjections | $f \in S \surj T$ | $f \in S \fun T \ \land \ T = \mathrm{ran}\,(f)$ |
| Set of bijections | $f \in S \bij T$ | $f \in S \inj T \ \land \ f \in S \surj T$ |
| First projection | $(E \mapsto F) \mapsto G \in \mathsf{prj_1}(r)$ | $E \mapsto F \in r \ \land \ G = E$ |
| Second projection | $(E \mapsto F) \mapsto G \in \mathsf{prj_2}(r)$ | $E \mapsto F \in r \ \land \ G = F$ |
| Identity | $E \mapsto F \in \mathrm{id}\,(S)$ | $E \in S \ \land \ F = E$ |

$\mathcal{SET}7$

### 5.8 Lambda Abstraction and Function Invocation

We now define *lambda abstraction*, which is a way to construct functions, and also function invocation, which is a way to call functions. But first we have to define the notion of *pattern of variables*. A pattern of variables is either an identifier or a pair made of two patterns of variables. Moreover, all variables composing the pattern must be distinct. For example, here are three patterns of variables:

$$\mathsf{abc}$$

$$\mathsf{abc} \mapsto \mathsf{def}$$

$$\mathsf{abc} \mapsto (\mathsf{def} \mapsto \mathsf{ghi})$$

Given a pattern of variables $x$, a predicate $P$, and an expression $E$, the construct $\lambda\, x \cdot (P \mid E)$ is a lambda abstraction, which is a function. Given a function $f$ and an expression $E$, the construct $f(E)$ is an expression denoting a function invocation. Here is our new syntax

$$
\begin{array}{lll}
\ldots & & \\
expression & ::= & \ldots \\
 & & set(expression) \\
set & ::= & \ldots \\
 & & \lambda\, pattern \cdot predicate \,|\, expression \\
pattern & ::= & variable \\
 & & pattern \mapsto pattern
\end{array}
$$

$\mathcal{SY}13$

In the followig table, $l$ stands for the list of variables in the pattern $L$.

| Operator | Predicate | Rewritten |
|---|---|---|
| Lambda abstraction | $F \mapsto G \ \in\ \lambda L \cdot P \,|\, E$ | $F \mapsto G \ \in\ \{l \cdot P \,|\, L \mapsto E\}$ |
| Function invocation | $F = f(E)$ | $E \mapsto F \ \in\ f$ |

$\mathcal{SET}8$

The function invocation construct $f(E)$ requires a well-formedness condition, which is the following:

| Expression | Well-formedness condition |
|---|---|
| $f(E)$ | $E \ \in\ \mathrm{dom}(f)$ |

$\mathcal{WF}2$

# 6 Arithmetic Language

## 6.1 Syntax

We add a new syntactic category: $number$. A number is an expression. Numbers are either 0, the sum, product, or power of two numbers. We also add the sets $\mathbb{N}$ and succ.

$$
\begin{array}{lll}
\ldots & & \\
expression & ::= & \ldots \\
 & & number \\
set & ::= & \ldots \\
 & & \mathbb{N} \\
 & & \text{succ} \\
number & ::= & 0 \\
 & & number + number \\
 & & number * number \\
 & & number \mathbin{\widehat{}} number
\end{array}
$$

$\mathcal{SY}14$

### 6.2 Peano Axioms and Recursive Definitions

The following predicates are added systematically to the hypotheses of sequents to prove:

$$
\begin{array}{l}
0 \in \mathbb{N} \\
\text{succ} \in \mathbb{N} \rightarrowtail \mathbb{N} \setminus \{0\} \\
\forall S \cdot \left( \begin{array}{l} 0 \in S \\ \forall n \cdot ( n \in S \Rightarrow \text{succ}(n) \in S ) \\ \Rightarrow \\ \mathbb{N} \subseteq S \end{array} \right) \\
\forall a \cdot ( a \in \mathbb{N} \Rightarrow a + 0 = a ) \\
\forall a \cdot ( a \in \mathbb{N} \Rightarrow a * 0 = 0 ) \\
\forall a \cdot ( a \in \mathbb{N} \Rightarrow a \mathbin{\widehat{}} 0 = \text{succ}(0) ) \\
\forall a, b \cdot ( a \in \mathbb{N} \wedge b \in \mathbb{N} \Rightarrow a + \text{succ}(b) = \text{succ}(a + b) ) \\
\forall a, b \cdot ( a \in \mathbb{N} \wedge b \in \mathbb{N} \Rightarrow a * \text{succ}(b) = (a * b) + a ) \\
\forall a, b \cdot ( a \in \mathbb{N} \wedge b \in \mathbb{N} \Rightarrow a \mathbin{\widehat{}} \text{succ}(b) = (a \mathbin{\widehat{}} b) * a )
\end{array}
$$

$\mathcal{AR}1$

### 6.3 Extension of the Arithmetic Language

We introduce the classical binary relations on numbers, the finiteness predicate, the interval between two numbers, and the subtraction, division, and modulo constructs.

$$
\begin{array}{lcl}
\ldots & & \\
predicate & ::= & \ldots \\
 & & number \leq number \\
 & & number < number \\
 & & \text{finite}(set) \\
set & ::= & \ldots \\
 & & number \mathinner{..} number \\
number & ::= & \ldots \\
 & & number - number \\
 & & number \mathbin{/} number \\
 & & number \bmod number \\
 & & \text{card}(set)
\end{array}
$$

$\mathcal{SY}15$

| Operator | Predicate | Rewritten |
|---|---|---|
| smaller than or equal | $a \leq b$ | $\exists c \cdot (\, c \in \mathbb{N} \;\wedge\; b = a + c \,)$ |
| smaller than | $a < b$ | $a \leq b \;\wedge\; a \neq b$ |
| interval | $c \in a \mathinner{..} b$ | $a \leq c \;\wedge\; c \leq b$ |
| subtraction | $c = a - b$ | $a = b + c$ |
| division | $c = a/b$ | $\exists r \cdot (\, r \in \mathbb{N} \;\wedge\; r < b \;\wedge\; a = c * b + r \,)$ |
| modulo | $r = a \bmod b$ | $a = (a/b) * b + r$ |
| finiteness | $\text{finite}(s)$ | $\exists n, f \cdot (\, n \in \mathbb{N} \;\wedge\; f \in 1 \mathinner{..} n \rightarrowtail\!\!\!\!\!\rightarrow s \,)$ |
| cardinality | $n = \text{card}(s)$ | $\exists f \cdot f \in 1 \mathinner{..} n \rightarrowtail s$ |

$\mathcal{AR}2$

The subtraction, division, modulo, and cardinal constructs are subjected to some well-formedness conditions, which are the following:

| Number | Well-definedness condition |
|---|---|
| $a - b$ | $b \leq a$ |
| $a/b$ | $b \neq 0$ |
| $a \mod b$ | $b \neq 0$ |
| card$(s)$ | finite$(s)$ |

$\mathcal{WF}3$

# Event-B: Syntax of the Mathematical Language

Christophe Métayer (ClearSy)

Laurent Voisin (ETH Zürich)

May 31st, 2005

# Contents

# 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 describes its (concrete) syntax. Chapter three introduces the notion of well-formed and well-typed formula (static checks). Finally, chapter four gives the well-definedness conditions for a formula (dynamic check).

# 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 | 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 |
| $\Leftrightarrow$ | U+21D4 | LOGICAL EQUIVALENCE |
| $\Rightarrow$ | U+21D2 | LOGICAL IMPLICATION |
| $\wedge$ | U+2227 | LOGICAL AND |
| $\vee$ | U+2228 | LOGICAL OR |
| $\neg$ | U+00AC | NOT SIGN |
| $\top$ | U+22A4 | TRUE PREDICATE |
| $\bot$ | U+22A5 | FALSE PREDICATE |
| $\forall$ | U+2200 | FOR ALL |
| $\exists$ | U+2203 | THERE EXISTS |
| , | U+002C | COMMA |
| $\cdot$ | U+00B7 | MIDDLE DOT |

The symbolic tokens used to build predicates from expressions are:

| Token | Code point | Token name |
|---|---|---|
| $=$ | U+003D | EQUALS SIGN |
| $\neq$ | U+2260 | NOT EQUAL TO |
| $<$ | U+003C | LESS-THAN SIGN |
| $\leq$ | U+2264 | LESS THAN OR EQUAL TO |
| $>$ | U+003E | GREATER-THAN SIGN |
| $\geq$ | U+2265 | GREATER THAN OR EQUAL TO |
| $\in$ | U+2208 | ELEMENT OF |
| $\notin$ | U+2209 | NOT AN ELEMENT OF |
| $\subset$ | U+2282 | SUBSET OF |
| $\not\subset$ | U+2284 | NOT A SUBSET OF |
| $\subseteq$ | U+2286 | SUBSET OF OR EQUAL TO |
| $\not\subseteq$ | 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 |
|---|---|---|
| $\leftrightarrow$ | U+2194 | RELATION |
| $\nleftrightarrow$ | U+E100 | TOTAL RELATION |
| $\leftrightarrow\!\!\!\!\rightarrow$ | U+E101 | SURJECTIVE RELATION |
| $\nleftrightarrow\!\!\!\!\rightarrow$ | U+E102 | TOTAL SURJECTIVE RELATION |
| $\rightarrow\!\!\!\!\!\!\rightarrow$ | U+21F8 | PARTIAL FUNCTION |
| $\rightarrow$ | U+2192 | TOTAL FUNCTION |
| $\rightarrowtail\!\!\!\!\!\!\rightarrow$ | U+2914 | PARTIAL INJECTION |
| $\rightarrowtail$ | U+21A3 | TOTAL INJECTION |
| $\twoheadrightarrow\!\!\!\!\!\!\rightarrow$ | U+2900 | PARTIAL SURJECTION |
| $\twoheadrightarrow$ | U+21A0 | TOTAL SURJECTION |
| $\rightarrowtail\!\!\!\!\!\!\twoheadrightarrow$ | 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 |
| $\mapsto$ | U+21A6 | MAPLET |
| $\varnothing$ | U+2205 | EMPTY SET |
| $\cap$ | U+2229 | INTERSECTION |
| $\cup$ | U+222A | UNION |
| $\setminus$ | U+2216 | SET MINUS |
| $\times$ | 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 |
| $\mapsto$ | U+21A6 | MAPLET |
| $\leftarrow\!\!\!\!\!\!+$ | U+E103 | RELATION OVERRIDING |
| $\circ$ | U+2218 | BACKWARD COMPOSITION |
| ; | U+003B | FORWARD COMPOSITION |
| $\otimes$ | U+2297 | DIRECT PRODUCT |
| $\parallel$ | U+2225 | PARALLEL PRODUCT |
| $^{-1}$ | U+223C | TILDE OPERATOR |
| $\lhd$ | U+25C1 | DOMAIN RESTRICTION |
| $\leftarrow\!\!\!\!\lhd$ | U+2A64 | DOMAIN SUBTRACTION |
| $\rhd$ | U+25B7 | RANGE RESTRICTION |
| $\rhd\!\!\!\!\rightarrow$ | U+2A65 | RANGE SUBTRACTION |

The following symbolic tokens are used in quantified expressions:

| Token | Code point | Token name |
|---|---|---|
| $\lambda$ | U+03BB | LAMBDA |
| $\bigcap$ | U+22C2 | N-ARY INTERSECTION |
| $\bigcup$ | U+22C3 | N-ARY UNION |
| $\mid$ | 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 |
| $\div$ | U+00F7 | DIVISION SIGN |
| $\widehat{\ }$ | 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*⟩

| ⟨*predicate*⟩ '∨' ⟨*predicate*⟩
| '¬' ⟨*predicate*⟩
| '⊤'
| '⊥'
| '∀' ⟨*ident-list*⟩ '·' ⟨*predicate*⟩
| '∃' ⟨*ident-list*⟩ '·' ⟨*predicate*⟩
| 'finite' '(' ⟨*expression*⟩ ')'
| ⟨*expression*⟩ '=' ⟨*expression*⟩
| ⟨*expression*⟩ '∈' ⟨*expression*⟩
| ⟨*expression*⟩ '≤' ⟨*expression*⟩
| ...

⟨*ident-list*⟩ ::= ⟨*ident-list*⟩ ',' ⟨*ident*⟩
| ⟨*ident*⟩

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 10).

### 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

$$P \Rightarrow Q \Rightarrow R$$

$$P \wedge Q \wedge R$$

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 $P \land Q \land R$ will be parsed as if it actually were written as $(P \land Q) \land R$.

The rationale for these associativities is quite simple. Operator $\land$ is associative in the algebraic sense, so formulae $(P \land Q) \land R$ and $P \land (Q \land R)$ 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 $\lor$.

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 $\bot$). 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 $(P \Leftrightarrow Q) \land (Q \Leftrightarrow R)$. 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 those 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

$$P \Rightarrow Q \Leftrightarrow R$$

$$P \land Q \lor R$$

$$\neg P \land Q$$

$$\forall x \cdot P \lor Q$$

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 '$\land$' and '$\lor$' to have the same priority, while others expect operator '$\land$' to have higher priority. So when faced with formula $P \lor Q \land R$, some people read it as $(P \lor Q) \land R$ 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)

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

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:

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

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:

$$\begin{array}{lll} \langle \textit{predicate} \rangle & ::= & \{ \langle \textit{quantifier} \rangle \} \langle \textit{unquantified-predicate} \rangle \\ \\ \langle \textit{quantifier} \rangle & ::= & \text{`}\forall\text{'} \langle \textit{ident-list} \rangle \text{`}\cdot\text{'} \\ & | & \text{`}\exists\text{'} \langle \textit{ident-list} \rangle \text{`}\cdot\text{'} \\ \\ \langle \textit{ident-list} \rangle & ::= & \langle \textit{ident} \rangle \{ \text{`,'} \langle \textit{ident} \rangle \} \\ \\ \langle \textit{unquantified-predicate} \rangle & ::= & \langle \textit{simple-predicate} \rangle [ \text{`}\Rightarrow\text{'} \langle \textit{simple-predicate} \rangle ] \\ & | & \langle \textit{simple-predicate} \rangle [ \text{`}\Leftrightarrow\text{'} \langle \textit{simple-predicate} \rangle ] \\ \\ \langle \textit{simple-predicate} \rangle & ::= & \langle \textit{literal-predicate} \rangle \{ \text{`}\wedge\text{'} \langle \textit{literal-predicate} \rangle \} \\ & | & \langle \textit{literal-predicate} \rangle \{ \text{`}\vee\text{'} \langle \textit{literal-predicate} \rangle \} \\ \\ \langle \textit{literal-predicate} \rangle & ::= & \{ \text{`}\neg\text{'} \} \langle \textit{atomic-predicate} \rangle \end{array}$$

⟨*atomic-predicate*⟩ ::= '⊥'
| '⊤'
| 'finite' '(' ⟨*expression*⟩ ')'
| ⟨*pair-expression*⟩ ⟨*relop*⟩ ⟨*pair-expression*⟩
| '(' ⟨*predicate*⟩ ')'

⟨*relop*⟩ ::= '=' | '≠'
| '∈' | '∉' | '⊂' | '⊄' | '⊆' | '⊈'
| '<' | '≤' | '>' | '≥'

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 $\lambda x \cdot x \in \mathbb{Z} \mid x = id(\mathbb{Z})$ is not well-formed, one must write instead $(\lambda x \cdot x \in \mathbb{Z} \mid x) = id(\mathbb{Z})$.

## 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 '↦'. 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 is just a short-hand for the first-one, which allows to write things more compactly. That second form is $\{E \mid P\}$. The difference with 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 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 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 inter$(\{E \mid P\})$.

**Projections.** In classical B [1], the first and second projection operators take two sets as argument, as for instance in the expression $\text{prj}_1(A, B)$. In that expression, arguments $A$ and $B$ are used for two different purposes. On the one end, they allow to infer the type associated to the instantiated operator. On the other hand, they define the domain of the instantiated operator, which is $A \times B$.

This approach seems unnecessarily restrictive, as it puts a strong constraint on the domain of the operator, namely that it must be a cartesian product. So, in event-B, these operators become unary and take a relation as argument. The argument is then their domain. The upgrade path from classical B is quite straightforward, just replace $\text{prj}_1(A, B)$ by $\text{prj}_1(A \times B)$.

### 3.3.2 A First Attempt

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

⟨*expression*⟩ ::= ⟨*expression*⟩ ⟨*binary-operator*⟩ ⟨*expression*⟩
| ⟨*unary-operator*⟩ ⟨*expression*⟩
| ⟨*expression*⟩ '$^{-1}$'
| ⟨*expression*⟩ '[' ⟨*expression*⟩ ']'
| ⟨*expression*⟩ '(' ⟨*expression*⟩ ')'
| '$\lambda$' ⟨*ident-pattern*⟩ '$\cdot$' ⟨*predicate*⟩ '|' ⟨*expression*⟩
| ⟨*quantifier*⟩ ⟨*ident-list*⟩ '$\cdot$' ⟨*predicate*⟩ '|' ⟨*expression*⟩
| ⟨*quantifier*⟩ ⟨*expression*⟩ '|' ⟨*predicate*⟩
| '{' ⟨*ident-list*⟩ '$\cdot$' ⟨*predicate*⟩ '|' ⟨*expression*⟩ '}'
| '{' ⟨*expression*⟩ '|' ⟨*predicate*⟩ '}'
| 'bool' '(' ⟨*predicate*⟩ ')'
| '{' [ ⟨*expression-list*⟩ ] '}'
| '(' ⟨*expression*⟩ ')'
| '$\varnothing$'
| '$\mathbb{Z}$' | '$\mathbb{N}$' | '$\mathbb{N}_1$'
| 'BOOL' | 'TRUE' | 'FALSE'
| ⟨*ident*⟩
| ⟨*integer-literal*⟩

⟨*binary-operator*⟩ ::= ‘$\mapsto$’ | ‘$\leftrightarrow$’ | ‘$\nleftrightarrow$’ | ‘$\leftrightarrow\!\!\!\!\!\rightarrow$’ | ‘$\nleftrightarrow\!\!\!\!\!\rightarrow$’ | ‘$\rightarrow\!\!\!\!\!\!\mid$’ | ‘$\rightarrow$’ | ‘$\rightarrow\!\!\!\!\!\!\mid\!\!\!\!\!\rightarrow$’ | ‘$\rightarrow\!\!\!\!\!\rightarrow$’ | ‘$\rightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow$’
| ‘$\twoheadrightarrow$’ | ‘$\rightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow$’ | ‘$\cup$’ | ‘$\cap$’ | ‘$\setminus$’ | ‘$\times$’ | ‘$\otimes$’ | ‘$\parallel$’ | ‘$\circ$’ | ‘;’ | ‘$\lhd$’ |
‘$\rhd$’ | ‘$\trianglelefteq$’ | ‘$\trianglerighteq$’ | ‘$\mid\!\!\!\!\rightarrow$’ | ‘..’ | ‘+’ | ‘−’ | ‘$*$’ | ‘$\div$’ | ‘mod’ | ‘$\frown$’

⟨*unary-operator*⟩ ::= ‘−’ | ‘card’ | ‘$\mathbb{P}$’ | ‘$\mathbb{P}_1$’ | ‘union’ | ‘inter’ | ‘dom’ | ‘ran’ |
‘$\mathrm{prj}_1$’ | ‘$\mathrm{prj}_2$’ | ‘id’

⟨*quantifier*⟩ ::= ‘$\bigcup$’ | ‘$\bigcap$’

⟨*ident-pattern*⟩ ::= ⟨*ident-pattern*⟩ ‘$\mapsto$’ ⟨*ident-pattern*⟩
| ‘(’ ⟨*ident-pattern*⟩ ‘)’
| ⟨*ident*⟩

⟨*expression-list*⟩ ::= ⟨*expression-list*⟩ ‘,’ ⟨*expression*⟩
| ⟨*expression*⟩

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.

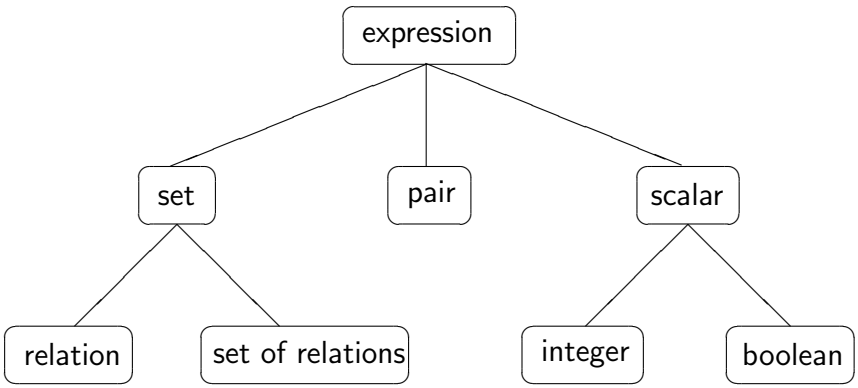



Figure 3.1: Kinds of expressions

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

| 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. | $S \nrightarrow T$ |
| 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. | $i \mathinner{..} j$ |
| 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

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 page before. 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:

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

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.

**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. However, we decide to parse them as right-associative. That choice is justified by the fact that we will parse function application as left-associative (this will be stated when we reach the *Relational and functional image* paragraph on the following page). As a consequence, one can write $f(a)(b)$ when one actually means $(f(a))(b)$. Then, to be consistent, one should also be able to describe properties of function $f$ without parenthesis, so formula $f \in A \nrightarrow B \nrightarrow C$ should be parsed as $f \in A \nrightarrow (B \nrightarrow C)$.

**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 next 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 $(a \mapsto b) \mapsto c \in (A \times B) \times C$.

| | $\cup$ | $\cap$ | $\setminus$ | $\times$ | $\circ$ | $;$ | $\otimes$ | $\parallel$ | $\lhd\!\!\!+$ | $\lhd$ | $\lhd\!\!\!=$ | $\rhd$ | $\rhd\!\!\!=$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\cup$ | $\times$ | | | | | | | | | | | | |
| $\cap$ | | $\times$ | $\times$ | | | | | | | | | $\times$ | $\times$ |
| $\setminus$ | | | | | | | | | | | | | |
| $\times$ | | | | | | | | | | | | | |
| $\circ$ | | | | | $\times$ | | | | | | | | |
| $;$ | | | | | | $\times$ | | | | | | $\times$ | $\times$ |
| $\otimes$ | | | | | | | | | | | | | |
| $\parallel$ | | | | | | | | | | | | | |
| $\lhd\!\!\!+$ | | | | | | | | | $\times$ | | | | |
| $\lhd$ | | $\times$ | $\times$ | | | $\times$ | $\times$ | | | | | $\times$ | $\times$ |
| $\lhd\!\!\!=$ | | $\times$ | $\times$ | | | $\times$ | $\times$ | | | | | $\times$ | $\times$ |
| $\rhd$ | | | | | | | | | | | | | |
| $\rhd\!\!\!=$ | | | | | | | | | | | | | |

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 $a \mathinner{..} b \mathinner{..} c$). 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 $\mathbb{Z}$, which is clearly not a pair. However, we didn't find a better way to name non-terminals (just numbering them 1, 2,... would miss some information).

⟨*expression*⟩ ::= '$\lambda$' ⟨*ident-pattern*⟩ '$\cdot$' ⟨*predicate*⟩ '|' ⟨*expression*⟩
| '$\bigcup$' ⟨*ident-list*⟩ '$\cdot$' ⟨*predicate*⟩ '|' ⟨*expression*⟩
| '$\bigcup$' ⟨*expression*⟩ '|' ⟨*predicate*⟩
| '$\bigcap$' ⟨*ident-list*⟩ '$\cdot$' ⟨*predicate*⟩ '|' ⟨*expression*⟩
| '$\bigcap$' ⟨*expression*⟩ '|' ⟨*predicate*⟩
| ⟨*pair-expression*⟩

⟨*ident-pattern*⟩ ::= ⟨*ident-pattern*⟩ { '$\mapsto$' ⟨*ident-pattern*⟩ }
| '(' ⟨*ident-pattern*⟩ ')'
| ⟨*ident*⟩

⟨*pair-expression*⟩ ::= ⟨*relation-set-expr*⟩ { '$\mapsto$' ⟨*relation-set-expr*⟩ }

⟨*relation-set-expr*⟩ ::= ⟨*set-expr*⟩ { ⟨*relational-set-op*⟩ ⟨*set-expr*⟩ }

⟨*relational-set-op*⟩ ::= '$\leftrightarrow$' | '$\nleftrightarrow$' | '$\twoheadleftarrow$' | '$\nleftrightarrow$'
| '$\rightarrowtail$' | '$\rightarrow$' | '$\rightarrowtail$' | '$\rightarrow$' | '$\twoheadrightarrow$' | '$\rightarrow$' | '$\rightarrow$'

⟨*set-expr*⟩ ::= ⟨*interval-expr*⟩ { '$\cup$' ⟨*interval-expr*⟩ }
| ⟨*interval-expr*⟩ { '$\times$' ⟨*interval-expr*⟩ }
| ⟨*interval-expr*⟩ { '$\lhd$' ⟨*interval-expr*⟩ }
| ⟨*interval-expr*⟩ { '$\circ$' ⟨*interval-expr*⟩ }
| ⟨*interval-expr*⟩ '$\|$' ⟨*interval-expr*⟩
| [ ⟨*domain-modifier*⟩ ] ⟨*relation-expr*⟩

⟨*domain-modifier*⟩ ::= ⟨*interval-expr*⟩ ( '$\lhd$' | '$\trianglelefteq$' )

⟨*relation-expr*⟩ ::= ⟨*interval-expr*⟩ '$\otimes$' ⟨*interval-expr*⟩
| ⟨*interval-expr*⟩ { ';' ⟨*interval-expr*⟩ }
[ ⟨*range-modifier*⟩ ]
| ⟨*interval-expr*⟩ { '$\cap$' ⟨*interval-expr*⟩ }
[ '\' ⟨*interval-expr*⟩ | ⟨*range-modifier*⟩ ]

⟨*range-modifier*⟩ ::= ( '$\rhd$' | '$\trianglerighteq$' ) ⟨*interval-expr*⟩

⟨*interval-expr*⟩ ::= ⟨*arithmetic-expr*⟩ [ '..' ⟨*arithmetic-expr*⟩ ]

⟨*arithmetic-expr*⟩ ::= [ '$-$' ] ⟨*term*⟩ { ( '$+$' | '$-$' ) ⟨*term*⟩ }

⟨*term*⟩ ::= ⟨*factor*⟩ { ( '$*$' | '$\div$' | 'mod' ) ⟨*factor*⟩ }

⟨*factor*⟩ ::= ⟨*image*⟩ [ '$\frown$' ⟨*image*⟩ ]

⟨*image*⟩ ::= ⟨*primary*⟩ { '[' ⟨*expression*⟩ ']' | '(' ⟨*expression*⟩ ')' }

⟨*primary*⟩ ::= ⟨*simple-expr*⟩ { '$^{-1}$' }

⟨*simple-expr*⟩ ::= 'bool' '(' ⟨*predicate*⟩ ')'
| ⟨*unary-op*⟩ '(' ⟨*expression*⟩ ')'
| '(' ⟨*expression*⟩ ')'
| '{' ⟨*ident-list*⟩ '·' ⟨*predicate*⟩ '|' ⟨*expression*⟩ '}'
| '{' ⟨*expression*⟩ '|' ⟨*predicate*⟩ '}'
| '{' [ ⟨*expression*⟩ { ',' ⟨*expression*⟩ } ] '}'
| '$\mathbb{Z}$' | '$\mathbb{N}$' | '$\mathbb{N}_1$' | 'BOOL' | 'TRUE' | 'FALSE' | '$\varnothing$'
| ⟨*ident*⟩
| ⟨*integer-literal*⟩

⟨*unary-op*⟩ ::= 'card' | '$\mathbb{P}$' | '$\mathbb{P}_1$' | 'union' | 'inter' | 'dom' | 'ran' | 'prj$_1$'
| 'prj$_2$' | 'id'

# 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:

- well-formedness,
- 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 10 and Section 3.3.6 on page 19. 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:

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

where

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

The production rules for lists of identifiers and identifiers are:

$$\begin{aligned} \text{ident-list: } L &::= I_1\ I_2\ \ldots\ I_n \\ \text{ident: } I &::= [name] \end{aligned}$$

where

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

The production rules for expressions are:

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

where

$$\textit{expr-binop} \in \left\{ \begin{array}{l} \text{funimage, relimage, mapsto,} \\ \text{rel, trel, srel, strel,} \\ \text{pfun, tfun, pinj, tinj, psur, tsur, tbij,} \\ \text{bunion, binter, setminus, cprod, dprod, pprod,} \\ \text{bcomp, fcomp, ovl, domres, domsub, ranres, ransub,} \\ \text{upto, plus, minus, mul, div, mod, expn} \end{array} \right\}$$

$$\textit{expr-unop} \in \left\{ \begin{array}{l} \text{uminus, converse, card, pow, pow1,} \\ \text{union, inter, dom, ran, prj1, prj2, id} \end{array} \right\}$$

$\textit{expr-quant} \in \{\text{qunion, qinter, cset}\}$
$\textit{expr-lit} \in \{\text{integer, natural, natural1, bool, true, false, emptyset}\}$
*int-lit* is an integer number.

## 4.2 Well-formedness

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 well-formed, 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. This is a big win in formula legibility.

For instance, the following formula is ill-formed (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 ill-formed (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 well-formedness 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 *wff* is synthesized and contains a boolean value which is TRUE if and only if the formula rooted at the current node is well-formed.

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

pred-bin: $P ::= P_1\ P_2$ [*pred-binop*]

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

pred-una: $P ::= P_1$

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

pred-quant: $P ::= L_1\ P_1$ [*pred-quant*]

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

pred-lit: $P ::=$ [*pred-lit*]

$$P.\mathit{bound} = \varnothing$$
$$P.\mathit{free} = \varnothing$$
$$P.\mathit{wff} = \mathrm{TRUE}$$

pred-simp: $P ::= E_1$

$$P.\mathit{bound} = E_1.\mathit{bound}$$
$$P.\mathit{free} = E_1.\mathit{free}$$
$$P.\mathit{wff} = E_1.\mathit{wff}$$

pred-rel: $P ::= E_1\ E_2$ [*pred-relop*]

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

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

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

expr-bin: $E ::= E_1\ E_2$ [*expr-binop*]

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

expr-una: $E ::= E_1$ [*expr-unop*]

$$E.\mathit{bound} = E_1.\mathit{bound}$$
$$E.\mathit{free} = E_1.\mathit{free}$$
$$E.\mathit{wff} = E_1.\mathit{wff}$$

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

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

expr-quant1: $E ::= L_1\ P_1\ E_1$ [*expr-quant*]

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

expr-quant2: $E ::= E_1\ P_1$ [*expr-quant*]

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

expr-bool: $E ::= P_1$

$$E.bound = P_1.bound$$
$$E.free = P_1.free$$
$$E.wff = P_1.wff$$

expr-eset: $E ::= M$

$$E.bound = M.bound$$
$$E.free = M.free$$
$$E.wff = M.wff$$

expr-ident: $E ::= I_1$

$$E.bound = \varnothing$$
$$E.free = \{I_1.name\}$$
$$E.wff = \text{TRUE}$$

expr-atom: $E ::=$ [*expr-lit*]

$$E.bound = \varnothing$$
$$E.free = \varnothing$$
$$E.wff = \text{TRUE}$$

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

$$E.\textit{bound} = \varnothing$$
$$E.\textit{free} = \varnothing$$
$$E.\textit{wff} = \text{TRUE}$$

pattern: $Q ::= Q_1\ Q_2$

$$Q.\textit{bound} = \varnothing$$
$$Q.\textit{free} = Q_1.\textit{free} \cup Q_2.\textit{free}$$
$$Q.\textit{wff} = \text{TRUE}$$

pattern-ident: $Q ::= I_1$

$$Q.\textit{bound} = \varnothing$$
$$Q.\textit{free} = \{I_1.\textit{name}\}$$
$$Q.\textit{wff} = \text{TRUE}$$

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

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

## 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, can not, 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 40).

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 well-formedness. 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 $\mathbb{Z} \times \text{BOOL}$.

A type variable is a meta-variable that can denote any type. In the sequel, we shall use lowercase Greek letters ($\alpha$, $\beta$, $\gamma$, ...) 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

$$\{\text{`}\texttt{a}\text{'} \mapsto \mathbb{Z}, \text{`}\texttt{b}\text{'} \mapsto \mathbb{P}(\mathbb{Z} \times \text{BOOL}), \text{`}\texttt{c}\text{'} \mapsto \alpha\}$$

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

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 $\alpha \times \text{BOOL} \equiv \mathbb{Z} \times \beta$ is satisfiable (take $\mathbb{Z}$ for $\alpha$ and BOOL for $\beta$). 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 $\alpha$), while the type equation $\alpha \equiv \beta$, although satisfiable, is not uniquely satisfiable (to satisfy it, we only need that $\alpha$ and $\beta$ 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 $\{\alpha \equiv \mathbb{Z},\ \beta \equiv \text{BOOL}\}$ is (uniquely) satisfiable, while the set $\{\alpha \equiv \mathbb{Z},\ \alpha \equiv \text{BOOL}\}$ 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 $R.styeqs$ is uniquely satisfiable. Moreover, in case of success, the resulting typing environment is $R.styenv$, 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):

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

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

$$A.inherited = B.synthesized$$

means

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

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, $\alpha$ denotes a type variable such that $\alpha \notin P.ityvars$.

The set of equations of the attribute grammar is:

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

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

pred-una: $P ::= P_1$

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

pred-quant: $P ::= L_1\ P_1$ [*pred-quant*]

$$L_1.inherited = P.inherited$$
$$P_1.inherited = L_1.synthesized$$
$$P.synthesized = P_1.synthesized$$

pred-lit: $P ::=$ [*pred-lit*]

$$P.synthesized = P.inherited$$

pred-simp: $P ::= E_1$

Let $\alpha$ be a fresh type variable in

$$E_1.ityvars = P.ityvars \cup \{\alpha\}$$
$$E_1.ityenv = P.ityenv$$
$$E_1.ityeqs = P.ityeqs$$
$$P.styvars = E_1.styvars$$
$$P.styenv = E_1.styenv$$
$$P.styeqs = E_1.styeqs \cup \{E1.type \equiv \mathbb{P}(\alpha)\}$$

pred-rel: $P ::= E_1\ E_2$ [*pred-relop*]

Let $\alpha$ be a fresh type variable in

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

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

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

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

$$I_1.inherited = L.inherited$$
$$I_2.inherited = I_1.synthesized$$
$$\vdots$$
$$I_n.inherited = I_{n-1}.synthesized$$
$$L.synthesized = I_n.synthesized$$

ident: $I ::= [name]$
 if $I.name \in \mathrm{dom}(I.ityenv)$ then
$I.synthesized = I.inherited$
$I.type = I.ityenv(I.name)$
 else let $\alpha$ be a fresh type variable in
$I.styvars = I.ityvars \cup \{\alpha\}$
$I.styenv = I.ityenv \cup \{I.name \mapsto \alpha\}$
$I.styeqs = I.ityeqs$
$I.type = \alpha$

expr-bin: $E ::= E_1\ E_2\ [expr\text{-}binop]$
 Let $\alpha$, $\beta$, $\gamma$ and $\delta$ be distinct fresh type variables in
$E_1.ityvars = E.ityvars \cup \{\alpha, \beta, \gamma, \delta\}$
$E_1.ityenv = E.ityenv$
$E_1.ityeqs = E.ityeqs$
$E_2.inherited = E_1.synthesized$
$E.styvars = E_2.styvars$
$E.styenv = E_2.styenv$
$E.styeqs = E_2.styeqs \cup \mathcal{E}$
$E.type = \tau$
 where $\mathcal{E}$ and $\tau$ are defined in Table 4.1 on the next page.

expr-una: $E ::= E_1\ [expr\text{-}unop]$
 Let $\alpha$ and $\beta$ be distinct fresh type variables in
$E_1.ityvars = E.ityvars \cup \{\alpha, \beta\}$
$E_1.ityenv = E.ityenv$
$E_1.ityeqs = E.ityeqs$
$E.styvars = E_1.styvars$
$E.styenv = E_1.styenv$
$E.styeqs = E_1.styeqs \cup \mathcal{E}$
$E.type = \tau$
 where $\mathcal{E}$ and $\tau$ are defined in Table 4.2 on page 32.

expr-lambda: $E ::= Q_1\ P_1\ E_1$
$Q_1.inherited = E.inherited$
$P_1.inherited = Q_1.synthesized$
$E_1.inherited = P_1.synthesized$
$E.synthesized = E_1.synthesized$
$E.type = \mathbb{P}(Q_1.type \times E_1.type)$

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

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

| *E.expr-unop* | $\mathcal{E}$ | $\tau$ |
|---|---|---|
| uminus | $\{\ E_1.type \equiv \mathbb{Z}\ \}$ | $\mathbb{Z}$ |
| converse | $\{\ E_1.type \equiv \mathbb{P}(\alpha \times \beta)\ \}$ | $\mathbb{P}(\beta \times \alpha)$ |
| card | $\{\ E_1.type \equiv \mathbb{P}(\alpha)\ \}$ | $\mathbb{Z}$ |
| pow, pow1 | $\{\ E_1.type \equiv \mathbb{P}(\alpha)\ \}$ | $\mathbb{P}(\mathbb{P}(\alpha))$ |
| union, inter | $\{\ E_1.type \equiv \mathbb{P}(\mathbb{P}(\alpha))\ \}$ | $\mathbb{P}(\alpha)$ |
| dom | $\{\ E_1.type \equiv \mathbb{P}(\alpha \times \beta)\ \}$ | $\mathbb{P}(\alpha)$ |
| ran | $\{\ E_1.type \equiv \mathbb{P}(\alpha \times \beta)\ \}$ | $\mathbb{P}(\beta)$ |
| prj1 | $\{\ E_1.type \equiv \mathbb{P}(\alpha \times \beta)\ \}$ | $\mathbb{P}(\alpha \times \beta \times \alpha)$ |
| prj2 | $\{\ E_1.type \equiv \mathbb{P}(\alpha \times \beta)\ \}$ | $\mathbb{P}(\alpha \times \beta \times \beta)$ |
| id | $\{\ E_1.type \equiv \mathbb{P}(\alpha)\ \}$ | $\mathbb{P}(\alpha \times \alpha)$ |

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

expr-quant1: $E ::= L_1\ P_1\ E_1$ [*expr-quant*]
Let $\alpha$ be a fresh type variable in
$L_1.ityvars = E.ityvars \cup \{\alpha\}$
$L_1.ityenv = E.ityenv$
$L_1.ityeqs = E.ityeqs$
$P_1.inherited = L_1.synthesized$
$E_1.inherited = P_1.synthesized$
$E.styvars = E_1.styvars$
$E.styenv = E_1.styenv$
$E.styeqs = E_1.styeqs \cup \mathcal{E}$
$E.type = \tau$
where $\mathcal{E}$ and $\tau$ are defined in the following table.

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

expr-quant2: $E ::= E_1\ P_1$ [*expr-quant*]
Let $\alpha$ be a fresh type variable in
$E_1.ityvars = E.ityvars \cup \{\alpha\}$
$E_1.ityenv = E.ityenv$
$E_1.ityeqs = E.ityeqs$
$P_1.inherited = E_1.synthesized$
$E.styvars = P_1.styvars$
$E.styenv = P_1.styenv$
$E.styeqs = P_1.styeqs \cup \mathcal{E}$
$E.type = \tau$
where $\mathcal{E}$ and $\tau$ are defined in the following table.

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

expr-bool: $E ::= P_1$
$P_1.inherited = E.inherited$
$E.synthesized = P_1.synthesized$
$E.type = \text{BOOL}$

expr-eset: $E ::= M$
$M.inherited = E.inherited$
$E.synthesized = M.synthesized$
$E.type = \mathbb{P}(M.type)$

expr-ident: $E ::= I_1$

$$I_1.inherited = E.inherited$$
$$E.synthesized = I_1.synthesized$$
$$E.type = I_1.type$$

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

Let $\alpha$ be a fresh type variable in

$$E.styvars = E.ityvars \cup \{\alpha\}$$
$$E.styenv = E.ityenv$$
$$E.styeqs = E.ityeqs$$
$$E.type = \tau$$

where $\tau$ is defined in the following table.

| $E.expr\text{-}lit$ | $\tau$ |
|---|---|
| integer, natural, natural1 | $\mathbb{P}(\mathbb{Z})$ |
| bool | $\mathbb{P}(\text{BOOL})$ |
| true, false | BOOL |
| emptyset | $\mathbb{P}(\alpha)$ |

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

$$E.synthesized = E.inherited$$
$$E.type = \mathbb{Z}$$

pattern: $Q ::= Q_1\ Q_2$

$$Q_1.inherited = Q.inherited$$
$$Q_2.inherited = Q_1.synthesized$$
$$Q.synthesized = Q_2.synthesized$$
$$Q.type = Q_1.type \times Q_2.type$$

pattern-ident: $Q ::= I_1$

$$I_1.inherited = Q.inherited$$
$$Q.synthesized = I_1.synthesized$$
$$Q.type = I_1.type$$

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

### 4.3.3 Examples

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

**Formula $x \in \mathbb{Z} \ \wedge\ 1 \leq x$.** 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.

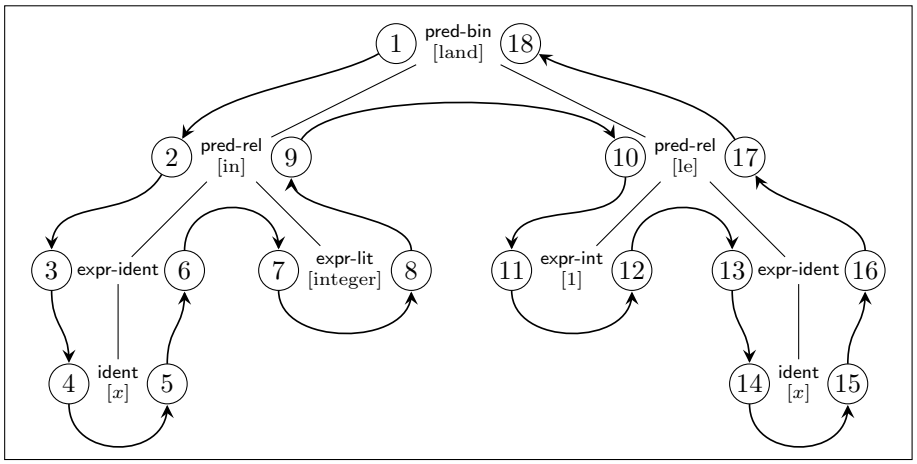



Figure 4.1: Type-check of formula $x \in \mathbb{Z} \ \wedge\ 1 \leq x$.

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

$$
1: \left| \begin{array}{l} \mathit{ityvars} = \varnothing \\ \mathit{ityenv}\ = \varnothing \\ \mathit{ityeqs}\ = \varnothing \end{array} \right.
$$

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

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

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:

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

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

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

Continuing our traversal of the tree, we get:

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

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:

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

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

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

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 $\{x \mapsto \mathbb{Z}\}$.

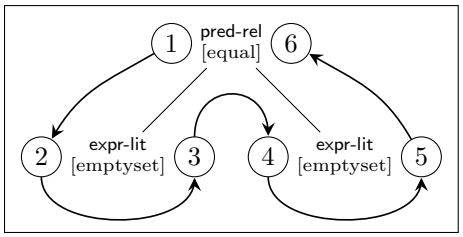



Figure 4.2: Type-check of formula $\varnothing = \varnothing$.

**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):

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

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

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

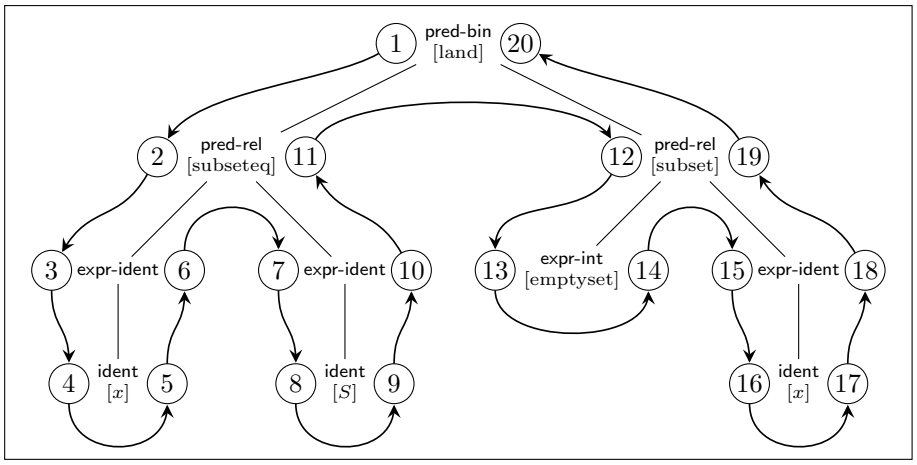



Figure 4.3: Type-check of formula $x \subseteq S \ \wedge \ \varnothing \subset x$.

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:
$$\left|\begin{array}{ll} \mathit{ityvars} & = \varnothing \\ \mathit{ityenv} & = \{S \mapsto \mathbb{P}(S)\} \\ \mathit{ityeqs} & = \varnothing \end{array}\right.$$

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

5, 6:
$$\left|\begin{array}{ll} \mathit{styvars} & = \{\alpha, \beta\} \\ \mathit{styenv} & = \begin{Bmatrix} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{Bmatrix} \\ \mathit{styeqs} & = \varnothing \\ \mathit{type} & = \beta \end{array}\right.$$

7, 8:
$$\left|\begin{array}{ll} \mathit{ityvars} & = \{\alpha, \beta\} \\ \mathit{ityenv} & = \begin{Bmatrix} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{Bmatrix} \\ \mathit{ityeqs} & = \varnothing \end{array}\right.$$

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

11:
$$\left|\begin{array}{ll} \mathit{styvars} & = \{\alpha, \beta\} \\ \mathit{styenv} & = \begin{Bmatrix} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{Bmatrix} \\ \mathit{styeqs} & = \begin{Bmatrix} \beta & \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(S) & \equiv \mathbb{P}(\alpha) \end{Bmatrix} \end{array}\right.$$

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

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

14:
$$\left|\begin{array}{ll} \mathit{styvars} & = \{\alpha, \beta, \gamma, \delta\} \\ \mathit{styenv} & = \begin{Bmatrix} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{Bmatrix} \\ \mathit{styeqs} & = \begin{Bmatrix} \beta & \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(S) & \equiv \mathbb{P}(\alpha) \end{Bmatrix} \\ \mathit{type} & = \mathbb{P}(\delta) \end{array}\right.$$

15, 16:
$$\left|\begin{array}{ll} \mathit{ityvars} & = \{\alpha, \beta, \gamma, \delta\} \\ \mathit{ityenv} & = \begin{Bmatrix} S \mapsto \mathbb{P}(S), \\ x \mapsto \beta \end{Bmatrix} \\ \mathit{ityeqs} & = \begin{Bmatrix} \beta & \equiv \mathbb{P}(\alpha), \\ \mathbb{P}(S) & \equiv \mathbb{P}(\alpha) \end{Bmatrix} \end{array}\right.$$

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

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

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

Assuming that initially $x$ denotes an integer (non empty initial typing envi-

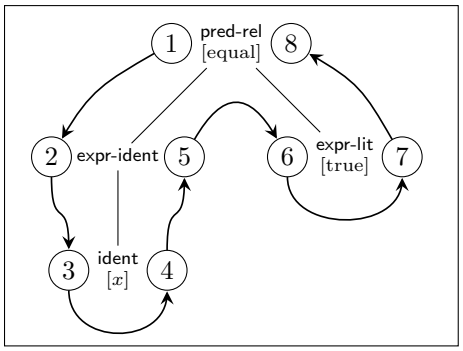



Figure 4.4: Type-check of formula $x = \text{TRUE}$.

ronment), we obtain the following values for attributes:

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

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

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

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 next 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 42 and 5.3 on page 43 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 $E.free$) 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** |
|---|---|
| $P \land Q \qquad P \Rightarrow Q$ | $\mathsf{WD}(P) \land (P \Rightarrow \mathsf{WD}(Q))$ |
| $P \lor Q$ | $\mathsf{WD}(P) \land (P \lor \mathsf{WD}(Q))$ |
| $P \Leftrightarrow Q$ | $\mathsf{WD}(P) \land \mathsf{WD}(Q)$ |
| $\neg P$ | $\mathsf{WD}(P)$ |
| $\forall L \cdot P \qquad \exists L \cdot P$ | $\forall L \cdot \mathsf{WD}(P)$ |
| $\top \qquad \bot$ | $\top$ |
| $\text{finite}(E)$ | $\mathsf{WD}(E)$ |
| $E = F \qquad E \neq F$<br>$E \in F \qquad E \notin F$<br>$E \subset F \qquad E \not\subset F$<br>$E \subseteq F \qquad E \not\subseteq F$ | $\mathsf{WD}(E) \land \mathsf{WD}(F)$ |

Table 5.1: WD lemmas for binary and unary expressions.

| **Expression** | **WD Lemma** |
|---|---|
| $F(E)$ | $\mathsf{WD}(F) \land \mathsf{WD}(E) \land E \in \mathrm{dom}(F)$ $\land\ F^{-1}; (\{E\} \lhd F) \subseteq \mathrm{id}(\mathrm{ran}(F))$ |
| $E[F]$ $\quad E \mapsto F$<br>$E \leftrightarrow F$ $\quad E \nleftrightarrow F$<br>$E \leftrightarrow\!\!\!\!\!\rightarrow F$ $\quad E \nleftrightarrow\!\!\!\!\!\rightarrow F$<br>$E \nrightarrow F$ $\quad E \rightarrow F$<br>$E \rightarrowtail F$ $\quad E \nrightarrow F$<br>$E \twoheadrightarrow F$ $\quad E \rightarrow\!\!\!\!\!\!\rightarrow F$<br>$E \rightarrowtail\!\!\!\!\!\!\rightarrow F$ $\quad E \cup F$<br>$E \cap F$ $\quad E \setminus F$<br>$E \times F$ $\quad E \otimes F$<br>$E \parallel F$ $\quad E \circ F$<br>$E ; F$ $\quad E \lhd\!\!\!\!- F$<br>$E \lhd F$ $\quad E \ntriangleleft F$<br>$E \rhd F$ $\quad E \ntriangleright F$<br>$E \mathinner{..} F$ $\quad E + F$<br>$E - F$ $\quad E * F$ | $\mathsf{WD}(E) \land \mathsf{WD}(F)$ |
| $E \div F$ $\quad E \bmod F$ | $\mathsf{WD}(E) \land \mathsf{WD}(F) \land F \neq 0$ |
| $E \mathbin{\widehat{}} F$ | $\mathsf{WD}(E) \land 0 \leq E \land \mathsf{WD}(F) \land 0 \leq F$ |
| $-E$ $\quad E^{-1}$<br>$\mathbb{P}(E)$ $\quad \mathbb{P}_1(E)$<br>$\mathrm{dom}(E)$ $\quad \mathrm{ran}(E)$<br>$\mathrm{prj}_1(E)$ $\quad \mathrm{prj}_2(E)$<br>$\mathrm{id}(E)$ $\quad \mathrm{union}(E)$ | $\mathsf{WD}(E)$ |
| $\mathrm{card}(E)$ | $\mathsf{WD}(E) \land \mathrm{finite}(E)$ |
| $\mathrm{inter}(E)$ | $\mathsf{WD}(E) \land E \neq \varnothing$ |

Table 5.2: WD lemmas for binary and unary expressions.

| **Expression** | **WD Lemma** |
|---|---|
| $\lambda Q \cdot P \mid E$ | $\forall \mathcal{F}_Q \cdot \mathsf{WD}(P) \land (P \Rightarrow \mathsf{WD}(E))$ |
| $\bigcup L \cdot P \mid E$<br>$\{L \cdot P \mid E\}$ | $\forall L \cdot \mathsf{WD}(P) \land (P \Rightarrow \mathsf{WD}(E))$ |
| $\bigcup E \mid P$<br>$\{E \mid P\}$ | $\forall \mathcal{F}_E \cdot \mathsf{WD}(P) \land (P \Rightarrow \mathsf{WD}(E))$ |
| $\bigcap L \cdot P \mid E$ | $(\forall L \cdot \mathsf{WD}(P) \land (P \Rightarrow \mathsf{WD}(E)))$<br>$\land \quad (\exists L \cdot P)$ |
| $\bigcap E \mid P$ | $(\forall \mathcal{F}_E \cdot \mathsf{WD}(P) \land (P \Rightarrow \mathsf{WD}(E)))$<br>$\land \quad (\exists \mathcal{F}_E \cdot P)$ |
| $\mathrm{bool}(P)$ | $\mathsf{WD}(P)$ |
| $\{E_1, E_2, \ldots, E_n\}$ | $\mathsf{WD}(E_1) \land \mathsf{WD}(E_2) \land \cdots \land \mathsf{WD}(E_n)$ |
| $I$ $\quad$ $\mathbb{Z}$<br>$\mathbb{N}$ $\quad$ $\mathbb{N}_1$<br>BOOL $\quad$ TRUE<br>FALSE $\quad$ $\varnothing$<br>$n$ | $\top$ |

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.