ВВЕДЕНИЕ

Языки и методы формальной спецификации, как средство проектирования и анализа программного обеспечения (ПО) появились более сорока лет назад. За это время было немало попыток разработать как универсальные, так и специализированные языки формальных спецификаций (ЯФС), которые могли бы стать практическим инструментом разработчиков ПО, таким же, как, например, языки программирования. За свою недолгую историю ЯФС уже прошли через несколько периодов подъема и спада. Оценки перспектив ЯФС варьировались от оптимистических прогнозов, предсказывающих появление языков описания требований, по которым будет генерироваться готовое ПО, до пессимистических утверждений, что ЯФС всегда будут только игрушкой для экспериментов в академических исследованиях. Вместе с тем, хотя сейчас даже термин «языки формальных спецификаций» известен далеко не всем программистам, нужно констатировать, что в этой отрасли программирования получены значительные результаты. Они выражаются, во-первых, в том, что есть несколько ЯФС, которые уже нельзя назвать экспериментальными, более того, некоторые ЯФС подкреплены соответствующими стандартами. Во-вторых, более четким стало представление о способах использования ЯФС, о месте формальных методов в жизненном цикле разработки ПО и в процессах разработки и использования ПО.

Второй из перечисленных факторов важен, поскольку способ использования языка, как оказалось, серьезно влияет на эволюцию языка. Если рассматривать историю языков спецификации общего назначения (универсальных, не специализированных), то видно, что они развивались в соответствии со следующим представлением о «правильном порядке» разработке ПО:

- описать эскизную модель (функциональности, поведения);
- доказать, что модель корректна (не противоречива, консистентна);
- детализировать (уточнить) модель;
- доказать, что детализация проведена корректно;
- повторять два предыдущих шага до тех пор, пока не будет получена готовая программа.

Установка на данную схему процесса разработки приводила к акцентированному вниманию на средства обобщенного описания функциональности, средства уточнения, средства аналитического доказательства корректности в стиле традиционных математических доказательств. Как следствие, появились языки, которые не содержали в себе конструкций, затрудняющих аналитические доказательства, при этом они лишались и тех конструкций, без которых трудно описать реализацию сколько-нибудь сложной программной системы.

У специализированных языков несколько другая история. Некоторые из них, например SDL, родились из практики проектирования систем релейного

управления, где проект традиционно был больше похож на чертеж, чем на текстовое (языковое) описание. Здесь эволюция заключалась во взаимном сближении графической и текстовой нотации на основе взаимных компромиссов и ограничений. При этом участниками компромисса была некоторая графическая нотация и языки программирования, опыт, накопленный в других языках формальных спецификаций, либо игнорировался, либо заимствовался с большим опозданием.
ЯФС традиционно рассматривались как средство проектирования. Новый взгляд на ЯФС появился тогда, когда актуальной стала задача анализа уже существующего (legacy) ПО. Существенное продвижение на этом фронте было связано с направлением Объектно-Ориентированного Анализа (ООА). Идеи ООА во многом созвучны с Объектно-Ориентированным Проектированием (ООП). Не удивительно, что оба эти направления предлагают близкие изобразительные средства для описания архитектуры и поведения систем. В последнее время наиболее известным средством такого рода является (преимущественно) графический язык UML. Заметим, что UML и подобные ему языки спецификации, безусловно, являются неплохими средствами проектирования, но обычно непригодны для доказательства правильности, на что делался акцент в классических языках спецификации.
Совершенно новые требования к языкам спецификации появились с идеей использования их как источников для генерации тестов. Оказалось, что разные виды приложений требуют совершенно различных подходов к спецификации и имеют непохожие друг на друга возможности для генерации тестов. В частности, одни виды спецификаций в большей степени пригодны для генерации последовательностей тестовых воздействий (тестовых цепочек), тогда как другие предоставляют удобные возможности для генерации тестовых оракулов – программ, оценивающих результат, полученный в ответ на тестовое воздействие.

Данное методическое пособие посвящено языку формальных спецификаций, разработанному в рамках методологии RAISE (Rigorous Approach to Industrial Software Engineering) [RAISE-method]. Отсюда возникло и название языка – RSL (RAISE Specification Language) [RAISE-language]. Методология и язык были разработаны в рамках европейских программ ESPRIT 1 и ESPRIT 2. Задачей разработчиков методологии и языка было создание, с одной стороны, достаточно универсального средства и, с другой стороны, достаточно компактного и простого, ориентированного в первую очередь не на академических исследователей, а на инженеров. Сейчас, по прошествии почти десяти лет после выхода описания языка, можно сказать, что авторами была проделана большая и полезная работа, они сумели скомпоновать хорошо продуманный язык и вложить в него разнообразные возможности, используемые в других языках, тем не менее, задача-максимум – сделать его инструментом инженеров – все еще не решена. Однако из имеющихся

языков спецификации общего назначения RSL в наибольшей степени приблизился к языку для промышленного использования.

RSL поддерживает различные подходы к спецификации. Его универсальность была достигнута за счет интеграции языковых возможностей уже существующих языков. Полезно знать, из каких «первоисточников» складывался язык и сам RAISE метод.

Имеется несколько способов классификации подходов к спецификации. Например, различают моделе-ориентированные и свойство-ориентированные спецификации или спецификации, основанные на описании состояний (state-based) и описании действий (action-based). Единой классификации не существует, мы будем рассматривать следующие четыре класса подходов к спецификации:

- исполняемые (executable) спецификации,
- алгебраические или ко-алгебраические (co-algebraic),
- сценарные (use cases or scenarios) и
- ограничения (constraints).

Исполнимые спецификации, исполнимые модели (executable specifications, executable models). Этот подход предполагает разработку прототипов (моделей) систем для демонстрации возможности достижения поставленной цели и проведения экспериментов при частичной реализации функциональности. Примерами таких методологий и языков являются SDL [SDL], VDMTools [IFAD], явные определения функций (explicit function definitions) в RSL.

Алгебраические спецификации (algebraic specifications) предполагают описание свойств композиций операций (композиции могут быть последовательными, параллельными, с временными ограничениями и без). Преимуществом этого подхода является то, что в идеале можно полностью абстрагироваться от структур данных, которые используются в качестве входных и выходных значений и, возможно, используются для сохранения внутреннего состояния моделей. Основной недостаток – это нетрадиционность приемов спецификации, что затрудняет их внедрение в промышленных разработках. В качестве примера зрелого (хотя и весьма ограниченного) языка алгебраических спецификаций можно назать ASN1, стандарт которого входит в группу стандартов, описывающих SDL [SDL]. RSL предоставляет достаточно мощные средства для алгебраических спецификаций (аксиомы).

Сценарные (use case/scenario) спецификации описывают не непосредственно целевую систему, а способы ее использования или взаимодействия с ней. Оказывается, что такие косвенные описания, с одной стороны, позволяют

судить о некоторых свойствах системы (тем самым, они, конечно, являются спецификациями), и, с другой стороны, такие спецификации могут служить хорошим руководством по использованию системы, что не всегда можно сказать о других видах спецификаций. Этот подход развивался многими авторами. Наибольшее распространение получили работы OMG группы [OMG] и продукты компании Rational Corporation [RUP]. Сообщество пользователей SDL использует также хорошо известную и стандартизованную нотацию MSC [MSC].

Ограничения (constraints) состоят из пред- и пост-условий функций, процедур и других операций и инвариантов данных. Имеются расширения этого подхода для объектно-ориентироанных (ОО) спецификаций. В этом случае к спецификациям операций добавляются спецификации методов классов, а к инвариантам – инварианты объектов и классов. Классическими языками, поддерживающими спецификацию ограничений, является VDM [Jones] и Z[Z]. RSL предоставляет гибкие возможности для спецификации ограничений в форме имплицитных спецификаций функций и ограничений подтипов. Идее спецификации ограничений следуют авторы, которые предлагают средства для спецификации ОО ПО и средства спецификационного расширения языков программирования: Eiffel [Eiffel], Larch [Larch], VDM [VDM-SL, VDM++], iContract [iContract], ADL [ADL].

В данном методическом пособии собраны справочные материалы по основным возможностям языка RSL. Их можно рассматривать как часть конспекта лекций, который читается на факультете ВМиК МГУ. Здесь описывается подмножество языка, которое не содержит один из видов модулей – объекты. RSL не является языком ОО спецификации, и его объекты совсем не похожи на объекты C++ или Java. Поэтому исключение этого вида модулей не вносит серьезных ограничений в возможности пользователей, при этом изложение основных концепций языка удается сделать более сжатым и понятным.
Следует обратить внимание на то, что в пособие не вошли материалы, описывающие методы использования языка RSL. В курсе лекций методам использования посвящено два блока. Первый – это методы проектирования с использованием формальных спецификаций; второй – это методы тестирования на основе формальных спецификаций. Таким образом, данное пособие предоставляет некоторый фундамент, но не исчерпывает всего материала курса.

1. Вводные замечания
1.1. Структура изложения материала

Предлагаемое пособие содержит справочные материалы по языку RSL, описывающие синтаксис, семантику и правила использования конструкций этого языка. Структура пособия следует структуре основных синтаксических категорий языка RSL, и каждый раздел содержит описание соответствующей синтаксической категории. Раздел состоит из нескольких секций, назначение которых и используемые в них соглашения приведены ниже.

Синтаксис. Для устранения возможной неоднозначности в терминологии названия синтаксических категорий приводятся на русском и английском языках. Синтаксис языка RSL описывается в традиционной форме, при этом принимаются следующие соглашения:

1. Необязательное вхождение **x**, где **x** – имя произвольной синтаксической категории, в металингвистическую формулу задается конструкцией opt- и определяется правилом:

```
opt-x ::=
    nil-x |
    x
```

при этом nil-x означает отсутствие x.

2. Возможные повторения в металингвистических формулах задаются с помощью конструкций -string, -list, -choice, -product и определяются правилами вида:

```
x-string ::=
  x |
  x x-string

x-list ::=
  x |
  x, x-list

x-list2 ::=
  x, x |
  x, x-list2

x-choice ::=
  x |
  x | x-choice      (в последней альтернативе '|' является символом RSL)

x-choice2 ::=
  x | x |
  x | x-choice2  (в каждой из двух альтернатив '|' является символом RSL)

x-product ::=
  x|
  x × x-product
```

x-product2 ::=
  x × x |
  x × x-product2

Если 'opt' появляется совместно с 'string' или 'list', 'opt' имеет наименьший приоритет. Это означает, что для любой синтаксической категории 'x' выполняются следующие правила:

opt-x-string ::=
  nil-x-string |
  x-string

opt-x-list ::=
  nil-x-list |
  x-list

Аналогично, если 'nil' появляется совместно с 'string' или 'list', 'nil' имеет наименьший приоритет.

3. Контекстные условия к произвольной синтаксической категории, входящей в альтернативу, задаются в виде напечатанного курсивом префикса к имени этой категории, например:

axiom_prefix_expr ::=
  • *logical*-value_expr

здесь контекстное условие означает, что максимальный тип соответствующего выражения должен быть **Bool**.

restriction ::=
  • *readonly_logical*-value_expr

здесь контекстное условие означает, что соответствующее выражение должно быть доступно только для чтения и его максимальный тип должен быть **Bool**.

Терминология. Эта секция содержит определения терминов и т.д. Определяемый термин в тексте выделяется курсивом.

Контекстно-независимые расширения. Секция содержит расширения определяемых конструкций. При задании контекстно-независимого расширения какой-либо конструкции в терминах других конструкций ее контекст, правила видимости, контекстные условия, атрибуты и семантика не формулируются отдельно, а задаются соответственно контекстом, правилами видимости и т.д. ее расширения.

Неявно предполагается, что все конструкции, содержащие комментарии (принадлежащие к синтаксической категории comment), имеют контекстно-независимое расширение до соответствующей конструкции, получаемой путем удаления этих комментариев.

Контекст (scope) и правила видимости. Секция содержит описание контекста и правил видимости.

Контекстные условия. Секция содержит описание условий, которым должны удовлетворять синтаксически правильные строки для обеспечения

их статической корректности. При этом *статически корректной* называется синтаксически правильная строка, для которой выполнены контекстные условия. Для удобства некоторые из этих условий в синтаксических правилах также изображаются с помощью напечатанных курсивом префиксов, как описано выше.

Контекстно-зависимые расширения. Секция содержит расширение конструкций. Когда для какой-либо конструкции задается контекстно-зависимое расширение в терминах других конструкций, ее атрибуты и семантика не формулируются отдельно, а задаются атрибутами и семантикой ее расширения. Контекстные условия расширяемой конструкции представляют собой контекстные условия, сформулированные для данной конструкции, плюс контекстные условия ее расширения.

Атрибуты. Секция содержит описание атрибутов для статически корректных строк. Атрибуты используются для описания контекстных условий.

Семантика. Секция содержит описание семантики статически корректных строк языка.

1.2. Декларативные конструкции

Декларативная конструкция является языковой конструкцией, представляющей одно или более определений. *Определение* задает идентификатор или операцию для некоторой сущности, такой как схема, тип, значение, переменная, канал или аксиома.

Для некоторых декларативных конструкций определения, которые они представляют, зависят от контекста, в котором встречаются эти конструкции. Для таких конструкций максимальные типы идентификаторов и (или) операций, заданных содержащимися в данных конструкциях определениями, определяются максимальным типом, заданным контекстом этих конструкций. Такой максимальный тип называют *максимальным контекстным типом* данной декларативной конструкции.

С любым определением связана некоторая область RSL текста, называемая *контекстом* этого определения. Внутри этого контекста и только там для обращения к вводимой данным определением сущности можно использовать ее идентификатор или операцию. Под контекстом декларативной конструкции подразумевается контекст ее определений, который устанавливается *правилами контекста* языка RSL.

Определение является *видимым* в некоторой точке RSL текста, если в данной точке для обращения к заданной с помощью этого определения сущности можно использовать ее идентификатор или операцию. В таком случае говорят, что в данной точке идентификатор или операция *представляют* данную сущность или являются ее *именем*. Видимость определений устанавливается *правилами видимости* языка RSL.

Два определения называются *совместимыми (compatible)*, если они задают различные идентификаторы (операции) или если оба они являются определениями значений (констант и функций), которые задают один и тот же идентификатор (операцию), но их максимальные типы различимы.
Контекстные условия, установленные для каждой индивидуальной синтаксической категории, обеспечивают совместимость всех видимых определений в любой точке RSL текста.

1.3. Правила контекста (Scope Rules)

Контекст декларативной конструкции может зависеть от контекста, в котором она встречается. Поэтому для каждой конструкции, включающей в себя декларативную конструкцию, должен быть задан контекст такого включения. Это описывается в секциях “Контекст и правила видимости” с использованием следующих соглашений:

1. Для декларативной конструкции, встречающейся непосредственно внутри не декларативной конструкции, контекст всегда указывается явно. Например, объявления в локальном выражении:

   ```
   local
     value x : Int = 3
   in x + 2 end
   ```

   Контекстом определения *x* является строка локального объявления и выражение *x + 2*.
2. Для декларативной конструкции, встречающейся непосредственно внутри декларативной конструкции, возможны следующие случаи:
   (a) Контекст указан явно. В этом случае итоговый контекст совпадает с этим явно указанным контекстом.
   (b) Указан непосредственный контекст. В этом случае итоговым контекстом является непосредственный контекст плюс возможные расширения. Расширения зависят от контекста внешней декларативной конструкции и задаются для всех ее вхождений.
   Например, объявления в базисном классовом выражении:

   ```
   scheme S =
     extend
       class
         value x : Int = 3,
       end
     with
       class
         value y : Int = x
       end
   ```

   Непосредственным контекстом определения *x* является строка объявлений первого классового выражения. Итоговым контекстом

x является эта область плюс строка объявлений второго классового выражения.

(c) Контекст не задан. В этом случае неявно подразумевается, что контекст внутренней конструкции задается контекстом непосредственно объемлющей ее внешней конструкции. Например:

```
value
  x : Int = y,
  y : Int
```

Контекст определения значения *x* равен контексту всего раздела объявления значений.

1.4. Правила видимости (Visibility Rules)

Правила видимости языка RSL таковы:

1. Определение не видимо вне своего контекста.
2. Определение потенциально видимо всюду внутри своего контекста. Однако в этом контексте могут существовать участки, где определение является *скрытым*, т.е. не видимым. Например, если идентификатор (операция), заданный некоторым определением, задается также во внутреннем контексте с помощью другого определения, то внешнее определение считается скрытым на протяжении всего контекста этого внутреннего определения. В случае, когда оба такие определения представляют собой определения значений, внешнее определение не скрывается, если максимальные типы этих двух значений различимы.

Например:

```
class
  variable v : Bool := true
  axiom
    local
      variable v : Int := 3
    in v = 7 end
end
```

В этом примере контекстом определения переменной *v : **Bool** := **true*** является все классовое выражение, тогда как контекстом определения локальной переменной *v : **Int** := 3* является только строка локального объявления и выражение *v = 7*. Следовательно, по правилу 2 в выражении *v = 7* видимым является только определение локальной переменной. Действительно, внутреннее определение *v* будет всегда скрывать внешнее внутри своего

контекста, если только оба они не являются определениями значений. Это иллюстрирует следующий пример:

```
class
  value v : Bool = true
  axiom
    local
      value v : Int = 3
    in v end
end
```

Здесь в локальном выражении определение локального значения не скрывает внешнее определение значения, поскольку максимальные типы этих двух определений различны. В данном локальном выражении видимыми будут оба определения значения, фактически же вхождение *v* должно определяться внешним определением, т.к. аксиомы представляют собой булевские выражения.

1.5. Перегрузка имен (Overloading)

Под *перегрузкой* идентификатора или операции в некоторой точке RSL текста понимается ситуация, когда в указанной точке видимыми являются несколько определений этого идентификатора или операции.
Перегрузка имен допускается только для операций и идентификаторов значений (констант и функций).
У всех операций есть одно или более предопределенное значение, контекстом которого является вся спецификация целиком и которое не может скрываться. Следовательно, если пользователь определяет некоторую операцию таким образом, что ее максимальный тип различается с максимальными типами ее предопределенных значений, то эта операция перегружается в тех точках, где видимыми являются как определение пользователя, так и предопределенные значения указанной операции. Пользователю не разрешается определять операцию так, чтобы ее максимальный тип не различался хотя бы с одним из максимальных типов ее предопределенных значений (см. контекстные условия в разделе 11).
При составлении спецификаций необходимо иметь в виду, что для того чтобы спецификация оказалась полезной и могла быть использована в дальнейшем, у каждого ее идентификатора и операции должна существовать единственная допустимая интерпретация. При этом под *интерпретацией* идентификатора (операции) понимается его соответствующее определение. При перегрузке какого-то идентификатора или операции его вхождение имеет несколько возможных интерпретаций (по одной для каждого его видимого определения), и, следовательно, возникает проблема найти допустимое соответствующее определение данного идентификатора или операции (если оно вообще существует).

Некоторые возможные интерпретации рассматриваемого идентификатора или операции могут оказаться недопустимыми, поскольку не будут удовлетворять контекстным условиям его вхождения. Поэтому, чем больше контекста будет принято во внимание, тем больше информации (контекстных условий) будет получено для исключения недопустимых интерпретаций. Однако если рассматриваемый контекст представляет собой выражение, максимальный тип которого будет одним и тем же для нескольких возможных интерпретаций входящих в него идентификаторов или операций, то дальнейший анализ контекста не даст ощутимых результатов, т.к. не позволит отдать предпочтение какой-то одной из имеющихся интерпретаций. Следовательно, все эти интерпретации будут допустимыми.

Более формально для данной конструкции *допустимые интерпретации* (*legal interpretation*) вхождений идентификаторов или операций в точке их использования находятся следующим образом:

1. Если данная конструкция не имеет в своем составе других конструкций, то рассматриваются все комбинации возможных интерпретаций идентификаторов и операций. В противном случае рассматриваются все комбинации интерпретаций, допустимых для составляющих конструкций.
2. Далее удаляются те комбинации, которые не удовлетворяют контекстным условиям данной конструкции.
3. В заключение, если конструкция представляет собой выражение, принадлежащее синтаксической категории value_expr, удаляются те комбинации, при которых максимальные типы данной конструкции не различимы.

Полученные в результате описанного процесса комбинации интерпретаций содержат допустимые для данной конструкции интерпретации вхождений ее идентификаторов и операций.

Перегрузка называется *разрешимой* (*resolvable*), если существует в точности одна допустимая интерпретация каждого идентификатора и операции в его ближайшем объемлющем ‘разрешающем контексте’. При этом под *разрешающим контекстом* понимается одна из следующих ситуаций:

- выражение value_expr в ограничении списка list_limitation;
- выражение value_expr в let-выражении explicit_let;
- выражение value_expr в case-выражении case_expr;
- выражение value_expr в пост-выражении post_expr;
- определяемая операция defined_item, представляющая собой конструкцию id_or_op;
- спецификация specification.

Рассмотрим несколько примеров, иллюстрирующих ситуацию перегрузки идентификаторов и операций.

Пример 1:

```
class
  value
    v : Int,
    v : Bool
  axiom
    v
end
```

В данном случае имеет место перегрузка вхождения *v* в аксиому, т.к. у него есть две возможные интерпретации: либо это целочисленное значение, либо булевское. Однако только последняя интерпретация удовлетворяет контекстным условиям (максимальным типом аксиомы должен быть тип **Bool**) и, следовательно, только эта интерпретация допустима. Таким образом, данная перегрузка является разрешимой, поскольку существует единственная допустимая интерпретация вхождения *v*.

Пример 2:

```
class
  value
    + : Bool × Bool → Bool,
    v : Real
  axiom
    true + false ≡ true,
    v ≡ 1.7 + 2.2
end
```

Здесь перегружаются два вхождения операции + в аксиомы, и каждое вхождение имеет три возможные интерпретации: предопределенное целочисленное сложение (с максимальным типом **Int** × **Int** $\xrightarrow{\sim}$ **Int**), или предопределенное вещественное сложение (с максимальным типом **Real** × **Real** $\xrightarrow{\sim}$ **Real**), или определенное пользователем булевское сложение (с максимальным типом **Bool** × **Bool** $\xrightarrow{\sim}$ **Bool**). Для первого вхождения операции + контекстным условиям удовлетворяет только определенное пользователем сложение, а для второго вхождения — только предопределенное вещественное сложение. Таким образом, для каждого из двух указанных вхождений операции +, есть в точности одна его допустимая интерпретация и, следовательно, данная перегрузка разрешима.

Пример 3:

```
value
  v : Int,
  v : Bool,
  f : Int → Int,
  f : Bool → Nat
axiom
  f(v) ≡ 7
```

В данном случае есть следующие две комбинации интерпретаций для *f* и *v*, удовлетворяющие контекстным условиям:

1. f : **Int** → **Int**, v : **Int**
2. f : **Bool** → **Nat**, v : **Bool**

Однако для обеих комбинаций максимальный тип выражения *f(v)* один и тот же (**Int**) и, следовательно, у данного выражения нет допустимых интерпретаций. Таким образом, данная перегрузка не разрешима.

2. Спецификации (Specifications)

Синтаксис:

```
specification ::=
  module_decl-string

module_decl ::=
  scheme_decl |
  object_decl
```

Терминология. *Модуль* – это либо некоторый объект[1], либо схема.
Контекст и правила видимости. Контекстом входящих в спецификацию объявлений module_decl-string являются сами эти объявления module_decl-string. Это означает, что порядок следования определений не существенен, т.е. схема может быть использована до своего определения.
Контекстные условия. Все объявления модулей module_decl должны быть совместимы.
Семантика. Спецификация употребляется для определения одного или более модулей.

3. Объявления (Declarations)

3.1. Общие положения

Синтаксис:

```
decl ::=
  scheme_decl |
  type_decl |
  value_decl |
  variable_decl |
  channel_decl |
  axiom_decl
```

Терминология. Объявление представляет собой список определений (definitions) одинакового *вида* – схем, типов, значений, переменных, каналов или аксиом. Каждое определение обычно задает некоторый идентификатор

[1] Как уже отмечалось, далее объекты RSL рассматриваться не будут.

или операцию для *сущности* этого вида, следовательно, обычно оно устанавливает одно или более *свойств* такой сущности.
Атрибуты. За исключением определений аксиом с каждым видом определений связано некоторое максимальное определение. В соответствующих разделах для каждого вида определений описывается максимальное определение, специфичное для данного вида.

3.2. Объявление схем (Scheme Declarations)

Синтаксис:

```
scheme_decl ::=
  scheme scheme_def-list

scheme_def ::=
  opt-comment-string id opt-formal_scheme_parameter = class_expr

formal_scheme_parameter ::=
  ( formal_scheme_argument-list )

formal_scheme_argument ::=
  object_def
```

Терминология. *Схема* представляет собой класс или параметризованный класс. *Параметризованный класс* – это отображение списка объектов на классы: каждый объект списка отображается в некоторый класс.
Максимальным классом схемы является максимальный класс соответствующего выражения, задающего класс в определении данной схемы.
Определение схемы scheme_def является *циклическим*, если конструкции opt-formal_array_parameter или class_expr зависят от схемы, которая сама задается этим определением scheme_def.
Конструкция *зависит* от схемы *S*, если она, игнорируя ограничения в выражениях для подтипов, использует *S* или любую схему, содержащую определение, в котором opt-formal_scheme_parameter или class_expr зависят от *S*.
Контекст и правила видимости. В определении схемы scheme_def контекстом необязательного параметра opt-formal_scheme_parameter является сам этот параметр opt-formal_scheme_parameter и выражение class_expr, входящее в данное определение схемы.
Контекстные условия. Все входящие в объявление схемы scheme_decl определения scheme_def должны быть совместимы.
Входящие в формальный параметр схемы formal_scheme_parameter аргументы formal_scheme_argument должны быть совместимы.
Определение схемы scheme_def не должно быть циклическим.
Атрибуты. В определении схемы scheme_def максимальным классом идентификатора id является максимальный класс входящего в это

определение выражения class_expr, и в случае присутствия необязательного параметра formal_scheme_parameter идентификатор id имеет также формальный параметр, которым и является formal_scheme_parameter.
Максимальное определение определения схемы scheme_def получается из исходного определения путем замещения в его составе определений объектов, входящих в formal_scheme_parameter (в случае его присутствия), на соответствующие максимальные определения этих объектов и выражения class_expr на соответствующее выражение для максимального класса.
Семантика. Определение схемы scheme_def задает конкретный идентификатор id для определяемой схемы.

- Определение схемы в форме:

  id = class_expr

  задает конкретный идентификатор *id* для класса, представленного выражением *class_expr*.
- Определение схемы в виде:

  id(formal_scheme_argument-list) = class_expr

  задает идентификатор *id* для параметризованного класса.

3.3. Объявление типов (Type Declarations)

Синтаксис:

```
type_decl ::=
  type type_def-list

type_def ::=
  sort_def |
  variant_def |
  union_def |
  short_record_def |
  abbreviation_def
```

Контекстные условия. Составляющие объявление типов type_decl определения типов type_def должны быть совместимы.

3.3.1. Определение абстрактных типов (Sort Definitions)

Синтаксис:

```
sort_def ::=
  opt-comment-string id
```

Терминология. *Сорт* или *абстрактный тип* – это некоторый тип, не имеющий литералов с предопределенным значением и предопределенных операций за исключением = и ≠ .

Атрибуты. Максимальным типом входящего в определение идентификатора id является тип, представленный этим идентификатором id.
Определение абстрактного типа является максимальным.
Семантика. Определение sort_def задает конкретный идентификатор id для некоторого абстрактного типа.
Поскольку абстрактный тип не обеспечен предопределенными значениями литералов или операций (кроме = и ≠) для генерации и манипулирования со своими значениями, составители спецификаций должны сами определять такие значения. Введенные ими определения могут косвенно устанавливать свойства этого абстрактного типа. Если, например, определены два значения одного и того же абстрактного типа и они специфицированы как различные, то косвенно от данного абстрактного типа требуется, чтобы он содержал по крайней мере два значения.

3.3.2. Определение вариантов (Variant Definitions)

Синтаксис:

```
variant_def ::=
  opt-comment-string id == variant-choice

variant ::=
  constructor |
  record_variant

record_variant ::=
  constructor ( component_kind-list )
component_kind ::=
  opt-destructor type_expr opt-reconstructor

constructor ::=
  id_or_op |
  _

destructor ::=
  id_or_op :

reconstructor ::=
  ↔ id_or_op
```

Контекстно-независимые расширения. Определение варианта variant_def является сокращением (краткой формой) для определения абстрактного типа, его конструкторов, деструкторов и реконструкторов.

3.3.3. Определение объединений (Union Definitions)

Синтаксис:

```
union_def ::=
  opt-comment-string id = name_or_wildcard-choice2
```

```
name_or_wildcard ::=
  type-name |
  _
```

Контекстные условия. Входящие в определение имена name должны представлять типы, причем их максимальные типы должны различаться.

Контекстно-зависимые расширения. Определение union_def вида:

type id = id_1 | … | id_n | _

эквивалентно определению вариантов:

```
type
  id ==
    id_from_id1 ( id_to_id1 : opt-qualification1 id1 ) | … |
    id_from_idn ( id_to_idn : opt-qualificationn idn ) | _
```

которое обеспечивает, чтобы все неявное приведение типов в выражениях и образцах (patterns), использующих функции $id\_from\_id_i$ $(1 \leq i \leq n)$, было замещено фактическим приведением типов, как будет объяснено в разделах 6.1 и 9.1.

Если определение объединения union_def не содержит универсальной альтернативы '_', соответствующее определение вариантов также не будет содержать такой альтернативы.

3.3.4. Определение сокращенной записи (Short Record Definitions)

Синтаксис:

```
short_record_def ::=
  opt-comment-string id :: component_kind-string
```

Контекстно-независимые расширения. Определение short_record_def является краткой формой для определения вариантов с единственным вариантом, включающим конструктор. Определение short_record_def вида:

type id :: $component\_kind_1$ … $component\text{-}kind_n$

представляет собой сокращенную запись определения:

type id == mk_id($component\_kind_1$, … ,$component\text{-}kind_n$)

где *mk_id* задает конструктор данного варианта.

3.3.5. Определение абревиатур (Abbreviation Definitions)

Синтаксис:

```
abbreviation_def ::=
  opt-comment-string id = type_expr
```

Терминология. Определение abbreviation_def является *циклическим*, если максимальный тип задающего его тип выражения type_expr зависит от типа, который сам задается посредством этого же определения abbreviation_def.
Максимальный тип *зависит* от некоторого типа *t*, если он использует этот тип *t*.
Контекстные условия. Определение abbreviation_def не должно быть циклическим.
Атрибуты. Максимальным типом входящего в определение идентификатора id является максимальный тип указанного в этом определении type_expr.
Максимальное определение для определения abbreviation_def получается путем замещения входящего в него типового выражения type_expr соответствующим выражением для максимального типа.
Семантика. Определение abbreviation_def задает конкретное имя id для типа, представленного выражением type_expr.

3.4. Объявление функций (Value Declarations)

В языке RSL константа рассматривается как частный случай функции, поэтому для объявления констант и функций предусмотрен единый раздел value declarations.
Синтаксис.

```
value_decl ::=
  value value_def-list

value_def ::=
  commented_typing |
  explicit_value_def |
  implicit_value_def |
  explicit_function_def |
  implicit_function_def |
```

Контекстные условия. Входящие в объявление value_decl определения value_def должны быть совместимы.

3.4.1. Прокомментированное указание типа (Commented Typing)

См. раздел 8.

3.4.2. Явное определение констант (Explicit Value Definitions)

Синтаксис:

```
explicit_value_def ::=
  opt-comment-string single_typing = pure-value_expr
```

Контекстные условия. Максимальный тип выражения value_expr должен быть меньше или равен максимального типа конструкции single_typing.

Выражение value_expr должно представлять собой чистое выражение (см. раздел 6.1).

Контекстно-зависимые расширения. Явное определение константы explicit_value_def является краткой формой для определения константы и аксиомы.

Пусть метафункция *express* преобразует связывание (binding) в выражение путем взятия в скобки всех операций, оставляя остальную часть связывания без изменения. Семантика операций в скобках описана в разделе 10.3. Тогда явное определение константы explicit_value_def вида:

value binding : type_expr = value_expr

является сокращением следующего фрагмента:

value binding : type_expr
axiom *express*(binding) = value_expr

3.4.3. Неявное определение констант (Implicit Value Definitions)

Синтаксис:

implicit_value_def ::=
  opt-comment-string single_typing *pure*-restriction

Контекстные условия. Ограничение restriction должно представлять собой чистое выражение.

Контекстно-зависимые расширения. Неявное определение константы implicit_value_def является краткой формой для определения константы и аксиомы.

Неявное определение константы implicit_value_def вида:

value binding : type_expr • value_expr

является краткой формой записи следующего фрагмента:

value binding : type_expr
axiom value_expr

3.4.4. Явное определение функций (Explicit Function Definitions)

Синтаксис:

explicit_function_def ::=
  opt-comment-string single_typing
    formal_function_application ≡ value_expr opt-pre_condition

formal_function_application ::=
  id_application |
  prefix_application |
  infix_application

id_application ::=
  value-id formal_function_parameter-string

formal_function_parameter ::=
  (opt-binding-list)

prefix_application ::=
  prefix_op id

infix_application ::=
  id infix_op id

Терминология. *Телом* явно определяемой функции является выражение, указанное в определении этой функции.

Контекст и правила видимости. В определении explicit_function_def контекстом конструкции formal_function_application является выражение value_expr и необязательное предусловие opt-pre_condition.

Контекстные условия. Связывание binding в указании типа single_typing должно быть именем или операцией id_or_op (т.е. декартово произведение product_binding не допускается).

Если метапеременная id_or_op в указании типа single_typing является идентификатором id, то в качестве значения метапеременной formal_function_application должна использоваться конструкция id_application этого идентификатора id. Если id_or_op в single_typing является префиксной операцией prefix_op, то в качестве значения метапеременной formal_function_application должна использоваться конструкция prefix_application этой операции prefix_op. И наконец, если id_or_op в single_typing является инфиксной операцией infix_op, то в качестве значения метапеременной formal_function_application должна использоваться конструкция infix_application этой операции infix_op.

Максимальный тип конструкции single_typing должен быть функциональным типом. Если значением метапеременной formal_function_application является infix_application, то параметрическая часть соответствующего функционального типа должна представлять собой декартово произведение двух типов. Если значением метапеременной formal_function_application является id_application, то соответствующий функциональный тип должен быть сформирован с использованием по крайней мере такого количества типов (каждый тип указывается перед функциональной стрелкой $\tilde{\to}$ или $\to$), сколько формальных параметров formal_function_parameter указано в определении данной функции.

Это означает, что существуют следующие три допустимые формы явного определения функции:

id : type_expr$_1$ $\tilde{\to}$ opt-access_desc-string$_1$ type_expr$_2$
  … $\tilde{\to}$ opt-access_desc-string$_n$ type_expr$_{n+1}$
id(opt-binding-list$_1$)(opt-binding-list$_2$)…(opt-binding-list$_m$) ≡
  value_expr opt-pre_condition

prefix_op : $\text{type\_expr}_1 \tilde{\to}$ opt-access_desc-string type_expr_2
prefix_op id ≡ value_expr opt-pre_condition

infix_op : $\text{type\_expr}_1 \times \text{type\_expr}_2 \tilde{\to}$ opt-access_desc-string type_expr_3
id_1 infix_op id_2 ≡ value_expr opt-pre_condition

При этом в каждой из указанных форм функциональная стрелка $\tilde{\to}$ может быть заменена функциональной стрелкой $\to$, и любое выражение функционального типа может быть заменено именем, представляющим данный тип согласно определению абревиатуры.
В первой из приведенных выше форм максимальный тип выражения *value_expr* должен быть меньше или равен максимальному типу выражения $\textit{type\_expr}_{n+1}$ при m = n или конструкции:

$\text{type\_expr}_{m+1} \ldots \tilde{\to}$ opt-access_desc-string_n type_expr_{n+1}

при m < n.
Во второй форме максимальный тип выражения *value_expr* должен быть меньше или равен максимальному типу выражения $\textit{type\_expr}_2$.
В третьей форме максимальный тип выражения *value_expr* должен быть меньше или равен максимальному типу выражения $\textit{type\_expr}_3$.
Выражение *value_expr* и предусловие *opt-pre_condition* могут только статически обращаться к тем переменным и каналам, которые указаны в описаниях статического доступа $\textit{opt-access\_desc-string}_1$ — $\textit{opt-access\_desc-string}_m$ в первой из вышеприведенных форм и в описаниях статического доступа *opt-access_desc-string* в двух последних формах.
Контекстно-зависимые расширения. Определение функции explicit_function_def является краткой формой для определения значения и аксиомы, причем конкретный вид этой формы зависит от вида конструкции formal_function_application.
Пусть метафункция *maximal* преобразует тип в соответствующий ему максимальный тип, и пусть метафункция *express* определена так же, как в разделе 3.4.2. Тогда если в качестве значения метапеременной formal_function_application используется id_application с единственным формальным параметром formal_function_parameter, то явное определение функции explicit_function_def вида:

value
id : $\text{type\_expr}_1 \times \ldots \times \text{type\_expr}_n \tilde{\to}$ opt-access_desc-string type_expr_{n+1}
$\text{id}(\text{binding}_1,\ldots,\text{binding}_n)$ ≡ value_expr opt-pre_condition

где n ≥ 1, является краткой формой следующих определений:

value
id : $\text{type\_expr}_1 \times \ldots \times \text{type\_expr}_n \tilde{\to}$ opt-access_desc-string type_expr_{n+1}
axiom
∀ binding_1 : *maximal*(type_expr_1),…, binding_n : *maximal*(type_expr_n) •
$\text{id}(\textit{express}(\text{binding}_1),\ldots,\textit{express}(\text{binding}_n))$ ≡ value_expr opt-pre_condition

При этом если список связываний $binding_1,\ldots,binding_n$ пуст, то выражение $type\_expr_1 \times \ldots \times type\_expr_n$ должно быть **Unit** и использовать квантор не нужно.
Заметим, что диапазон значений каждого формального параметра $binding_i$ шире максимального типа соответствующего формального параметра $type\_expr_i$.
Аналогичное расширение имеет место, если функциональную стрелку $\xrightarrow{\sim}$ заменить функциональной стрелкой $\rightarrow$.
Подобным же образом рассматривается случай, когда конструкция id_application содержит более одного формального параметра formal_function_parameter.
Если в качестве значения метапеременной formal_function_application используется prefix_application, то явное определение функции explicit_function_def вида:

value
prefix_op : $type\_expr_1 \xrightarrow{\sim}$ opt-access_desc-string $type\_expr_2$
prefix_op id ≡ value_expr opt-pre_condition

является краткой формой для определений:

value
prefix_op : $type\_expr_1 \xrightarrow{\sim}$ opt-access_desc-string $type\_expr_2$
axiom
∀ id : *maximal*($type\_expr_1$) •
prefix_op id ≡ value_expr opt-pre_condition

В данном случае следует обратить внимание на то, что формальный параметр *id* диапазоном своих значений превосходит максимальный тип соответствующего формального параметра $type\_expr_1$.
Аналогичное расширение справедливо, если в качестве функциональной стрелки используется символ $\rightarrow$.
Если в качестве значения метапеременной formal_function_application используется infix_application, то явное определение функции explicit_function_def вида:

value
infix_op : $type\_expr_1 \times type\_expr_2 \xrightarrow{\sim}$ opt-access_desc-string $type\_expr_3$
id_1 infix_op id_2 ≡ value_expr opt-pre_condition

является краткой формой для следующего фрагмента определений:

value
infix_op : $type\_expr_1 \times type\_expr_2 \xrightarrow{\sim}$ opt-access_desc-string $type\_expr_3$
axiom
∀ id_1 : *maximal*($type\_expr_1$),id_2 : *maximal*($type\_expr_2$)•
id_1 infix_op id_2 ≡ value_expr opt-pre_condition

Здесь также каждый формальный параметр id_i диапазоном своих значений превосходит максимальный тип соответствующего формального параметра $type\_expr_i$.
Аналогичное расширение имеет место для случая функциональной стрелки $\rightarrow$.

3.4.5. Неявное определение функций (Implicit Function Definitions)

Синтаксис:

```
implicit_function_def ::=
  opt-comment-string single_typing formal_function_application
    post_condition opt-pre_condition
```

Контекст и правила видимости. В определении implicit_function_def контекстом конструкции formal_function_application является постусловие post_condition и необязательное предусловие opt-pre_condition.
Контекстные условия. Связывание binding в указании типа single_typing должно быть идентификатором или операцией id_or_op (т.е. декартово произведение product_binding не допускается).
Если метапеременная id_or_op в конструкции single_typing является идентификатором id, то в качестве значения метапеременной formal_function_application должно использоваться id_application этого идентификатора id. Если id_or_op в single_typing является префиксной операцией prefix_op, то в качестве значения метапеременной formal_function_application должно использоваться prefix_application этой операции prefix_op. И наконец, если id_or_op в single_typing является инфиксной операцией infix_op, то значением метапеременной formal_function_application должно быть infix_application этой операции infix_op.
Максимальным типом конструкции single_typing должен быть функциональный тип. Если значением метапеременной formal_function_application является infix_application, то параметрическая часть функционального типа должна представлять собой декартово произведение двух типов. Если значением метапеременной formal_function_application является id_application, то функциональный тип должен быть сформирован с использованием по крайней мере такого количества типов (все эти типы указываются перед функциональной стрелкой $\tilde{\rightarrow}$ или $\rightarrow$), сколько формальных параметров formal_function_parameter указано в определении данной функции.
Это означает, что существуют следующие три допустимые формы неявного определения функции:

```
id : type_expr1 ~→ opt-access_desc-string1 type_expr2
       … ~→ opt-access_desc-stringn type_exprn+1
id(opt-binding-list1)(opt-binding-list2)…(opt-binding-listm)
  post_condition opt-pre_condition                          (m ≤ n)
```

prefix_op : $\text{type\_expr}_1 \xrightarrow{\sim} \text{opt-access\_desc-string type\_expr}_2$
prefix_op id post_condition opt-pre_condition

infix_op : $\text{type\_expr}_1 \times \text{type\_expr}_2 \xrightarrow{\sim} \text{opt-access\_desc-string type\_expr}_3$
id_1 infix_op id_2 post_condition opt-pre_condition

При этом в каждой из указанных форм функциональная стрелка $\xrightarrow{\sim}$ может быть заменена функциональной стрелкой $\rightarrow$, и любое из выраженеий функционального типа может быть заменено именем, которое представляет данный тип согласно определению абревиатуры.

Конструкции post_condition и opt-pre_condition могут только статически обращаться по чтению к тем переменным и каналам, которые указаны в описаниях статического доступа *opt-access_desc-string*$_1$ — *opt-access_desc-string*$_m$ в первой из вышеприведенных форм и в описаниях статического доступа *opt-access_desc-string* в двух последних формах.

Контекстно-зависимые расширения. Неявное определение функции implicit_function_def является краткой формой для определения значения и аксиомы, причем конкретный вид этой формы зависит от вида конструкции formal_function_application.

Пусть метафункции *maximal* и *express* определены так же, как в предыдущем разделе. Тогда если в качестве значения метапеременной formal_function_application используется id_application с единственным формальным параметром formal_function_parameter, то неявное определение функции implicit_function_def вида:

value
$\text{id} : \text{type\_expr}_1 \times \ldots \times \text{type\_expr}_n \xrightarrow{\sim} \text{opt-access\_desc-string type\_expr}_{n+1}$
$\text{id}(\text{binding}_1,\ldots,\text{binding}_n)$ post_condition opt-pre_condition

где $n \geq 1$, является краткой формой следующих определений:

value
$\text{id} : \text{type\_expr}_1 \times \ldots \times \text{type\_expr}_n \xrightarrow{\sim} \text{opt-access\_desc-string type\_expr}_{n+1}$
axiom
$\forall\ \text{binding}_1 : \textit{maximal}(\text{type\_expr}_1),\ldots,\ \text{binding}_n : \textit{maximal}(\text{type\_expr}_n) \bullet$
$\text{id}(\textit{express}(\text{binding}_1),\ldots,\textit{express}(\text{binding}_n))$ post_condition opt-pre_condition

Если список связываний $\text{binding}_1,\ldots,\text{binding}_n$ пуст, то выражение $\text{type\_expr}_1 \times \ldots \times \text{type\_expr}_n$ должно быть **Unit** и использовать квантор не нужно.

Здесь так же, как и в случае явного определения функции диапазон значений каждого формального параметра *binding*$_i$ шире максимального типа соответствующего формального параметра *type_expr*$_i$.

Аналогичное расширение имеет место, когда в качестве функциональной стрелки используется символ $\rightarrow$.

Подобным же образом рассматривается случай, когда конструкция id_application содержит более одного формального параметра formal_function_parameter.

Если в качестве значения метапеременной formal_function_application используется prefix_application, то неявное определение функции implicit_function_def вида:

value
  prefix_op : $\text{type\_expr}_1 \xrightarrow{\sim} \text{opt-access\_desc-string type\_expr}_2$
  prefix_op id post_condition opt-pre_condition

является краткой формой определений:

value
  prefix_op : $\text{type\_expr}_1 \xrightarrow{\sim} \text{opt-access\_desc-string type\_expr}_2$
axiom
  $\forall$ id : *maximal*(type_expr_1) •
    prefix_op id post_condition opt-pre_condition

В данном случае диапазон значений формального параметра *id* шире максимального типа соответствующего формального параметра $type\_expr_1$.
Аналогичное расширение имеет место для функциональной стрелки $\rightarrow$.
Если в качестве значения метапеременной formal_function_application используется infix_application, то неявное определение функции implicit_function_def вида:

value
  $\text{infix\_op} : \text{type\_expr}_1 \times \text{type\_expr}_2 \xrightarrow{\sim} \text{opt-access\_desc-string type\_expr}_3$
  id_1 infix_op id_2 post_condition opt-pre_condition

является краткой формой для определений:

value
  $\text{infix\_op} : \text{type\_expr}_1 \times \text{type\_expr}_2 \xrightarrow{\sim} \text{opt-access\_desc-string type\_expr}_3$
axiom
  $\forall$ id_1 : *maximal*(type_expr_1),id_2 : *maximal*(type_expr_2)•
  id_1 infix_op id_2 post_condition opt-pre_condition

Здесь каждый формальный параметр id_i диапазоном своих значений превосходит максимальный тип соответствующего формального параметра $type\_expr_i$.
Аналогичное расширение справедливо для функциональной стрелки $\rightarrow$.

3.5. Объявление переменных (Variable Declarations)

Синтаксис:

variable_decl ::=
  variable variable_def-list

variable_def ::=
  single_variable_def |
  multiple_variable_def

single_variable_def ::=
  opt-comment-string id : type_expr opt-initialisation

initialisation ::=
  := *pure*-value_expr

multiple_variable_def ::=
  opt-comment-string id-list2 : type_expr

Терминология. *Переменная* – это некая сущность, в которой могут храниться значения некоторого типа. *Присваивание* значения переменной означает запоминание этого значения в указанной переменной. Значение может быть присвоено переменной явно с помощью выражения присваивания (см. раздел 6.20).
Под *состоянием* спецификации понимается присваивание значений всем переменным, определенным в объектах данной спецификации.
Определение переменной single_variable_def является *циклическим*, если максимальный тип выражения type_expr зависит от переменной, которая сама задается этим определением single_variable_def.
Максимальный тип *зависит* от переменной *v* , если он ссылается на *v* или на любую другую переменную или канал, чей максимальный тип зависит от *v*.
Контекстно-независимые расширения. Множественное определение переменных multiple_variable_def есть краткая запись двух или более определений каждой отдельной переменной, т.е. определение переменных multiple_variable_def вида:

variable id_1, … ,id_n : type_expr

является краткой формой для определений:

variable
  id_1 : type_expr,
    ⋮
  id_n : type_expr

Контекстные условия. Все входящие в объявление переменных variable_decl определения переменных variable_def должны быть совместимы.
Определение переменной single_variable_def не должно быть циклическим.
В инициализации initialisation выражение value_expr должно представлять собой чистое выражение, и его максимальный тип должен быть меньше или равен максимальному типу определяемой переменной.
Атрибуты. В определении переменной single_variable_def максимальным типом соответствующего id является максимальный тип выражения type_expr.
Максимальное определение для определения переменной single_variable_def получается путем замены входящего в это определение типового выражения type_expr на соответствующее выражение для максимального типа и удаления инициализации (если она присутствует).
Семантика. Определение переменной single_variable_def задает конкретный идентификатор id для некоторой переменной, в которой могут храниться

значения максимального типа для типа, представленного выражением type_expr. Типом этой переменной является тип, представленный выражением type_expr.
В дополнении к этому может быть задано инициализирующее выражение, значение которого представляет собой начальное значение определяемой переменной.
Если инициализация initialisation не задана, начальным значением переменной считается некоторое произвольным образом выбранное значение внутри ее типа.

3.6. Объявление каналов (Channel Declarations)

Синтаксис:

channel_decl ::=
  channel channel_def-list

channel_def ::=
  single_channel_def |
  multiple_channel_def

single_channel_def ::=
  opt-comment-string id : type_expr

multiple_channel_def ::=
  opt-comment-string id-list2 : type_expr

Терминология. *Канал* – это среда передачи данных, по которой могут взаимодействовать параллельно выполняющиеся выражения.
Для того чтобы два выражения взаимодействовали посредством канала, одно из них должно передавать данные в этот канал (т.е. использовать его для вывода), в то время как другое должно получать из него данные (использовать канал для ввода). Взаимодействие является *синхронизированным*: выражение, передающее данные в канал, осуществляет вывод только в том случае, если принимающее выражение одновременно осуществляет ввод данных из этого канала.
Определение канала single_channel_def является *циклическим*, если максимальный тип выражения type_expr зависит от канала, который сам задается тем же определением single_channel_def.
Максимальный тип *зависит* от канала *c* , если он ссылается на *c* или на любой другой канал или переменную, максимальный тип которой зависит от *c*.
Контекстно-независимые расширения. Множественное определение multiple_channel_def является сокращенной формой для двух или более определений каждого отдельного канала, т.е. определение каналов multiple_variable_def вида:

channel id_1, … ,id_n : type_expr

является краткой формой для определений:

channel
id_1 : type_expr,
⋮
id_n : type_expr

Контекстные условия. Все входящие в объявление каналов channel_decl определения каналов channel_def должны быть совместимы.
Определение канала single_channel_def не должно быть циклическим.
Атрибуты. В определении канала single_channel_def максимальным типом соответствующего id является максимальный тип выражения type_expr.
Максимальное определение для определения канала single_channel_def получается путем замены входящего в это определение типового выражения type_expr на соответствующее выражение для максимального типа.
Семантика. Определение канала single_channel_def задает конкретный идентификатор id для некоторого канала, по которому могут передаваться значения, тип которых представлен выражением type_expr.

3.7. Объявление аксиом (Axiom Declarations)

Синтаксис:

axiom_decl ::=
axiom opt-axiom_quantification axiom_def-list

axiom_quantification ::=
forall typing-list •

axiom_def ::=
opt-comment-string opt-axiom_naming *readonly_logical*-value_expr

axiom_naming ::=
[id]

Контекстно-независимые расширения. Любое объявление аксиом axiom_decl может быть расширено до объявления следующего вида:

axiom
$opt\text{-}axiom\_naming_1$ □ $value\_expr_1$,
⋮
$opt\text{-}axiom\_naming_n$ □ $value\_expr_n$

В случае отсутствия конструкции axiom_quantification объявление аксиом axiom_decl вида:

axiom
$opt\text{-}axiom\_naming_1$ $value\_expr_1$,
⋮
$opt\text{-}axiom\_naming_n$ $value\_expr_n$

является краткой формой объявления:

axiom
  opt-axiom_naming$_1$ □ value_expr$_1$,
  ⋮
  opt-axiom_naming$_n$ □ value_expr$_n$

Конструкция axiom_quantification является сокращенной записью для дистрибутивной квантификации, т.е. объявление axiom_decl вида:

axiom forall typing-list •
  opt-axiom_naming$_1$ value_expr$_1$,
  ⋮
  opt-axiom_naming$_n$ value_expr$_n$

представляет собой краткую форму записи объявления:

axiom
  opt-axiom_naming$_1$ □ ∀ typing-list • value_expr$_1$,
  ⋮
  opt-axiom_naming$_n$ □ ∀ typing-list • value_expr$_n$

Контекстные условия. Все входящие в объявление аксиом axiom_decl определения axiom_def должны быть совместимы.
Входящее в определение аксиомы axiom_def выражение value_expr должно быть доступно только для чтения (read-only), и его максимальный тип должен быть **Bool**.
Семантика. Определение axiom_def устанавливает в терминах некоторой аксиомы (в виде соответствующего булевского выражения value_expr) свойства сущностей, заданных где-то в другом месте спецификации.
Аксиома может быть снабжена именем (id в конструкции axiom_naming).

4. Описание классов (Class Expressions)

4.1. Общие положения

Синтаксис:

class_expr ::=
  basic_class_expr |
  extending_class_expr |
  hiding_class_expr |
  renaming_class_expr |
  scheme_instantiation

Терминология. *Модель* представляет собой некоторую ассоциацию идентификаторов и операций с какими-либо сущностями. Модель *обеспечивает* какой-либо идентификатор или операцию, если она ассоциирует этот идентификатор или операцию с некоторой сущностью.
Модель *удовлетворяет* определению, если она обеспечивает идентификатор или операцию, заданные этим определением, если сущность, ассоциируемая

с данным идентификатором или операцией, является сущностью определяемого вида и, наконец, если свойства, устанавливаемые указанным определением, *сохраняются* в данной модели.
Класс представляет собой набор моделей.
Идентификатор или операция является *недоспецифицированным* (*under-specified*), если в классе существует по крайней мере две модели, в которых данный идентификатор или операция ассоциируется с различными сущностями.
Класс $class\_expr_1$ является *подклассом* класса $class\_expr_2$, если все модели класса $class\_expr_1$ являются моделями класса $class\_expr_2$.
Для задания классов используются классовые выражения (class expressions).
Некоторые классы называются *максимальными*. Классовое выражение представляет *максимальный класс* тогда и только тогда, когда все определения, заданные этим выражением, являются максимальными.
Определение схемы является максимальным, если его классовое выражение является максимальным.
Определение типа является максимальным, если заданный с помощью этого определения идентификатор типа представляет максимальный тип.
Определение функции (value definition) является максимальным, если представляет собой указание типа, в котором типовое выражение является максимальным.
Определение переменной является максимальным, если представляет собой определение одной отдельной переменной без инициализации, типовое выражение которой является максимальным.
Определение канала является максимальным, если представляет собой определение одного отдельного канала, типовое выражение которого является максимальным.
Определение аксиомы не является максимальным.
Атрибуты. С каждым выражением для задания класса ассоциируется некоторый максимальный класс. Конкретный максимальный класс для каждого вида классовых выражений определяется в соответствующем разделе данного пособия.
Класс, представленный некоторым классовым выражением, является подклассом ассоциируемого с данным выражением максимального класса.
Семантика. Классовое выражение class_expr употребляется для обозначения набора определений и представляет класс, состоящий из всех моделей, которые удовлетворяют каждому из этих определений. Каждая модель ассоциирует идентификаторы и операции, определенные в классовом выражении class_expr, с конкретными сущностями. Для каждой разновидности классового выражения в соответствующем разделе пособия будет указано, какие именно определения задает это выражение class_expr.

4.2. Базисные классы (Basic Class Expressions)

Синтаксис:

basic_class_expr ::=
  class opt-decl-string **end**

Контекст и правила видимости. Непосредственным контекстом конструкции opt-decl-string является сама opt-decl-string. Это означает, что порядок следования определений в конструкции opt-decl-string несущественен.
Контекстные условия. Все входящие в данное выражение объявления decl должны быть совместимы.
Атрибуты. Максимальный класс выражения basic_class_expr задается выражением class_expr, состоящим из максимальных определений, полученных из не аксиоматических определений, входящих в состав объявлений указанного выражения basic_class_expr.
Значение. Выражение basic_class_expr употребляется для обозначения определений, появляющихся в объявлениях decls.

4.3. Расширяющие классы (Extending Class Expressions)

Синтаксис:

extending_class_expr ::=
  extend class_expr **with** class_expr

Контекст и правила видимости. Контекст первого классового выражения class_expr расширяется до второго классового выражения class_expr.
Контекстные условия. Выражения class_expr должны быть совместимы.
Атрибуты. Максимальный класс представляется выражением, формируемым из исходного выражения extending_class_expr, путем замены входящих в него выражений class_expr максимальными.
Семантика. Классовое выражение extending_class_expr используется для обозначения тех определений, которые обозначаются выражениями class_expr.

4.4. Определяемые операции (Defined Items)

Синтаксис:

defined_item ::=
  id_or_op |
  disambiguated_item

disambiguated_item ::=
  id_or_op : type_expr

Контекстные условия. В конструкции disambiguated_item метапеременная id_or_op должна представлять некоторую функцию, причем ее максимальный тип должен совпадать с максимальным типом выражения type_expr.

Семантика. Типовое выражение type_expr внутри конструкции disambiguated_item требуется в тех случаях, когда id_or_op за счет перегрузки имен (overloading) представляет несколько значений с различными максимальными типами. В таких случаях выражение type_expr идентифицирует в точности одно из этих значений.

5. Описание типов (Type Expressions)

5.1. Общие положения

Синтаксис:

```
type_expr ::=
  type_literal |
  type-name |
  product_type_expr |
  set_type_expr |
  list_type_expr |
  map_type_expr |
  function_type_expr |
  subtype_type_expr |
  bracketed_type_expr
```

Терминология. *Тип* представляет собой набор значений. Различаются три вида типов:

- *предопределенные типы* представлены встроенными в язык литералами. Такие типы включают, например, целые числа или булевские значения. Подробнее см. раздел 5.2.
- *абстрактные типы* представлены идентификаторами, которые вводятся в определениях абстрактных типов (sort definitions, см. раздел 3.3.1.), вариантов (variant definitions, см. раздел 3.3.2.), объединений (union definitions, см. раздел 3.3.3.) и сокращенных записей (short record definitions, см. раздел 3.3.4.).
- *составные типы* построены из других типов путем применения *типового оператора (type operator)* к одному или более типов.

Будем говорить, что тип является *чистым функциональным* типом, если он может быть представлен некоторым выражением функционального типа, в котором отсутствует строка описания доступа.

Тип t_1 является *подтипом* типа t_2, если все значения, принадлежащие типу t_1, принадлежат также типу t_2 .

Некоторые типы называются *максимальными*. Тип является максимальным, если он может быть представлен максимальным типовым выражением.

Типовое выражение является *максимальным* тогда и только тогда, когда оно является одним из следующих:

- **Unit**, **Bool**, **Int**, **Real** или **Char** (т.е. любым типом, состоящим из литералов, за исключением **Nat** и **Text**).
- именем, соответствующее определение которого представляет собой определение абстрактного типа, варианта, объединения или сокращенной записи.
- именем, соответствующее определение которого является определением аббревиатуры, в состав которого входит максимальное типовое выражение.
- типовым выражением, построенным из максимальных типовых выражений и описаний статического доступа путем применения одного из типовых операторов $\times$, **-infset**, $^{\omega}$, $\xrightarrow{\sim}$ или $\xrightarrow[\mathbf{m}]{}$ (т.е. любых типовых операторов за исключением **-set**, * и $\rightarrow$).
- выражением для задания подтипа, если входящее в его состав типовое выражение является максимальным и ограничение, с помощью которого задается подтип, сохраняется для всех значений этого (максимального) типа, представленного данным максимальным типовым выражением.
- заключенным в скобки типовым выражением, где указанное в скобках типовое выражение является максимальным.

Два максимальных типа считаются *неразличимыми* тогда и только тогда, когда существуют два типовых выражения, представляющие эти типы и различающиеся не больше, чем описаниями статического доступа содержащихся в них функциональных типов.

Два максимальных типа считаются *различимыми* тогда и только тогда, когда они не являются неразличимыми.

Использование в спецификации определений объединений (union definitions) вызывает потенциальное приведение типов, которое является частичной функцией между максимальными типами.

Будем говорить, что имеет место *потенциальное приведение (potential coercion)* максимального типа t_1 к максимальному типу t_2 тогда и только тогда, когда выполняются следующие условия:

1. t_1 идентичен с t_2, в этом случае потенциальное приведение типа t_1 к t_2 представляет собой тождественную функцию.
2. задано определение объединения вида:

   id_2 = … | id_1 | …

   в котором id_1 имеет максимальный тип t_1 и id_2 имеет максимальный тип t_2, в этом случае потенциальное приведение типа t_1 к типу t_2 представляет собой функцию $id_2\_from\_id_1$.
3. заданы типы t_{11}, t_{12}, t_{21}, t_{22} и описание статического доступа opt-access_desc-string такие, что имеется потенциальное приведение

типов t_{11} к t_{21} и t_{12} к t_{22}, кроме того, типы t_1 и t_2 построены одним из перечисленных ниже способов. В этом случае потенциальное приведение типа t_1 к типу t_2 выводится из других потенциальных приведений очевидным образом.

(a) t_1 есть $\ldots \times t_{11} \times \ldots$ и t_2 есть $\ldots \times t_{21} \times \ldots$
(b) t_1 есть t_{11}–***infset*** и t_2 есть t_{21}***-infset***.
(c) t_1 есть t_{11}^{ω} и t_2 есть t_{21}^{ω}.
(d) t_1 есть $t_{11} \xrightarrow[\mathbf{m}]{} t_{12}$ и t_2 есть $t_{21} \xrightarrow[\mathbf{m}]{} t_{22}$.
(e) t_1 есть $t_{11} \xrightarrow{\sim}$ *opt-access_desc-string* t_{12} и t_2 есть $t_{21} \xrightarrow{\sim}$ *opt-access_desc-string* t_{22}.

4. задан тип t_3 такой, что f_{13} является потенциальным приведением типа t_1 к t_3 и f_{32} соответствующим приведением t_3 к t_2; в этом случае потенциальным приведением типа t_1 к t_2 является $f_{32} \circ f_{13}$.

Максимальный тип t_1 называется *приводимым* к максимальному типу t_2, если существует одно и только одно потенциальное приведение типа t_1 к t_2.
Максимальный тип t_1 *меньше или равен* максимального типа t_2, если он приводим к подтипу типа t_2.
Максимальный тип t называется *верхней границей* набора максимальных типов, если все максимальные типы данного набора меньше или равны t.
Максимальный тип называется *наименьшей верхней границей* набора максимальных типов, если он является верхней границей данного набора и меньше или равен всех остальных верхних границ этого набора.
Атрибуты. С любым типовым выражением type_expr ассоциируется некоторый максимальный тип. Тип, представленный типовым выражением type_expr, является подтипом ассоциируемого с ним максимального типа. В ниже следующих разделах для каждого конкретного случая типового выражения определяется соответствующий максимальный тип.
Семантика. Типовое выражение type_expr представляет некоторый тип. Конкретные типы для каждой разновидности типовых выражений определяются в следующих разделах.

5.2. Предопределенные типы (Type Literals)

Синтаксис.

type_literal ::=
Unit |
Bool |
Int |
Nat |
Real |
Text |
Char

Атрибуты. Максимальным типом **Unit** является **Unit**.
Максимальным типом **Bool** является **Bool**.
Максимальным типом **Int** является **Int**.
Максимальным типом **Nat** является **Int**.
Максимальным типом **Real** является **Real**.
Максимальным типом **Text** является **Char**$^{\omega}$.
Максимальным типом **Char** является **Char**.
Семантика. Тип type_literal представляет предопределенный тип.
Тип **Unit** имеет единственное значение ().
Значениями типа **Bool** являются булевские константы **true** и **false**.
Значениями типа **Int** являются целые числа (…, -2, -1, 0, 1, 2, …).
Литерал **Nat** является сокращенной записью выражения, задающего следующий подтип {| i : **Int** • i ≥ 0 |}.
Значениями типа **Real** являются вещественные числа (…, -4.3, …, 12.23, …).
Значениями типа **Char** являются ASCII символы в одинарных кавычках ('a', 'b', …).
Литерал **Text** является сокращенной записью типового выражения **Char***.

5.3. Имена (Names)

Контекстные условия. В типовом выражении type_expr, являющимся именем name, это имя name должно представлять некоторый тип.
Атрибуты и семантика описаны в разделе 10.

5.4. Декартово произведение типов (Product type expressions)

Синтаксис.

```
product_type_expr ::=
  type_expr-product2
```

Терминология. *Декартово произведение* – это значение вида $(v_1,...,v_n)$.
Длиной декартова поризведения типов product_type_expr называется количество составляющих его типовых выражений type_expr.
Атрибуты. Максимальным типом выражения product_type_expr вида $type\_expr_1 \times ... \times type\_expr_n$ является декартово произведение $t_1 \times \ldots \times t_n$, где t_1, …, t_n являются максимальными типами соответственно выражений $type\_expr_1, ..., type\_expr_n$.
Семантика. Декартово произведение типов product_type_expr вида $type\_expr_1 \times ... \times type\_expr_n$ представляет тип всех декартовых произведений вида $(v_1,...,v_n)$, где тип каждого v_i представлен выражением $type\_expr_i$.

5.5. Множественные типы (Set type expressions)

Синтаксис.

set_type_expr ::=
  finite_set_type_expr |
  infinite_set_type_expr

finite_set_type_expr ::=
  type_expr-**set**

infinite_set_type_expr ::=
  type_expr-**infset**

Терминология. *Множество* – это, возможно, пустой неупорядоченный набор различных значений одного и того же типа.
Атрибуты. Максимальным типом выражения set_type_expr вида *type_expr*-**set** или *type_expr*-**infset** является *t*-**infset** , где *t* представляет собой максимальный тип выражения *type_expr*.
Семантика. Множественный тип, определяемый типовым выражением set_type_expr, представляет тип из подмножеств множества значений, тип которых представлен входящим в это выражение type_expr. Если в качестве типового оператора используется **-set**, то данный тип содержит все конечные подмножества. Если типовым оператором является **-infset**, то соответствующий множественный тип содержит все (конечные и бесконечные) подмножества.
Множество характеризуется своими элементами.

5.6. Типы списки (List type expressions)

Синтаксис.

list_type_expr ::=
  finite_list_type_expr |
  infinite_list_type_expr

finite_list_type_expr ::=
  type_expr*

infinite_list_type_expr ::=
  type_expr$^{\omega}$

Терминология. *Список* – это, возможно, пустая последовательность значений одного и того же типа, допускающая повторения элементов.
Атрибуты. Максимальным типом выражения list_type_expr вида *type_expr** или *type_expr*$^{\omega}$ является t^{ω}, где *t* представляет собой максимальный тип выражения *type_expr*.
Семантика. Выражение list_type_expr представляет тип списков значений, тип которых определяется входящим в это выражение type_expr. Если в

качестве типового оператора используется *, то данный тип содержит все конечные списки. Если типовым оператором является $^\omega$, то соответствующий тип содержит все (конечные и бесконечные) списки.
Список может быть применен к некоторому значению из множества индексов этого списка для определения соответствующего элемента данного списка.
Список характеризуется множеством индексов и эффектом своего применения к элементам данного множества индексов.

5.7. Типы отображения (Map type expressions)

Синтаксис.

map_type_expr ::=
  type_expr $\xrightarrow[\mathbf{m}]{}$ type_expr

Терминология. *Отображение* можно рассматривать как (возможно, пустой) набор пар (v_1, v_2), где v_1 - это некоторое значение из домена отображения, v_2 - некоторое значение из области значений отображения и v_1 отображается в v_2. *Доменом* (областью определения) отображения называется множество значений v_1, для каждого из которых существует некоторое значение v_2, такое что пара (v_1, v_2) входит в данное отображение. *Областью значений* отображения является множество значений v_2, для каждого из которых существует некоторое значение v_1, такое что пара (v_1, v_2) входит в данное отображение.
Атрибуты. Максимальным типом выражения map_type_expr вида *type_expr*$_1$ $\xrightarrow[\mathbf{m}]{}$ *type_expr*$_2$ является $t_1 \xrightarrow[\mathbf{m}]{} t_2$, где t_1 и t_2 представляют собой максимальные типы выражений *type_expr*$_1$ и *type_expr*$_2$ соответственно.
Семантика. Типовое выражение map_type_expr представляет тип всех отображений, доменом и областью значений каждого из которых являются соответственно некоторое подмножество множества значений типа, представленного первым выражением type_expr, и некоторое подмножество множества значений, тип которых представлен вторым выражением type_expr.
Отображение может быть применено к любому значению из его домена с целью определения соответствующего значения в области значений данного отображения.
Отображение характеризуется доменом и эффектом своего применения к элементам данного домена.

5.8. Функциональные типы (Function type expressions)

Синтаксис.

function_type_expr ::=
  type_expr function_arrow result_desc

function_arrow ::=
  $\tilde{\rightarrow}$ |
  $\rightarrow$

result_desc ::=
  opt-access_desc-string type_expr

Терминология. Выражение для задания функционального типа состоит из двух основных частей: параметрической части (где описывается тип параметра) и результирующей части (где описывается тип результата и статический доступ). *Функция применяется* в некотором состоянии к значению одного какого-либо типа (*тип параметра*), после чего она может:

- вернуть значение другого типа (*тип результата*);
- осуществить доступ к переменным посредством чтения из них или записи в них;
- осуществить доступ к каналам посредством ввода или вывода по этим каналам.

Атрибуты. Максимальным типом выражения function_type_expr вида:

$\text{type\_expr}_1 \tilde{\rightarrow}$ opt-access_desc-string type_expr_2

или:

$\text{type\_expr}_1 \rightarrow$ opt-access_desc-string type_expr_2

является $t_1 \tilde{\rightarrow}$ *oads* t_2, где t_1 и t_2 представляют собой максимальные типы выражений $\textit{type\_expr}_1$ и $\textit{type\_expr}_2$ соответственно и *oads* задает описание статического доступа *opt-access_desc-string*.

Семантика. Выражение function_type_expr задает тип функций, переводящих тип параметра, представленный выражением type_expr, в тип результата, представленный выражением type_expr конструкции result_desc. Описания доступа access_desc конструкции result_desc специфицируют, к каким переменным и каналам может осуществляться доступ при применении функций. В зависимости от значения метапеременной function_arrow функции являются частично вычислимыми или всюду вычислимыми, как описывается ниже.

- *Частично вычислимые (partial) функции*
  Выражение function_type_expr вида:

  $\text{type\_expr}_1 \tilde{\rightarrow}$ opt-access_desc-string type_expr_2

  определяет тип частичных функций из типа, представленного первым выражением *type_expr*, в тип, представленный вторым выражением

type_expr. Этот тип содержит функции, удовлетворяющие следующему ограничению: для любой такой функции *f* и для любого *x*, принадлежащего максимальному типу первого выражения *type_expr*, эффект от применения функционального выражения *f(x)* в любом состоянии таков, что:

- осуществляется только доступ к переменным и каналам, описанным в конструкции *opt-access_desc-string*;
- если вычисление выражения завершается (возможно, после выполнения ряда взаимодействий), то результирующее значение принадлежит максимальному типу второго выражения *type_expr*;
- если вычисление выражения представляет сходящийся процесс (is convergent) и *x* принадлежит типу, представленному первым выражением *type_expr*, то результирующее значение принадлежит типу, представленному вторым выражением *type_expr*.

- *Всюду вычислимые (total) функции*
  Выражение function_type_expr вида:

  type_expr $\rightarrow$ result_desc

  представляет собой краткую запись выражения:

  $\{\mid f : type\_expr \rightarrow result\_desc \bullet \forall x : type\_expr \bullet f(x)$ **post true** $\mid\}$

  которое обеспечивает, что *f* и *x* не являются свободными переменными выражений *type_expr* и *result_desc*.
  Это означает, что выражение function_type_expr вида:

  $type\_expr_1 \rightarrow$ opt-access_desc-string $type\_expr_2$

  определяет тип всюду определенных функций из типа, представленного первым выражением *type_expr*, в тип, представленный вторым выражением *type_expr*. Этот тип содержит те функции, принадлежащие типу:

  $type\_expr_1 \rightarrow$ opt-access_desc-string $type\_expr_2$

  которые удовлетворяют следующему ограничению: для любой такой функции *f* и для любого *x*, принадлежащего типу, представленному первым выражением *type_expr*, эффект от применения функционального выражения *f(x)* в любом состоянии представляет собой сходящийся процесс.

5.9. Подтипы (Subtype Expressions)

Синтаксис.

subtype_expr ::=
{| single_typing *pure*-restriction |}

Контекст и правила видимости. Контекстом указания типа single_typing является ограничение restriction.
Контекстные условия. Ограничение restriction должно быть представлять собой чистое выражение.
Атрибуты. Максимальным типом выражения subtype_expr является максимальный тип входящего в это выражение указания типа single_typing.
Семантика. Выражение subtype_expr представляет подтип типа, заданного указанием типа single_typing. Данный подтип содержит любое значение, сохраняющее истинность ограничения restriction во всех состояниях, при этом вычисление ограничения производится в контексте определений, получаемых путем сопоставления указанного значения со связыванием, также представленным в указании типа single_typing.

5.10. Типовое выражение в скобках (Bracketed Type Expressions)

Синтаксис.

```
bracketed_type_expr ::=
  ( type_expr )
```

Атрибуты. Максимальным типом выражения bracketed_type_expr является максимальный тип входящего в него выражения type_expr.
Семантика. Выражение bracketed_type_expr представляет тот же тип, что и типовое выражение type_expr.

5.11. Описания доступа (Access Descriptions)

Синтаксис.

```
access_desc ::=
  access_mode access-list

access_mode ::=
  read |
  write |
  in |
  out

access ::=
  variable_or_channel-name |
  enumerated_access |
  completed_access |
  comprehended_access

enumerated_access ::=
  { opt-access-list }

completed_access ::=
  any
```

comprehended_access ::=
{ access | *pure*-set_limitation }

Терминология. *Описание статического доступа по чтению* представляет собой некоторое множество переменных. *Описание статического доступа по записи* также представляет собой некоторое множество переменных. *Описание статического доступа по вводу* так же, как *описание статического доступа по выводу* представляет собой некоторое множество каналов.

Статический доступ – это множество переменных или множество каналов.

Контекст и правила видимости. В конструкции comprehended_access контекст ограничения set_limitation расширяется до конструкции access.

Контекстные условия. Если значением метапеременной access является имя name, это имя должно представлять:

- переменную, если оно появляется в списке access-list описания access_desc со значениями **read** или **write** в качестве режима доступа access_mode;
- канал, если оно появляется в списке access-list описания access_desc со значениями **in** или **out** в качестве режима доступа access_mode.

Входящее в конструкцию comprehended_access ограничение set_limitation должно представлять собой чистое выражение.

Атрибуты. С описаниями opt-access_desc-string и access_desc связаны четыре описания статического доступа: описание статического доступа по чтению, описание статического доступа по записи, описание статического доступа по вводу и описание статического доступа по выводу.

Для пустой строки описания nil-access_desc-string описание статического доступа по чтению, записи, вводу и выводу являются пустыми.

Для строки access_desc-string описание статического доступа по чтению представляет собой объединение описаний статического доступа по чтению всех входящих в данную строку описаний access_desc. Аналогично для описаний доступа по записи, вводу и выводу.

Для описания access_desc с режимом доступа **read** описание статического доступа по чтению равно объединению статических доступов его составляющих access, тогда как его описания статического доступа по записи, вводу и выводу пусты. Аналогично для описания access_desc с режимом доступа **in** или **out**. Для описания access_desc с режимом доступа **write** описания статического доступа по чтению и по записи оба равны объединению статических доступов его составляющих access, тогда как его описания статического доступа по вводу и выводу пусты.

С каждой конструкцией access ассоциируется некоторый статический доступ.

Для имени name, представляющего собой некоторый идентификатор, статическим доступом является множество, которое содержит в качестве

своего единственного элемента переменную или канал, представленную данным идентификатором.
Статическим доступом перечисления enumerated_access является пустое множество в случае отсутствия составляющих его элементов access, в противном случае статический доступ перечисления представляет собой объединение статических доступов входящих в него конструкций access.
Статическим доступом конструкции completed_access вида **any** является множество всех переменных или каналов, которые определены в ближайшем объемлющем данную конструкцию классовом выражении или которые могли быть определены в любом расширении этого классового выражения.
Статическим доступом конструкции comprehended_access является статический доступ входящей в нее конструкции access.
Семантика. Описания доступа access_desc в выражении функционального типа ограничивают множество функций, представленное данным выражением, путем установления, как и к каким переменным и каналам может осуществляться доступ.
Описания доступа opt-access_desc-string и access_desc представляют четыре множества переменных и каналов:

- множество переменных с режимом доступа access_mode равным **read**, из которых может осуществляться чтение, — множество чтения;
- множество переменных с режимом доступа access_mode равным **write**, в которые может осуществляться запись (т.е. они могут изменяться за счет присваивания), — множество записи; переменные с режимом доступа **write** автоматически имеют режим доступа **read**;
- множество каналов с режимом доступа access_mode равным **in**, по которым может осуществляться ввод, — множество ввода;
- множество каналов с режимом доступа access_mode равным **out**, по которым может осуществляться вывод, — множество вывода.

Для пустой строки описания nil-access_desc-string множества чтения, записи, ввода и вывода являются пустыми.
Для строки access_desc-string множество чтения представляет собой объединение множеств чтения всех составляющих ее описаний access_desc. Аналогичное утверждение справедливо для множеств записи, ввода и вывода.
Для описания access_desc с режимом доступа **read** множество чтения равно объединению множеств всех составляющих его конструкций access, тогда как его множества записи, ввода и вывода пусты. Аналогично для описания access_desc с режимом доступа **in** или **out**.
Для описания access_desc с режимом доступа **write** множество чтения и множество записи равны объединению множеств всех составляющих его конструкций access, тогда как его множества ввода и вывода пусты.
Конструкция access представляет некоторое множество переменных или каналов.

Множество, представленное именем name, содержит в качестве своего единственного элемента переменную или канал с данным именем.
Множество, представленное перечислением enumerated_access, является пустым в случае отсутствия составляющих это перечисление элементов access, в противном случае оно представляет собой объединение множеств составляющих данное перечисление конструкций access.
Множеством, представленным конструкцией completed_access вида **any**, является множество всех переменных или каналов, которые определены в ближайшем объемлющем данную конструкцию классовом выражении или которые могли быть определены в любом расширении этого классового выражения.
Конструкция comprehended_access представляет собой следующее множество. Для каждой модели из множества моделей, заданного ограничением set_limitation, составляющая access представляет некоторое конкретное множество. Объединение всех таких множеств и образует искомое множество.

6. Value Expressions (выражения)

6.1. Общие замечания

Синтаксис.

value_expr ::=
value_literal |
value_or_variable-name |
pre_name |
basic_expr |
product_expr |
set_expr |
list_expr |
map_expr |
function_expr |
application_expr |
quantified_expr |
equivalence_expr |
post_expr |
disambiguation_expr |
bracketed_expr |
infix_expr |
prefix_expr |
comprehended_expr |
initialise_expr |
assignment_expr |
input_expr |
output_expr |
structured_expr

Терминология. Выражение value_expr *вычисляется* (*выполняется*) в контексте некоторого набора определений и в некотором состоянии. Иногда используется терминология, что выражение value_expr вычисляется ‘в некоторой модели’ (имея в виду модель, удовлетворяющую заданным определениям) и в некотором состоянии. Часто говорят, что выражение value_expr вычисляется, не упоминая при этом какого-либо конкретного набора определений или какого-либо конкретного состояния. Однако такие сокращения могут быть использованы только в тех ситуациях, когда соответствующий набор определений или состояние однозначным образом задаются контекстом изложения, и, следовательно, подобные сокращения не могут привести к путанице.

Результирующее воздействие (effect) выражения value_expr на некоторое состояние заключается в вычислении этого выражения в данном состоянии, после чего оно может:

- вернуть некоторое значение;
- произвести доступ к переменным путем чтения или записи;
- предложить произвести доступ к каналам для ввода или вывода.

Более формально результирующее воздействие выражения value_expr на некоторое состояние состоит в том, чтобы выполнить одно из следующих действий:

1. возможно, после изменения данного состояния *завершиться* возвратом некоторого значения;
2. возможно, после изменения данного состояния *предложить осуществить взаимодействие* путем ожидания ввода из канала или вывода в канал и продолжиться с дальнейшим результирующим воздействием;
3. разрешить внешний выбор между эффектами, подпадающими под категории 1 или 2;
4. *попасть в тупиковую ситуацию* путем останова (в таком случае любой внешний выбор между указанным эффектом и некоторым другим сводится к этому самому другому эффекту);
5. разрешить внутренний выбор между эффектами, подпадающими под категории 3 или 4;
6. *разойтись* путем продолжения вычисления без завершения, без предложения осуществить взаимодействие и без тупиковой ситуации (в таком случае любой внешний или внутренний выбор между данным результирующим воздействием и каким-либо другим также расходится).

Результирующее воздействие является *недетерминированным*, если оно допускает внутренний выбор между другими результирующими воздействиями. В подобных обстоятельствах, в частности, может существовать более одного значения, которое может вернуть выражение, и

более одного изменения состояния, которое может достигаться при его вычислении.

Будем говорить, что результирующее воздействие выражения value_expr на некоторое состояние *сходится*, если выполнены следующие условия:

- это результирующее воздействие не разрешает внутренний выбор между каким-либо результирующим воздействием и тупиковой ситуацией (т.е., в частности, оно не может расходиться или быть тупиком);
- это результирующее воздействие не разрешает внутренний выбор между каким-либо результирующим воздействием и неким другим результирующим воздействием, завершающимся без предложения взаимодействия (поэтому, в частности, если оно может завершиться, то оно завершается, и при этом существует единственное возвращаемое значение и достигается единственное изменение состояния).

Если в выражение value_expr входит несколько составляющих его выражений value_expr, то должен быть установлен порядок их вычисления; обычно они вычисляются слева направо. Порядок вычислений имеет значение в тех случаях, когда составляющие выражения value_expr производят запись в переменные, взаимодействуют по каналам или попадают в тупиковую ситуацию.

Для выражений с описанным доступом используется следующая терминология. Говорят, что выражение value_expr *статически производит чтение (из)* переменных своего описания статического доступа по чтению, *статически производит запись в* переменные своего описания статического доступа по записи, *статически производит ввод из* каналов своего описания статического доступа по вводу и *статически производит вывод в* каналы своего описания статического доступа по выводу.

Говорят, что выражение value_expr *статически производит доступ к переменной (статически обращается к переменной)*, если оно статически производит чтение или запись в эту переменную. Выражение value_expr *статически производит доступ к каналу (статически обращается к каналу)*, если оно статически производит ввод или вывод по этому каналу.

Выражение value_expr называется *чистым*, если оно не производит статически доступа к какой-либо переменной или каналу.

Выражение value_expr называется *read-only (доступным только для чтения)*, если оно не производит статически запись в какую-либо переменную и не производит статически доступ к какому-либо каналу.

Предположим, имеется однозначное потенциальное приведение (см. раздел 5.1.) типа t_1 к типу t_2, т.е. тип t_1 приводим к типу t_2. Тогда, если некоторое выражение имеет максимальный тип t_1 и встречается в контексте, требующим чтобы оно имело максимальный тип t, подтипом которого является тип t_2, говорят, что имеет место *неявное приведение* типа t_1 к типу t_2, представляющее собой применение указанного потенциального приведения (из t_1 к t_2) к данному выражению. Заметим, что в данных обстоятельствах t_1

меньше или равен t, поэтому неявное приведение типов (которое может представлять собой применение тождественной функции) имеет место всякий раз, когда контекстное условие требует, чтобы максимальный тип выражения был меньше или равен другого типа. Подобное неявное приведение типа имеет место и для образцов (см. раздел 9.1.).
Аналогично, если:

- максимальный тип t_1 приводим к максимальному типу t_2,
- максимальный тип t_3 приводим к максимальному типу t_4,
- t_1 и t_3 не имеют наименьшей верхней границы,
- t_2 и t_4 имеют наименьшую верхнюю границу t,
- два выражения типов t_1 и t_3 соответственно встречаются в контексте, который требует, чтобы они имели наименьшую верхнюю границу.

Тогда говорят, что имеют место неявные приведения типов указанных двух выражений t_1 к t_2 и t_3 к t_4 соответственно. Эта формулировка может быть обобщена для набора произвольного количества выражений.
Контекстные условия. Неявные приведения типов должны быть однозначными.
Контекстно-зависимые расширения. Выражение, для которого существует неявное приведение типов, не являющееся тождественной функцией, представляет собой краткую запись выражения, полученного путем применения данного приведения типов.
Атрибуты. С выражением value_expr ассоциируется некоторый максимальный тип (такой, что если выражение value_expr завершается, то его значение принадлежит этому максимальному типу). С выражением value_expr связаны также четыре описания статического доступа: описание статического доступа по чтению, описание статического доступа по записи, описание статического доступа по вводу и описание статического доступа по выводу. Эти описания статического доступа таковы, что выражение value_expr статически производит чтение некоторой переменной, если результирующее воздействие данного выражения заключается в потенциальном чтении этой переменной, аналогично для записи, ввода и вывода.
Для каждой конкретной разновидности выражения value_expr ниже определены соответствующие максимальные типы и описания статического доступа.

6.2. Литералы (Value Literals)

Синтаксис.

value_literal ::=
  unit_literal |
  bool_literal |
  int_literal |
  real_literal |
  text_literal |
  char_literal

unit_literal ::=
  ()

bool_literal ::=
  true |
  false

Значением метапеременной int_literal является любая непустая строка цифр. Значением метапеременной real_literal являются две непустые строки цифр, разделенные десятичной точкой. Метапеременная text_literal представляет собой текст (возможно, пустой) в двойных кавычках. Значением метапеременной char_literal является любой допустимый в RSL символ, взятый в апострофы.

Атрибуты. Максимальным типом unit_literal является **Unit**.
Максимальным типом bool_literal является **Bool**.
Максимальным типом int_literal является **Int**.
Максимальным типом real_literal является **Real**.
Максимальным типом text_literal является **Char**$^{\omega}$.
Максимальным типом char_literal является **Char**.
Выражение value_literal не производит статически доступа к каким-либо переменным или каналам.

Семантика. Результирующее воздействие выражения value_literal заключается в возвращении значения, представленного указанным литералом.

Текстовый литерал вида "c_1 … c_n" представляет собой сокращенную запись выражения $\langle$'c_1', …, 'c_n'$\rangle$.

6.3. Имена

Контекстные условия. Для выражения value_expr, являющегося именем name, это имя должно представлять какое-либо значение или переменную.

Атрибуты. Максимальный тип имени name определен в разделе 10.

Имя name, представляющее некоторое значение, не производит статически доступа к каким-либо переменным или каналам. Имя name, представляющее

переменную, статически производит чтение из этой переменной и не имеет другого статического доступа.
Семантика. См. раздел 10.

6.4. Пред-имена (Pre-names)

Синтаксис.

pre_name ::=
  variable-name`

Контекст и правила видимости. Если в ближайшем объемлющем постусловии присутствуют какие-либо определения локальных переменных, они не видимы в данном имени name.
Контекстные условия. Выражение pre_name должно появляться внутри постусловия.
Имя name должно представлять некоторую переменную.
Атрибуты. Максимальным типом имени pre_name является максимальный тип входящего в это выражение имени name.
Выражение pre_name статически производит чтение переменной, которое оно представляет, и не имеет другого статического доступа.
Семантика. Выражение pre_name встречается внутри постусловий (см. раздел 6.13.). Результирующее воздействие выражения pre_name состоит в том, чтобы вернуть содержание переменной, представленной именем name, в пред-состоянии (pre-state).

6.5. Базисные выражения (Basic Expressions)

Синтаксис.

basic_expr ::=
  chaos |
  skip |
  stop |
  swap

Атрибуты. Максимальным типом выражения **skip** является **Unit**. Максимальным типом выражений **chaos**, **stop** и **swap** может быть любой максимальный тип.
Выражение basic_expr не производит статически доступа к каким-либо переменным или каналам.
Семантика.

- Результирующее воздействие выражения **chaos** расходится.
- Результирующее воздействие выражения **skip** заключается в возвращении значения типа **Unit**.

- Результирующим воздействием выражения **stop** является тупиковая ситуация.
- Результирующее воздействие выражения **swap** не специфицируется; это может быть завершение, переход в тупиковую ситуацию, разрешение внутреннего выбора или расходящийся процесс.

6.6. Декартовы произведения (Product Expressions)

Синтаксис.

product_expr ::=
(value_expr-list2)

Атрибуты. Максимальным типом выражения product_expr вида (*value_expr*$_1$,..., *value_expr*$_n$) является декартово произведение $t_1 \times \ldots \times t_n$, где t_1, …, t_n являются соответственно максимальными типами выражений *value_expr*$_1$, …, *value_expr*$_n$.
Выражение product_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются составляющие его выражения value_expr.
Семантика. Результирующее воздействие выражения product_expr вида (*value_expr*$_1$,..., *value_expr*$_n$) состоит в вычислении слева направо значений v_i каждого составляющего выражения *value_expr*$_i$, после чего возвращается значение декартова произведения $(v_1, \ldots, v_n)$.

6.7. Множественные выражения (Set expressions)

Синтаксис.

set_expr ::=
ranged_set_expr |
enumereted_set_expr |
comprehended_set_expr

6.7.1. Множества, заданные диапазоном (Ranged Set Expressions)

Синтаксис.

ranged_set_expr ::=
{ *readonly_integer*-value_expr .. *readonly_integer*-value_expr }

Контекстные условия. Входящие в данную конструкцию выражения value_expr должны представлять собой read-only выражения и их максимальный тип должен быть **Int**.
Атрибуты. Максимальным типом выражения ranged_set_expr является **Int-infset**.

Выражение ranged_set_expr статически производит чтение из тех переменных, из которых статически производят чтение составляющие его выражения value_expr, и не имеет другого статического доступа.

Семантика. Результирующее воздействие выражения ranged_set_expr заключается в возвращении множества целых чисел в диапазоне, ограниченном нижней и верхней границами. При этом первое выражение value_expr вычисляется и возвращает нижнюю границу i_1, затем вычисляется второе выражение value_expr и возвращает верхнюю границу i_2. Результирующее множество содержит все целые числа i такие, что $i_1 \le i \le i_2$. Если $i_1 > i_2$, множество пусто.

6.7.2. Множества, заданные перечислением (Enumerated Set Expressions)

Синтаксис.

enumerated_set_expr ::=
  { *readonly*-opt-value_expr-list }

Контекстные условия. Входящие в данную конструкцию выражения value_expr должны представлять собой read-only выражения.

Максимальные типы составляющих выражений value_expr должны иметь наименьшую верхнюю границу.

Атрибуты. Максимальным типом выражения enumerated_set_expr, имеющего в своем составе одно или более выражений value_expr, является *t*-**infset**, где *t* – это наименьшая верхняя граница максимальных типов этих составляющих выражений value_expr. Максимальным типом выражения enumerated_set_expr, не имеющего составляющих выражений value_expr (пустое множество), является *t*-**infset**, где *t* – это типовая переменная, представляющая произвольный максимальный тип.

Выражение enumerated_set_expr статически производит чтение из тех переменных, из которых статически производят чтение составляющие его выражения value_expr, и не имеет другого статического доступа.

Семантика. Результирующее воздействие выражения enumerated_set_expr заключается в возвращении множества явно специфицированных значений. При этом результирующее воздействие выражения enumerated_set_expr вида $\{value\_expr_1, \ldots, value\_expr_n\}$ достигается путем вычисления слева направо значений v_i каждого составляющего выражения $value\_expr_i$, после чего возвращается значение в виде множества $\{v_1, \ldots, v_n\}$. Если список value_expr-list отсутствует, возвращается пустое множество.

6.7.3. Сокращенные множественные выражения (Comprehended Set Expressions)

Синтаксис.

comprehended_set_expr ::=
  { *readonly*-value_expr | set_limitation }

set_limitation ::=
  typing-list opt-restriction

restriction ::=
  • *readonly_logical*-value_expr

Контекстно-независимые расширения. Пустое ограничение nil-restriction представляет собой сокращение ограничения '• **true**'.
Ограничение '• value_expr' является сокращенной записью ограничения '• value_expr ≡ **true**'.
Контекст и правила видимости. В выражении comprehended_set_expr контекст ограничения set_limitation расширяется до выражения value_expr, входящего в состав данного выражения.
Непосредственным контекстом конструкций typing, входящих в состав ограничения set_limitation, является ограничение restriction, входящее в тот же состав.
Контекстные условия. Входящее в состав выражения comprehended_set_expr выражение value_expr должно представлять собой read-only выражение.
Входящее в состав ограничения restriction выражение value_expr должно представлять собой read-only выражение и его максимальный тип должен быть **Bool**.
Атрибуты. Максимальным типом выражения comprehended_set_expr является *t*-**infset**, где *t* – это максимальный тип входящего в его состав выражения value_expr.
Выражение comprehended_set_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются составляющие его выражение value_expr и ограничение set_limitation.
Конструкция set_limitation, в которой присутствует ограничение restriction, статически производит доступ к тем переменным и каналам, к которым статически обращается данное ограничение restriction.
Ограничение restriction статически производит чтение из тех переменных, из которых статически производит чтение входящее в его состав выражение value_expr, и не имеет другого статического доступа.
Семантика. Результирующее воздействие выражения comprehended_set_expr заключается в возвращении множества, элементы которого получаются путем вычисления выражения value_expr во всех моделях, удовлетворяющих заданному ограничению.

Для каждой модели из множества моделей, представленного ограничением set_limitation (см. ниже), вычисляется указанное выражение value_expr. Вычисление выражения comprehended_set_expr представляет собой сходящийся процесс.
Конструкция set_limitation представляет некоторое подмножество моделей, которые удовлетворяют определениям, заданным в списке typing-list, и для которых выполняется ограничение restriction.
Ограничение restriction выполняется, если вычисление входящего в его состав выражения value_expr представляет собой сходящийся процесс, в результате которого возвращается значение **true**.

6.8. Выражения, задающие списки (List expressions)

Синтаксис.

list_expr ::=
  ranged_list_expr |
  enumereted_list_expr |
  comprehended_list_expr

6.8.1. Списки, заданные диапазоном (Ranged List Expressions)

Синтаксис.

ranged_list_expr ::=
  ⟨ *integer*-value_expr .. *integer*-value_expr ⟩

Контекстные условия. Максимальным типом входящих в данную конструкцию выражений value_expr должен быть **Int**.
Атрибуты. Максимальным типом выражения ranged_list_expr является $\mathbf{Int}^{\omega}$.
Выражение ranged_list_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются составляющие его выражения value_expr.
Семантика. Результирующее воздействие выражения ranged_list_expr заключается в возвращении списка целых чисел в диапазоне, ограниченном нижней и верхней границами. При этом вычисляется первое выражение value_expr и возвращает нижнюю границу i_1, затем вычисляется второе выражение value_expr и возвращает верхнюю границу i_2. Результирующий список содержит в возрастающем порядке все целые числа i такие, что $i_1 \le i \le i_2$. Если $i_1 > i_2$, список пуст.

6.8.2. Списки, заданные перечислением (Enumerated List Expressions)

Синтаксис.

enumerated_list_expr ::=
  ⟨ opt-value_expr-list ⟩

Контекстные условия. Максимальные типы составляющих данную конструкцию выражений value_expr должны иметь наименьшую верхнюю границу.

Атрибуты. Максимальным типом выражения enumerated_list_expr, имеющего в своем составе одно или более выражений value_expr, является t^{ω}, где t – это наименьшая верхняя граница максимальных типов этих составляющих выражений value_expr.

Максимальным типом выражения enumerated_list_expr, не имеющего в своем составе выражений value_expr (пустой список), является t^{ω}, где t – это типовая переменная, представляющая произвольный максимальный тип.

Выражение enumerated_list_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются составляющие его выражения value_expr.

Семантика. Результирующее воздействие выражения enumerated_list_expr заключается в возвращении списка явно специфицированных значений. При этом результирующее воздействие выражения enumerated_list_expr вида $\langle value\_expr_1, \ldots, value\_expr_n \rangle$ достигается путем вычисления слева направо значений v_i каждого выражения $value\_expr_i$, после чего возвращается значение в виде списка $\langle v_1, \ldots, v_n \rangle$. Если конструкция value_expr-list отсутствует, возвращается пустой список.

6.8.3. Сокращенные выражения для списков (Comprehended List Expressions)

Синтаксис.

comprehended_list_expr ::=
  ⟨ value_expr | list_limitation ⟩

list_limitation ::=
  binding **in** *readonly_list*-value_expr opt-restriction

Контекст и правила видимости. В выражении comprehended_list_expr контекст ограничения list_limitation расширяется до выражения value_expr, входящего в состав данного выражения.

Непосредственным контекстом конструкции binding, входящей в состав ограничения list_limitation, является ограничение restriction, входящее в тот же состав.

Контекстные условия. Входящее в состав ограничения list_limitation выражение value_expr должно представлять собой read-only выражение и его максимальный тип должен быть типом список.

Атрибуты. Максимальным типом выражения comprehended_list_expr является t^{ω}, где t – это максимальный тип входящего в его состав выражения value_expr.

В конструкции list_limitation максимальным контекстным типом связывания binding является t, где t^{ω} – это максимальный тип входящего в состав данной конструкции выражения value_expr.
Выражение comprehended_list_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются составляющие его выражение value_expr и ограничение set_limitation.
Конструкция list_limitation статически производит чтение из тех же переменных, что и входящее в ее состав выражение value_expr. Если в данной конструкции присутствует ограничение restriction, то она также статически производит доступ к тем переменным и каналам, к которым статически обращается указанное ограничение restriction. Никакого другого статического доступа данная конструкция не имеет.
Семантика. Результирующее воздействие сокращенного выражения comprehended_list_expr состоит в том, чтобы возвратить список, генерируемый на основе некоторого другого списка.
Для каждой модели из списка моделей, представленного ограничением list_limitation (см. ниже), вычисляется выражение value_expr, входящее в состав исходного сокращенного выражения, и возвращаемое значение включается в результирующий список на соответствующую позицию. Список моделей, представленный ограничением list_limitation, обрабатывается слева направо.
По ограничению list_limitation список моделей строится следующим образом. Входящее в состав данного ограничения выражение value_expr возвращает некоторый список. Затем этот список просматривается слева направо, и каждый его элемент сопоставляется со связыванием binding для получения набора определений. Если для модели (она в точности одна), удовлетворяющей полученным определениям, выполняется ограничение restriction, то данная модель включается в результирующий список моделей на соответствующую позицию.

6.9. Выражения, задающие отображения (Map expressions)

Синтаксис.

```
map_expr ::=
  enumereted_map_expr |
  comprehended_map_expr
```

6.9.1. Отображения, заданные перечислением (Enumerated Map Expressions)

Синтаксис.

```
enumerated_map_expr ::=
  [ opt-value_expr_pair-list ]
```

value_expr_pair ::=

readonly-value_expr ↦ *readonly*-value_expr

Контекстные условия. В выражении enumerated_map_expr максимальные типы доменов входящих в него пар value_expr_pair должны иметь наименьшую верхнюю границу и максимальные типы областей значения тех же пар value_expr_pair должны иметь наименьшую верхнюю границу.
Составляющие пары value_expr_pair выражения value_expr должны представлять собой read-only выражения.
Атрибуты. Максимальным типом выражения enumerated_map_expr, имеющего в своем составе одну или более пар value_expr_pair, является $t_1 \xrightarrow{\mathbf{m}} t_2$, где t_1 – это наименьшая верхняя граница максимальных типов доменов и t_2 – наименьшая верхняя граница максимальных типов областей значения этих пар value_expr_pair.
Максимальным типом выражения enumerated_map_expr, не имеющего в своем составе выражений value_expr (пустое отображение), является $t_1 \xrightarrow{\mathbf{m}} t_2$, где t_1 и t_2 – это типовые переменные, представляющие произвольные максимальные типы.
Максимальным типом домена и максимальным типом области значения пары value_expr_pair являются максимальные типы первого и второго составляющего эту пару выражения value_expr соответственно.
Выражение enumerated_map_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются составляющие его пары value_expr_pair.
Пара value_expr_pair статически производит чтение из тех переменных, из которых статически производят чтение составляющие его выражения value_expr, и не имеет другого статического доступа.
Семантика. Результирующее воздействие выражения enumerated_map_expr заключается в возвращении отображения явно специфицированных пар. При этом результирующее воздействие выражения enumerated_map_expr вида [$value\_expr\_pair_1$,…, $value\_expr\_pair_n$] состоит в вычислении слева направо пар значений (v_1, v_2) для каждой пары $value\_expr\_pair_i$, после чего возвращается значение в виде отображения, содержащего все эти пары. Результирующее воздействие пары value_expr_pair заключается в вычислении значения v_1 первого выражения value_expr, затем значения v_2 второго выражения value_expr, после чего возвращается пара (v_1, v_2). Если конструкция value_expr_pair-list отсутствует, возвращается пустое отображение.

6.9.2. Сокращенные выражения для отображений (Comprehended Map Expressions)

Синтаксис.

comprehended_map_expr ::=
[value_expr_pair | set_limitation]

Контекст и правила видимости. В выражении comprehended_map_expr контекст ограничения set_limitation расширяется до входящей в состав данного выражения пары value_expr_pair.

Атрибуты. Максимальным типом выражения comprehended_map_expr является $t_1 \xrightarrow{\mathbf{m}} t_2$, где t_1 – это максимальный тип домена и t_2 – максимальный тип области значения входящей в данное выражение пары value_expr_pair.

Выражение comprehended_map_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются входящие в его состав пара value_expr_pair и ограничение set_limitation.

Семантика. Результирующее воздействие выражения comprehended_map_expr заключается в возвращении отображения, пары которого получаются путем вычисления входящей в это выражение конструкции value_expr_pair во всех моделях, удовлетворяющих заданному ограничению. Для каждой модели из множества моделей, представленного ограничением set_limitation, вычисляется пара value_expr_pair. Если процесс вычисления указанной пары value_expr_pair сходится, результирующее значение пары включается в искомое отображение. Если вычисление такой пары не является сходящимся процессом, оно не вносит никакой пары в искомое отображение. Вычисление выражения comprehended_map_expr представляет собой сходящийся процесс.

6.10. Аппликативные выражения (Application Expressions)

Синтаксис.

application_expr ::=
  list_or_map_or_function-value_expr actual_function_parameter-string

actual_function_parameter ::=
  (opt-value_expr-list)

Контекстно-независимые расширения. Любое аппликативное выражение application_expr может быть разложено в выражение вида *value_expr*$_1$*(value_expr*$_2$ *)*.

Аппликативное выражение application_expr вида:

value_expr actual_function_parameter$_1$ … actual_function_parameter$_n$

где $n > 1$, является сокращенной записью выражения:

(…(value_expr actual_function_parameter$_1$)…) actual_function_parameter$_n$

Аппликативное выражение application_expr вида *value_expr()* представляет собой сокращенную запись выражения *value_expr(())*.

Аппликативное выражение application_expr вида:

value_expr (value_expr$_1$,…, value_expr$_n$)

где $n > 1$, является сокращенной записью выражения:

value_expr ((value_expr$_1$,…, value_expr$_n$))

Контекстные условия. В аппликативном выражении application_expr, имеющем только один фактический параметр actual_function_parameter, максимальный тип выражения value_expr должен быть либо списком, либо отображением, либо функциональным типом. Кроме того, если максимальный тип выражения value_expr является:

- типом список — t^{ω}, то максимальным типом фактического параметра actual_function_parameter должен быть **Int**;
- типом отображение — $t_1 \xrightarrow{\mathbf{m}} t_2$, то максимальный тип фактического параметра actual_function_parameter должен быть меньше или равен типу t_1;
- функциональным типом — $t_1 \tilde{\to} \; oads \; t_2$, то максимальный тип фактического параметра actual_function_parameter должен быть меньше или равен типу t_1.

Атрибуты. Максимальный тип аппликативного выражения application_expr, имеющего только один фактический параметр actual_function_parameter, определяется максимальным типом входящего в его состав выражения value_expr. А именно, если максимальный тип данного выражения value_expr является:

- типом список — t^{ω}, то максимальный тип аппликативного выражения application_expr представляет собой тип t элементов этого списка;
- типом отображение — $t_1 \xrightarrow{\mathbf{m}} t_2$, то максимальный тип аппликативного выражения application_expr представляет собой тип t_2 области значения этого отображения;
- функциональным типом — $t_1 \tilde{\to} \; oads \; t_2$, то максимальный тип аппликативного выражения application_expr представляет собой тип результата t_2.

Максимальным типом фактического параметра actual_function_parameter, имеющего в своем составе одно выражение value_expr, является максимальный тип этого выражения value_expr. Фактический параметр actual_function_parameter указанного вида статически производит доступ к тем переменным и каналам, к которым статически обращается выражение value_expr.

Аппликативное выражение application_expr, имеющее только один фактический параметр actual_function_parameter, статически производит доступ к тем переменным и каналам, к которым статически обращаются входящие в его состав выражение value_expr и фактический параметр actual_function_parameter. Кроме того, если максимальным типом указанного выражения value_expr является функциональный тип $t_1 \tilde{\to} \; oads \; t_2$, то

аппликативное выражение application_expr статически производит доступ также и к тем переменным и каналам, к которым статически обращается тело функции согласно описанию доступа *oads*.

Семантика. Результирующее воздействие аппликативного выражения application_expr заключается в применении некоторого значения, представляющего собой список, отображение или функцию, к какому-либо фактическому параметру.

Результирующее воздействие аппликативного выражения application_expr вида $value\_expr_1(value\_expr_2)$ состоит в вычислении выражения $value\_expr_2$ для получения фактического параметра, затем вычислении значения выражения $value\_expr_1$ и, наконец, применении полученного значения к указанному фактическому параметру. При этом в зависимости от вида значения, которое применяется к фактическому параметру, такое применение выполняется следующим образом:

- если данное значение представляет собой список, то фактический параметр должен принадлежать множеству индексов данного списка (множеству целых чисел в диапазоне от 1 до длины списка). В этом случае возвращаемым значением становится элемент списка, стоящий в данной позиции. В противном случае результат применения является недоспецифицированным.
- если данное значение представляет собой отображение, то фактический параметр должен принадлежать домену отображения. В этом случае возвращаемым значением является то значение, в которое отображается указанный фактический параметр, (если такое значение в точности одно) или недетерминированный выбор между значениями, в которые отображается данный фактический параметр, (если их несколько). Если фактический параметр не принадлежит указанному отображению, то результат применения является недоспецифицированным.
- если данное значение является функцией, то соответствующая функция применяется к указанному фактическому параметру.

6.11. Квантифицированные выражения (Quantified Expressions)

Синтаксис.

```
quantified_expr ::=
  quantifier typing-list restriction

quantifier ::=
  ∀ |
  ∃ |
  ∃!
```

Контекст и правила видимости. В квантифицированном выражении quantified_expr контекстом входящих в него конструкций typing является ограничение restriction.

Атрибуты. Максимальным типом квантифицированного выражения quantified_expr является **Bool**.

Квантифицированное выражение quantified_expr статически производит доступ к тем переменным и каналам, к которым статически обращается входящее в его состав ограничение restriction.

Семантика. Результирующее воздействие квантифицированного выражения quantified_expr заключается в возвращении булевского значения, зависящего от значения, которое возвращает некоторый предикат для каждой модели из некоторого множества моделей. Моделями, о которых идет речь, являются все модели, удовлетворяющие определениям, представленным указаниями типа typing-list.

Если в качестве квантора quantifier используется $\forall$, возвращаемое значение равно **true** тогда и только тогда, когда ограничение restriction выполняется для всех рассматриваемых моделей. Если квантором quantifier является $\exists$, возвращаемое значение равно **true** тогда и только тогда, когда ограничение restriction выполняется по крайней мере для одной рассматриваемой модели. И, наконец, если в качестве квантора quantifier используется $\exists!$, возвращаемое значение равно **true** тогда и только тогда, когда ограничение restriction выполняется в точности для одной рассматриваемой модели.

Вычисление квантифицированного выражения quantified_expr представляет собой сходящийся процесс.

6.12. Выражения эквивалентности (Equivalence Expressions)

Синтаксис.

equivalence_expr ::=
  value_expr ≡ value_expr opt-pre_condition

pre_condition ::=
  pre *readonly_logical*-value_expr

Контекстные условия. Максимальные типы входящих в данную конструкцию выражений value_expr должны иметь наименьшую верхнюю границу.

Выражение value_expr, входящее в состав предусловия pre_condition, должно представлять собой read-only выражение и его максимальный тип должен быть **Bool**.

Контекстно-зависимые расширения. Выражение эквивалентности equivalence_expr вида:

$value\_expr_1 \equiv value\_expr_2$ **pre** $value\_expr_3$

представляет собой сокращенную форму выражения:

(value_expr$_3$ ≡ **true**) ⇒ value_expr$_1$ ≡ value_expr$_2$

Атрибуты. Максимальным типом выражения эквивалентности equivalence_expr является **Bool**.

Предусловие pre_condition статически производит чтение из тех переменных, из которых статически производит чтение составляющее его выражение value_expr, и не обращается статически ни к каким каналам.

Выражение эквивалентности equivalence_expr статически производит чтение тех переменных, к которым статически обращаются по чтению или записи входящие в него выражения value_expr. Оно также статически производит чтение переменных, из которых статически осуществляет чтение предусловие pre_condition (в случае его присутствия). Выражение эквивалентности equivalence_expr не обращается статически ни к каким каналам.

Семантика. Результирующее воздействие выражения эквивалентности equivalence_expr заключается в возвращении некоторого булевского значения, зависящего от того, обладают ли два входящих в него выражения value_expr одинаковым результирующим воздействием при вычислении каждого из них в текущем состоянии. Результирующие воздействия указанных двух выражений value_expr используются только для определения этого булевского значения и игнорируются в дальнейшем.

Значение, возвращаемое выражением эквивалентности equivalence_expr вида:

value_expr$_1$ ≡ value_expr$_2$

равно **true** тогда и только тогда, когда выражение *value_expr$_1$*, вычисленное в текущем состоянии, представляет в точности такое же результирующее воздействие, что и выражение *value_expr$_2$* , вычисленное в том же состоянии. Это означает, что указанные два выражения *value_expr* должны представлять одно и то же результирующее воздействие в отношении возвращаемых значений, изменений состояния, предложений взаимодействия, внешних выборов, тупиковых ситуаций, внутренних выборов и расходимости.

Вычисление выражения эквивалентности equivalence_expr представляет собой сходящийся процесс.

6.13. Пост-выражения (Post-expressions)

Синтаксис.

```
post_expr ::=
  value_expr post_condition opt-pre_condition

post_condition ::=
  opt-result_naming post readonly_logical-value_expr
```

result_naming ::=
as binding

Контекст и правила видимости. В постусловии post_condition контекстом конструкции opt-result_naming является выражение value_expr.
Контекстные условия. В постусловии post_condition входящее в него выражение value_expr должно представлять собой read-only выражение и его максимальный тип должен быть **Bool**.
Контекстно-зависимые расширения. Пост-выражение post_expr вида:

$value\_expr_1$ **post** $value\_expr_2$

представляет собой сокращенную запись выражения:

$value\_expr_1$ **as** id **post** $value\_expr_2$

где *id* - это некоторый идентификатор уже вне контекста.
Пост-выражение post_expr вида:

$value\_expr_1$ post_condition **pre** $value\_expr_2$

представляет собой сокращенную запись выражения:

($value\_expr_2 \equiv$ **true**) $\Rightarrow$ $value\_expr_1$ post_condition

Атрибуты. Максимальным типом пост-выражения post_expr является **Bool**.
Контекст постусловия post_condition определяет максимальный контекстный тип данного постусловия post_condition.
В пост-выражении post_expr максимальным контекстным типом постусловия post_condition является максимальный тип составляющего его выражения value_expr.
В конструкции result_naming максимальным контекстным типом связывания binding является максимальный контекстный тип ближайшего объемлющего постусловия.
Пост-выражение post_expr статически производит чтение тех переменных, к которым статически обращается по чтению или записи входящее в него выражение value_expr. Оно также статически производит чтение переменных, из которых статически осуществляет чтение предусловие pre_condition (в случае его присутствия). Пост-выражение post_expr не обращается статически ни к каким каналам.
Семантика. Результирующее воздействие пост-выражения post_expr заключается в возвращении некоторого булевского значения, которое зависит от результирующего воздействия выражения value_expr, вычисленного в пред-состоянии (pre-state). Результирующее воздействие выражения value_expr используется только для определения этого булевского значения и игнорируется в дальнейшем.
Значение, возвращаемое пост-выражением post_expr вида:

$value\_expr_1$ **as** binding **post** $value\_expr_2$

равно **true** тогда и только тогда, когда:

- вычисление выражения *value_expr*$_1$ представляет собой сходящийся процесс;
- вычисление выражения *value_expr*$_2$ сходится, и его результат равен **true** при выполнении этого вычисления в пост-состоянии (post-state), которое является результатом вычисления выражения *value_expr*$_1$ в пред-состоянии (pre-state) и в контексте определений, полученных путем сопоставления возвращаемого выражением *value_expr*$_1$ значения со связыванием *binding*.

Внутри выражения *value_expr*$_2$ на значения переменных в пред-состоянии можно ссылаться путем добавления суффикса ` к именам соответствующих переменных (конструкция pre_name). Переменные пост-состояния доступны по своим обычным именам (без суффикса `).

Вычисление пост-выражения post_expr представляет собой сходящийся процесс.

6.14. Выражения со снятием неопределенности (Disambiguation Expressions)

Синтаксис.

```
disambiguation_expr ::=
  value_expr : type_expr
```

Контекстные условия. Максимальный тип выражения value_expr должен быть меньше или равен максимального типа выражения type_expr.

Атрибуты. Максимальным типом выражения со снятием неопределенности disambiguation_expr является максимальный тип выражения type_expr. Выражение disambiguation_expr статически производит доступ к тем переменным и каналам, к которым статически обращается входящее в его состав выражение value_expr.

Семантика. Результирующее воздействие выражения со снятием неопределенности disambiguation_expr совпадает с результирующим воздействием выражения value_expr.

6.15. Выражения в скобках (Bracketed Expressions)

Синтаксис.

```
bracketed_expr ::=
  ( value_expr )
```

Атрибуты. Максимальным типом выражения в скобках bracketed_expr является максимальный тип выражения value_expr.

Выражение в скобках bracketed_expr статически производит доступ к тем переменным и каналам, к которым статически обращается составляющее его выражение value_expr.

Семантика. Результирующее воздействие выражения в скобках bracketed_expr совпадает с результирующим воздействием входящего в его состав выражения value_expr.

6.16. Инфиксные выражения (Infix Expressions)

Синтаксис.

```
infix_expr ::=
  stmt_infix_expr |
  axiom_infix_expr |
  value_infix_expr
```

6.16.1. Операторные инфиксные выражения (Statement Infix Expressions)

Синтаксис.

```
stmt_infix_expr ::=
  value_expr infix_combinator value_expr
```

Контекстные условия. Для инфиксного комбинатора infix_combinator, являющегося:

□ или ⨅: максимальные типы обоих выражений value_expr должны иметь наименьшую верхнюю границу.

‖ или ⫫: оба указанных выражения value_expr должны иметь максимальный тип **Unit**.

; : максимальный тип первого выражения value_expr должен быть **Unit**.

Атрибуты. Для инфиксного комбинатора infix_combinator, являющегося:

□ или ⨅: максимальным типом операторного инфиксного выражения stmt_infix_expr является наименьшая верхняя граница максимальных типов составляющих его выражений value_expr.

‖ или ⫫: максимальным типом операторного инфиксного выражения stmt_infix_expr является тип **Unit**.

; : максимальным типом операторного инфиксного выражения stmt_infix_expr является максимальный тип второго входящего в него выражения value_expr.

Операторное инфиксное выражение stmt_infix_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются входящие в его состав выражения value_expr.

Семантика. См. определение инфиксных комбинаторов infix_combinator (раздел 13).

6.16.2. Аксиоматические инфиксные выражения (Axiom Infix Expressions)

Синтаксис.

axiom_infix_expr ::=
  logical-value_expr infix_connective *logical*-value_expr

Контекстные условия. Максимальным типом обоих указанных выражений value_expr должен быть тип **Bool**.
Контекстно-зависимые расширения. См. определение инфиксных связок infix_connective (раздел 12.1).

6.16.3. Инфиксные выражения (Value Infix Expressions)

Синтаксис.

value_infix_expr ::=
  value_expr infix_op value_expr

Контекстные условия. Тип $t_1 \times t_2$, где t_1 и t_2 – это максимальные типы указанных двух выражений value_expr, должен быть меньше или равен параметрической части (часть, где задается тип параметров операции) максимального типа инфиксной операции infix_op.
Атрибуты. Максимальным типом инфиксного выражения value_infix_expr является результирующая часть (часть, где задается тип результата) максимального типа инфиксной операции infix_op.
Инфиксное выражение value_infix_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются составляющие его выражения value_expr, а также к переменным и каналам, к которым статически обращается тело соответствующей функции согласно описаниям статического доступа максимального типа инфиксной операции infix_op.
Семантика. Результирующее воздействие инфиксного выражения value_infix_expr заключается в вычислении значения v_1 первого выражения value_expr и затем значения v_2 второго выражения value_expr, после чего возвращается результат применения функции, определяемой инфиксной операцией infix_op, к паре (v_1, v_2).

6.17. Префиксные выражения (Prefix Expressions)

Синтаксис.

prefix_expr ::=
  axiom_prefix_expr |
  universal_prefix_expr |
  value_prefix_expr

6.17.1. Аксиоматические префиксные выражения (Axiom Prefix Expressions)

Синтаксис.

axiom_prefix_expr ::=
  prefix_connective *logical*-value_expr

Контекстно-независимые расширения. См. определение префиксных связок prefix_connective (раздел 12.2).

6.17.2. Универсальные префиксные выражения (Universal Prefix Expressions)

Синтаксис.

universal_prefix_expr ::=
  □ *readonly_logical*-value_expr

Контекстно-независимые расширения. Универсальное префиксное выражение universal_prefix_expr вида:

□ value_expr

эквивалентно выражению:

□ (value_expr ≡ **true**)

Контекстные условия. Максимальным типом выражения value_expr должен быть тип **Bool**.
Выражение value_expr должно представлять собой read-only выражение.
Атрибуты. Максимальным типом универсального префиксного выражения universal_prefix_expr является тип **Bool**.
Универсальное префиксное выражение universal_prefix_expr не обращается статически ни к каким переменным или каналам.
Семантика. Универсальное префиксное выражение universal_prefix_expr вида:

□ value_expr

возвращает значение **true** тогда и только тогда, когда для всех состояний, в которых значения переменных не выходят за границы типов этих переменных, процесс вычисления выражения value_expr сходится и его результат равен **true**.
Вычисление универсального префиксного выражения universal_prefix_expr представляет собой сходящийся процесс.

6.17.3. Префиксные выражения (Value Prefix Expressions)

Синтаксис.

value_prefix_expr ::=
  prefix_op value_expr

Контекстные условия. Максимальный тип выражения value_expr должен быть меньше или равен параметрической части максимального типа префиксной операции prefix_op.
Атрибуты. Максимальным типом префиксного выражения value_prefix_expr является результирующая часть максимального типа префиксной операции prefix_op.
Префиксное выражение value_prefix_expr статически производит доступ к тем переменным и каналам, к которым статически обращается входящее в него выражение value_expr, а также к переменным и каналам, к которым статически обращается тело соответствующей функции согласно описаниям статического доступа максимального типа префиксной операции prefix_op.
Семантика. Результирующее воздействие префиксного выражения value_prefix_expr заключается в возвращении значения, полученного путем применения функции, определяемой префиксной операцией prefix_op, к значению, возвращаемому выражением value_expr.

6.18. Сокращенные выражения (Comprehended Expressions)

Синтаксис.

comprehended_expr ::=
  associative_commutative-infix_combinator { value_expr | set_limitation }

Контекст и правила видимости. В сокращенном выражении comprehended_expr контекст ограничения set_limitation расширяется до выражения value_expr, входящего в состав данного выражения.
Контекстные условия. Инфиксный комбинатор infix_combinator должен обладать свойствами ассоциативности и коммутативности, т.е. он должен быть одним из следующих: ‖, □, ⨅.
Для инфиксного комбинатора ‖ максимальным типом выражения value_expr должен быть **Unit**.
Атрибуты. Максимальным типом сокращенного выражения comprehended_expr является максимальный тип входящего в его состав выражения value_expr.
Сокращенное выражение comprehended_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются составляющие его выражение value_expr и ограничение set_limitation.
Семантика. Результирующее воздействие сокращенного выражения comprehended_expr заключается в применении инфиксного комбинатора

infix_combinator к некоторому множеству выражений value_expr вместо всего лишь двух таких выражений. Подобное расширение действия инфиксного комбинатора на множество выражений возможно за счет того, что все допустимые здесь инфиксные комбинаторы infix_combinator являются коммутативными и ассоциативными.

Данное множество содержит выражение value_expr для каждой модели из множества моделей, представленного ограничением set_limitation. Это выражение value_expr вычисляется в соответствующей модели и в конкретном текущем состоянии. Результирующим воздействием вычисления сокращенного выражения comprehended_expr является:

- результирующее воздействие от разрешения внешнего выбора между результирующими воздействиями вычисления выражения value_expr в этих моделях (в случае □),
- результирующее воздействие от разрешения внутреннего выбора между результирующими воздействиями вычисления выражений value_expr в этих же моделях (в случае ⨅),
- результирующее воздействие от параллельного вычисления выражений value_expr в тех же моделях (в случае ‖).

Если множество содержит единственное выражение value_expr, сокращенное выражение comprehended_expr представляет то же результирующее воздействие, что и указанное выражение value_expr. Если множество пусто, то:

□ {} ≡ **stop**

⨅ {} ≡ **swap**

‖ {} ≡ **skip**

6.19. Инициализирующие выражения (Initialise Expressions)

Синтаксис.

initialise_expr ::=
  initialise

Атрибуты. Максимальным типом инициализирующего выражения initialise_expr является **Unit**.

Инициализирующее выражение initialise_expr статически производит запись во все переменные, инициализируемые этим выражением, и не обращается статически ни к каким каналам.

Семантика. Результирующее воздействие инициализирующего выражения initialise_expr состоит в том, чтобы присвоить переменным их начальные значения. Инициализирующее выражение initialise_expr возвращает **Unit** значение. Начальное значение переменной может быть задано явно в определении этой переменной; если оно там явно не задано, то, тем не менее,

существует некоторое определенное значение, которое всегда присваивается данной переменной инициализирующим выражением initialise_expr.
С помощью данного выражения инициализируются все переменные, определенные доступом '**any**' (см. раздел 5.11).

6.20. Выражения присваивания (Assignment Expressions)

Синтаксис.

assignment_expr ::=
variable-name := value_expr

Контекстные условия. Имя name должно представлять некоторую переменную.
Максимальный тип выражения value_expr должен быть меньше или равен максимального типа имени name.
Атрибуты. Максимальным типом выражения присваивания assignment_expr является **Unit**.
Выражение присваивания assignment_expr статически производит запись в переменную, представленную входящим в его состав именем name, а также статически производит доступ к переменным или каналам, к которым статически обращается выражение value_expr.
Семантика. Результирующее воздействие выражения присваивания assignment_expr состоит в том, чтобы записать (присвоить) значение выражения value_expr в переменную, представленную именем name. Значением, возвращаемым выражением присваивания assignment_expr, является **Unit** значение.

6.21. Input-выражения (Input Expressions)

Синтаксис.

input_expr ::=
channel-name ?

Контекстные условия. Имя name должно представлять некоторый канал.
Атрибуты. Максимальным типом input-выражения input_expr является максимальный тип входящего в его состав имени name.
Выражение input_expr статически производит ввод по каналу, представленному именем name, и не обращается статически ни к каким переменным.
Семантика. Результирующее воздействие выражения input_expr состоит в том, чтобы предложить ввести данные по каналу, представленному указанным именем name. Значением, возвращаемым выражением input_expr, является значение, введенное по указанному каналу, в случае, если

предложение произвести ввод совпадет с параллельным предложением произвести вывод по тому же каналу.

6.22. Output-выражения (Output Expressions)

Синтаксис.

output_expr ::=
  channel-name ! value_expr

Контекстные условия. Имя name должно представлять некоторый канал. Максимальный тип выражения value_expr должен быть меньше или равен максимального типа имени name.
Атрибуты. Максимальным типом output-выражения output_expr является **Unit**.
Выражение output_expr статически производит вывод по каналу, представленному входящим в его состав именем name, а также статически обращается к переменным или каналам, к которым статически производит доступ выражение value_expr.
Семантика. Результирующее воздействие выражения output_expr состоит в том, чтобы предложить вывести данные по каналу, представленному указанным именем name. Предлагаемым для вывода значением является значение, возвращаемое выражением value_expr. Выражение output_expr возвращает **Unit** значение в случае, если предложение произвести вывод совпадет с параллельным предложением произвести ввод по тому же каналу.

6.23. Составные выражения (Structured Expressions)

Синтаксис.

structured_expr ::=
  local_expr |
  let_expr |
  if_expr |
  case_expr |
  while_expr |
  until_expr |
  for_expr

6.23.1. Локальные выражения (Local Expressions)

Синтаксис.

local_expr ::=
  local opt-decl-string **in** value_expr **end**

Контекст и правила видимости. Контекстом объявления opt-decl-string является само это объявление opt-decl-string и выражение value_expr. Это

означает, что порядок определений в объявлении opt-decl-string несущественен.

Контекстные условия. Входящие в локальное выражение объявления decl должны быть совместимы.

Максимальный тип выражения value_expr не должен включать в себя абстрактные типы, переменные и каналы, определенные в объявлениях decl.

Атрибуты. Максимальным типом локального выражения local_expr является максимальный тип содержащегося в нем выражения value_expr.

Локальное выражение local_expr статически производит доступ к тем нелокальным переменным и каналам (т.е. к переменным и каналам, не определяемым в объявлении opt-decl-string), к которым статически обращается входящее в его состав выражение value_expr.

Семантика. Результирующим воздействием локального выражения local_expr является результирующее воздействие содержащегося в нем выражения value_expr, вычисленного в контексте определений, заданных в объявлениях decl. Данное выражение value_expr вычисляется в каждой модели, удовлетворяющей указанным определениям, и затем производится недетерминированный выбор между результирующими воздействиями этих вычислений.

В конце вычисления выражения value_expr любое ожидающее обработки взаимодействие по каналу, объявленному в объявлении opt-decl-string, замещается на **stop**.

6.23.2. Let-выражения (Let Expressions)

Синтаксис.

```
let_expr ::=
  let let_def-list in value_expr end

let_def ::=
  typing |
  explicit_let |
  implicit_let

explicit_let ::=
  let_binding = value_expr
implicit_let ::=
  single_typing restriction

let_binding ::=
  binding |
  record_pattern |
  list_pattern
```

Контекстно-независимые расширения. Выражение let_expr, содержащее в себе более одного определения let_def, является сокращенной формой записи

для некоторого числа вложенных let-выражений с единственным определением. То есть выражение let_expr вида:

let let_def$_1$, …, let_def$_n$ **in** value_expr **end**

является сокращенной формой записи следующего выражения:

let let_def$_1$ **in**
⋮
let let_def$_n$ **in** value_expr **end**
⋮
end

Контекст и правила видимости. В выражении let_expr вида:

let let_def **in** value_expr **end**

контекстом определения *let_def* является выражение *value_expr*.

Атрибуты. Максимальным типом выражения let_expr является максимальный тип содержащегося в нем выражения value_expr.

В явном определении explicit_let максимальным контекстным типом конструкции let_binding является максимальный тип выражения value_expr.

Выражение let_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются входящие в его состав определения let_def и выражение value_expr.

Указание типа typing не обращается статически ни к каким переменным или каналам.

Явное определение explicit_let статически производит доступ к тем переменным и каналам, к которым статически обращается входящее в его состав выражение value_expr.

Неявное определение implicit_let статически производит доступ к тем переменным и каналам, к которым статически обращается входящее в его состав ограничение restriction.

Семантика. Выражение let_expr, содержащее одно определение let_def и имеющее вид:

let let_def **in** value_expr **end**

вычисляется следующим образом. Определение *let_def* представляет некоторое множество моделей, как описано ниже. Указанное выражение *value_expr* вычисляется в каждой такой модели и между результирующими эффектами этих вычислений производится недетерминированный выбор.

Различаются три вида определений let_def:

- определение let_def в форме указания типа typing представляет множество моделей, которые удовлетворяют определениям, заданных указанием типа typing;
- определение let_def в неявной форме implicit_let представляет некоторое подмножество моделей, которые удовлетворяют определениям,

заданных указанием типа single_typing: тех из них, где выполняется ограничение restriction;

- определение let_def в явной форме explicit_let представляет множество моделей, получаемое следующим образом. Выражение value_expr вычисляется, и затем возвращаемое им значение сопоставляется со связыванием let_binding. Если данное значение соответствует связыванию let_binding, результатом является некоторый набор определений и искомое множество моделей содержит единственную модель, удовлетворяющую этим определениям. Если же, напротив, данное значение не соответствует связыванию let_binding, то искомое множество моделей пусто.

6.23.3. If- выражения (If Expressions)

Синтаксис.

```
if_expr ::=
  if logical-value_expr then
    value_expr
opt-elsif_branch-string
opt-else_branch
end

elsif_branch ::=
  elsif logical-value_expr then value_expr

else_branch ::=
  else value_expr
```

Контекстно-независимые расширения. If-выражение if_expr, содержащее ветви elsif_branch, представляет собой сокращенную форму записи некоторого числа вложенных if-выражений if_expr без ветвей elsif_branch. Выражение if_expr вида:

```
if value_expr1 then value_expr1′
elsif value_expr2 then value_expr2′
  ⋮
elsif value_exprn then value_exprn′
opt-else_branch
end
```

является сокращенной формой записи выражения:

```
if value_expr1 then value_expr1′ else
  if value_expr2 then value_expr2′ else
  ⋮
    if value_exprn then value_exprn′ opt-else_branch end
  ⋮
  end
end
```

Выражение if_expr вида:

if value_expr$_1$ **then** value_expr$_2$ **end**

является краткой записью выражения:

if value_expr$_1$ **then** value_expr$_2$ **else skip end**

Контекстные условия. В выражении if_expr вида:

if *value_expr$_1$* ***then*** *value_expr$_2$* ***else*** *value_expr$_3$* ***end***

максимальным типом выражения *value_expr$_1$* должен быть тип **Bool** и максимальные типы выражений *value_expr$_2$* и *value_expr$_3$* должны иметь наименьшую верхнюю границу.

Атрибуты. Для if-выражения вида:

if *value_expr$_1$* ***then*** *value_expr$_2$* ***else*** *value_expr$_3$* ***end***

атрибуты определяются следующим образом. Максимальным типом указанного выражения является наименьшая верхняя граница максимальных типов выражений *value_expr$_2$* и *value_expr$_3$*.

Рассматриваемое выражение if_expr статически обращается к тем переменным и каналам, к которым статически обращаются выражения *value_expr$_1$*, *value_expr$_2$* и *value_expr$_3$*.

Семантика. Результирующее воздействие выражения if_expr состоит в том, чтобы определить применимую альтернативу и получить эффект от ее применения. Выражение if_expr вида:

if *value_expr$_1$* ***then*** *value_expr$_2$* ***else*** *value_expr$_3$* ***end***

вычисляется путем вычисления выражения *value_expr$_1$*, которое возвращает некоторое булевское значение — тестовое значение. Если полученное тестовое значение равно **true**, вычисляется выражение *value_expr$_2$*. В противном случае (значение тестового выражения равно **false**) вычисляется выражение *value_expr$_3$*.

6.23.4. Case-выражения (Case Expressions)

Синтаксис.

case_expr ::=
  case value_expr **of** case_branch-list **end**
case_branch ::=
  pattern → value_expr

Контекст и правила видимости. В ветви case_branch контекстом образца pattern является выражение value_expr.

Контекстные условия. В выражении case_expr максимальные типы выражений value_expr, содержащихся в ветвях case_branch, должны иметь наименьшую верхнюю границу.

Атрибуты. Максимальным типом case-выражения case_expr является наименьшая верхняя граница максимальных типов выражений value_expr в соответствующих ветвях case_branch.
В выражении case_expr максимальным контекстным типом образцов pattern в ветвях case_branch является максимальный тип выражения value_expr.
Выражение case_expr статически производит доступ к тем переменным и каналам, к которым статически обращаются содержащиеся в нем выражение value_expr и ветви case_branch.
Ветвь case_branch статически производит доступ к тем переменным и каналам, к которым статически обращается входящее в ее состав выражение value_expr.
Семантика. Результирующее воздействие выражения case_expr состоит в том, чтобы вычислить выражение value_expr, определить соответствующую ветвь case_branch и затем вычислить выражение value_expr, содержащееся в этой ветви. Выражение value_expr вычисляется и возвращает некоторое значение — тестовое значение. Затем просматриваются слева направо ветви case_branch до тех пор, пока данное тестовое значение не совпадет с каким-либо образцом pattern. Такое успешное сопоставление образца с тестовым значением дает в результате некоторый набор определений. В завершении в контексте этих определений вычисляется соответствующее выражение value_expr, содержащееся в ветви case_branch, на которой произошло отождествление образца и тестового значения.
Если ни на одной ветви case_branch не было достигнуто успеха в сопоставлении тестового значения с соответствующими образцами, результирующим воздействием выражения case_expr является недетерминированный выбор между результирующими воздействиями в пустом множестве, т.е. **swap**.

6.23.5. While-выражения (While Expressions)

Синтаксис.

```
while_expr ::=
  while logical-value_expr do unit-value_expr end
```

Контекстные условия. Максимальным типом первого выражения value_expr должен быть тип **Bool**.
Максимальным типом второго выражения value_expr должен быть тип **Unit**.
Атрибуты. Максимальным типом выражения while_expr является **Unit**.
Выражение while_expr статически обращается к тем переменным и каналам, к которым статически обращаются составляющие его выражения value_expr.
Семантика. Результирующее воздействие выражения while_expr состоит в том, чтобы повторять вычисление второго выражения value_expr, пока вычисление первого булевского выражения value_expr возвращает значение

true. Значением, возвращаемым выражением while_expr, является **Unit** значение.
Имеет место следующая эквивалентность:

while value_expr$_1$ **do** value_expr$_2$ **end**

≡

if value_expr$_1$ **then**
  value_expr$_2$; **while** value_expr$_1$ **do** value_expr$_2$ **end**
else skip end

6.23.6. Until-выражения (Until Expressions)

Синтаксис.

until_expr ::=
  do *unit*-value_expr **until** *logical*-value_expr **end**

Контекстно-независимые расширения. Выражение until_expr вида:

do value_expr$_1$ **until** value_expr$_2$ **end**

представляет собой краткую форму записи выражения:

value_expr$_1$; **while** ~ value_expr$_2$ **do** value_expr$_1$ **end**

6.23.7. For-выражения (For Expressions)

Синтаксис.

for_expr ::=
  for list_limitation **do** *unit*-value_expr **end**

Контекстные условия. Максимальным типом выражения value_expr должен быть тип **Unit**.
Контекстно-зависимые расширения. Выражение for_expr вида:

for list_limitation **do** value_expr **end**

представляет собой краткую форму записи выражения:

let id = ⟨ value_expr | list_limitation ⟩ **in skip end**

7. Связывания (Bindings)

Синтаксис.

binding ::=
  id_or_op |
  product_binding

product_binding ::=
  (binding-list2)

Терминология. Значение может быть *сопоставлено* со связыванием binding для получения некоторого набора определений значений.

Контекстные условия. Максимальный контекстный тип для связывания binding, представляющего собой операцию op, должен быть каким-либо функциональным типом, различимым с максимальным типом (типами) предопределенного значения (значений) операции op. Если операция op является инфиксной операцией infix_op, то типом параметра данного функционального типа должен быть тип декартова произведения длины 2.

Максимальным контекстным типом конструкции product_binding должен быть тип декартова произведения той же длины, что и список binding-list2.

В конструкции product_binding составляющие ее связывания binding должны быть совместимы.

Атрибуты. Контекст связывания binding определяет *максимальный контекстный тип* для данного связывания binding. Для конструкций, содержащих в своем составе связывания binding, этот максимальный контекстный тип указывается в соответствующих разделах.

В качестве максимального типа конструкции id_or_op, представляющей собой связывание binding, принимается максимальный контекстный тип данного связывания binding.

В конструкции product_binding вида (*binding*$_1$,..., *binding*$_n$) , контекстный тип которой имеет вид $t_1 \times \ldots \times t_n$, максимальными контекстными типами связываний *binding*$_1$,..., *binding*$_n$ являются типы $t_1, \ldots, t_n$ соответственно.

Семантика. Совпадение некоторого значения v максимального контекстного типа t со связыванием binding вида id_or_op дает следующее определение:

id_or_op : t = v

Совпадение значения некоторого декартова произведения $(v_1,\ldots,v_n)$ максимального контекстного типа $t_1 \times \ldots \times t_n$ со связыванием product_binding вида (*binding*$_1$,..., *binding*$_n$) дает набор определений, полученных при совпадении каждого значения v_i максимального контекстного типа t_i со связыванием *binding*$_i$.

8. Указания типа (Typings)

Синтаксис.

```
typing ::=
  single_typing |
  multiple_typing

single_typing ::=
  binding : type_expr

multiple_typing ::=
  binding-list2 : type_expr
```

commented_typing ::=
  opt-comment-string typing

Контекстно-независимые расширения. Все множественные указания типа multiple_typing и typing-list являются сокращенной формой для единичных указаний типа. Так, множественное указание типа multiple_typing вида:

$binding_1,\ldots, binding_n$: type_expr

является сокращенной формой следующей записи:

$(binding_1,\ldots, binding_n)$: type_expr $\times \ldots \times$ type_expr

где декартово произведение типов имеет длину *n*.
Список typing-list вида:

$binding_1$: $type\_expr_1,\ldots, binding_n$: $type\_expr_n$

является сокращенной формой следующей записи:

$(binding_1,\ldots, binding_n)$: $type\_expr_1 \times \ldots \times type\_expr_n$

Разложение на единичные указания типа списка typing-list, включающего множественные указания типа multiple_typing, начинается с разложения этих множественных указаний типа multiple_typing.
Атрибуты. Максимальным типом единичного указания типа single_typing является максимальный тип выражения type_expr.
В единичном указании типа single_typing максимальным контекстным типом содержащегося в ней связывания binding является максимальный тип соответствующего выражения type_expr.
Максимальное определение определения value_def, представляющего собой конструкцию commented_typing, получается путем замещения в составляющем ее указании типа typing типового выражения type_expr соответствующим выражением максимального типа.
Семантика. Единичное указание типа single_typing вида:

id_or_op : type_expr

представляет определение некоторого значения — *id_or_op* вводится для обозначения значения, тип которого представлен типовым выражением *type_expr*. Единичное указание типа single_typing вида:

$(binding_1,\ldots, binding_n)$: $type\_expr_1 \times \ldots \times type\_expr_n$

представляет набор определений, представленных каждым единичным указанием типа:

$binding_1$: $type\_expr_1$
$\vdots$
$binding_n$: $type\_expr_n$

9. Patterns (образцы)
9.1. Общие замечания

Синтаксис.

pattern ::=
  value_literal |
  pure_value-name |
  wildcard_pattern |
  product_pattern |
  record_pattern |
  list_pattern

Терминология. Значение может *сопоставляться* с образцом pattern или внутренним образцом inner_pattern (см. раздел 9.8), и такое сопоставление может приводить к *неудаче* или *успеху*. В случае успеха результатом сопоставления является некоторый набор определений.
Для каждого вида образцов pattern и внутренних образцов inner_pattern задается свой критерий успешного сопоставления, а также результирующий набор определений для случая успешного сопоставления. Значение, которое сопоставляется с образцом pattern или внутренним образцом inner_pattern, называют ‘тестовым значением’.
Атрибуты. Контекст образца pattern или внутреннего образца inner_pattern определяет *максимальный контекстный тип* для данного образца pattern или внутреннего образца inner_pattern. Для каждой конструкции, содержащей в своем составе образцы pattern или внутренние образцы inner_pattern, этот максимальный контекстный тип указывается в соответствующих разделах.
Контекстно-зависимые расширения. Для образца pattern или внутреннего образца inner_pattern, представляющего собой литеральное значение value_literal, имя name, образец запись record_pattern или образец равенство equality_pattern, определено следующее контекстное условие: максимальный тип, ассоциируемый с этим образцом pattern или внутренним образцом inner_pattern, должен быть меньше или равен его максимального контекстного типа. Для этих образцов применяется *неявное приведение типов*. Неявное приведение типов представляет собой потенциальное приведение (см. раздел 5.1) максимального типа к максимальному контекстному типу. Если неявное приведение типов не является тождественной функцией, то соответствующий образец pattern или внутренний образец inner_pattern представляет собой сокращенную форму образца записи, полученного путем применения указанного неявного приведения типов:

- Если неявное приведение типов является одной функцией *f*, то образец pattern или внутренний образец inner_pattern *p* представляет собой сокращенную форму образца записи record_pattern *f*(= *p*), если *p* является именем name, и *f*(*p*) в противном случае.

- Если неявное приведение типов является композицией функций $f_1 \circ \ldots \circ f_n$ (n ≥ 2), то образец pattern или внутренний образец inner_pattern p представляет собой сокращенную форму образца записи record_pattern $f_1(\ldots(f_n(= p))\ldots)$, если p является именем name, и $f_1(\ldots(f_n(p))\ldots)$ в противном случае.

Аналогичные приведения типов имеют место в выражениях value_expr (см. раздел 6.1).

9.2. Литеральные значения (Value Literals)

Контекстные условия. Для образца pattern или внутреннего образца inner_pattern, представляющего собой литеральное значение value_literal (см. раздел 6.2), максимальный тип этого литерального значения value_literal должен быть меньше или равен максимального контекстного типа указанного образца pattern или внутреннего образца inner_pattern.

Атрибуты. Максимальным типом литерального значения value_literal, представляющего собой некоторый образец pattern, является максимальный тип этого литерального значения value_literal.

Семантика.

- *Успешное сопоставление*: литеральное значение value_literal должно быть равно тестовому значению.
- *Результирующие определения*: нет.

9.3. Имена (Names)

Контекстные условия. Для образца pattern, представляющего собой имя name, это имя name должно представлять некоторое значение.

Максимальный тип этого имени name (см. раздел 10) должен быть меньше или равен максимального контекстного типа указанного образца pattern.

Семантика.

- *Успешное сопоставление*: значение, представленное именем name, должно быть равно тестовому значению.
- *Результирующие определения*: нет.

9.4. Универсальные образцы (Wildcard Patterns)

Синтаксис.

wildcard_pattern ::=

_

Семантика.

- *Успешное сопоставление*: все значения успешно сопоставляются с универсальным образцом wildcard_pattern.

- *Результирующие определения*: нет.

9.5. Образцы декартовы произведения (Product Patterns)

Синтаксис.

product_pattern ::=
(inner_pattern-list2)

Контекстные условия. Максимальный контекстный тип образца декартово произведение product_pattern должен быть типом декартово произведение той же длины, что и список внутренних образцов inner_pattern-list2.
Составляющие данный образец внутренние образцы inner_pattern должны быть совместимы.
Атрибуты. В образце декартово произведение product_pattern вида (*inner_pattern*$_1$,…, *inner_pattern*$_n$), максимальный контекстный тип которой имеет вид $t_1 \times \ldots \times t_n$, максимальными контекстными типами внутренних образцов *inner_pattern*$_1$,…, *inner_pattern*$_n$ являются типы t_1, …, t_n соответственно.
Семантика.

- *Успешное сопоставление*: пусть образец декартово произведение product_pattern имеет вид:

  (inner_pattern$_1$,…, inner_pattern$_n$)

  тогда тестовое значение должно быть декартовым произведением вида $(v_1,\ldots,v_n)$ и в дополнении к этому каждое значение v_i должно сопоставляться с соответствующим внутренним образцом *inner_pattern*$_i$.
- *Результирующие определения*: набор определений, получаемый путем сопоставления каждого значения v_i с внутренним образцом *inner_pattern*$_i$.

9.6. Образцы записи (Record Patterns)

Синтаксис.

record_pattern ::=
pure_value-name (inner_pattern-list)

Контекстные условия. В образце записи record_pattern имя name должно представлять некоторое значение и его максимальный тип должен быть типом чистой функции. Тип результирующей части этого функционального типа должен быть меньше или равен максимального контекстного типа данного образца записи record_pattern. Кроме того, если образец запись record_pattern имеет вид:

name(inner_pattern$_1$,…, inner_pattern$_n$) (n > 1)

то параметрическая часть данного функционального типа должна иметь вид $t_1 \times \ldots \times t_n$.
В образце записи record_pattern составляющие его внутренние образцы inner_pattern должны быть совместимы.
Атрибуты. В образце записи record_pattern вида *name(inner_pattern)* максимальным контекстным типом внутреннего образца *inner_pattern* является параметрическая часть максимального типа указанного имени *name*.
В образце записи record_pattern вида *name(inner_pattern$_1$,…, inner_pattern$_n$)* максимальными контекстными типами внутренних образцов *inner_pattern$_1$,…, inner_pattern$_n$* являются типы t_1, …, t_n соответственно, где тип параметрической части максимального типа имени *name* имеет вид $t_1 \times \ldots \times t_n$.
Семантика.

- *Успешное сопоставление*: пусть образец запись record_pattern имеет вид:

  name(inner_pattern$_1$,…, inner_pattern$_n$) (n > 1)

  и пусть v — тестовое значение, тогда должны существовать значения $v_1,\ldots,v_n$ такие, что:

  v = name(v_1,…, v_n)

  и в дополнении к этому каждое значение v_i должно сопоставляться с соответствующим внутренним образцом *inner_pattern$_i$*.
- *Результирующие определения*: набор определений, получаемый путем сопоставления каждого значения v_i с внутренним образцом *inner_pattern$_i$*, при этом значения $v_1,\ldots,v_n$ недетерминированно выбираются таким образом, чтобы были выполнены условия успешного совпадения, описанные выше.

9.7. Образцы списки (List Patterns)

Синтаксис.

list_pattern ::=
  enumerated_list_pattern |
  concatenated_list_pattern

Контекстные условия. Максимальным контекстным типом образца списка list_pattern должен быть какой-либо списочный тип.

9.7.1. Образцы списки, заданные перечислением (Enumerated List Patterns)

Синтаксис.

enumerated_list_pattern ::=
  ⟨ opt-inner_pattern-list ⟩

Контекстные условия. Составляющие данную конструкцию внутренние образцы inner_pattern должны быть совместимы.
Атрибуты. Максимальным контекстным типом каждого внутреннего образца inner_pattern является тип элемента максимального контекстного типа содержащего их образца списка list_pattern.
Семантика.

- *Успешное сопоставление*: пусть образец список enumerated_list_pattern имеет вид:

  $\langle \text{inner\_pattern}_1, \ldots, \text{inner\_pattern}_n \rangle \quad (n > 0)$

  тогда тестовое значение должно представлять собой список вида $\langle v_1, \ldots, v_n \rangle$ и в дополнении к этому каждое значение v_i должно сопоставляться с соответствующим внутренним образцом $inner\_pattern_i$.
- *Результирующие определения*: набор определений, получаемый путем сопоставления каждого значения v_i с внутренним образцом $inner\_pattern_i$.

9.7.2. Образцы списки, заданные конкатенацией (Concatenated List Patterns)

Синтаксис.

```
concatenated_list_pattern ::=
  enumerated_list_pattern ^ inner_pattern
```

Контекстные условия. Входящие в данную конструкцию образец список enumerated_list_pattern и внутренний образец inner_pattern должны быть совместимы.
Атрибуты. Максимальным контекстным типом образца списка enumerated_list_pattern и внутреннего образца inner_pattern является максимальный контекстный тип содержащего их образца списка concatenated_list_pattern.
Семантика.

- *Успешное сопоставление*: пусть образец список concatenated_list_pattern имеет вид:

  enumerated_list_pattern ^ inner_pattern

  тогда тестовое значение должно представлять собой список вида $l_1 \hat{} l_2$, где l_1 сопоставляется с образцом списком enumerated_list_pattern и l_2 сопоставляться с внутренним образцом *inner_pattern*.
- *Результирующие определения*: набор определений, получаемый путем сопоставления списка l_1 с образцом списком enumerated_list_pattern и списка l_2 с внутренним образцом *inner_pattern*.

9.8. Внутренние образцы (Inner Patterns)

Синтаксис.

```
inner_pattern ::=
  value_literal |
  id_or_op |
  wildcard_pattern |
  product_pattern |
  record_pattern |
  list_pattern |
  equality_pattern
```

Для внутренних образцов используется общая для всех образцов терминология, атрибуты и контекстно-зависимые расширения (см. раздел 9.1).
Определения всех перечисленных здесь конкретных видов образцов кроме id_or_op и equality_pattern даны в разделах 9.2 – 9.7.

9.8.1. Идентификаторы или операции (Identifiers or Operators)

Контекстные условия. Максимальный контекстный тип для внутреннего образца inner_pattern, представляющего собой операцию op, должен быть каким-либо функциональным типом, различимым с максимальным типом (типами) предопределенного значения (значений) операции op. Если операция op является инфиксной операцией infix_op, то типом параметра данного функционального типа должен быть тип декартова произведения длины 2.
Атрибуты. В качестве максимального типа конструкции id_or_op, представляющей собой внутренний образец inner_pattern, принимается максимальный контекстный тип данного внутреннего образца inner_pattern.
Семантика.

- *Успешное сопоставление*: все значения максимального контекстного типа успешно сопоставляются с идентификатором или операцией id_or_op.
- *Результирующие определения*: пусть *v* является тестовым значением и *t* — максимальным контекстным типом конструкции id_or_op, тогда результирующим является следующее определение:

  id_or_op : t = v

9.8.2. Образцы равенства (Equality Patterns)

Синтаксис.

```
equality_pattern ::=
  = pure_value-name
```

Контекстные условия. В образце равенстве equality_pattern имя name должно представлять некоторое значение.
Максимальный тип этого имени name (см. раздел 10) должен быть меньше или равен максимального контекстного типа содержащего его образца равенства equality_pattern.
Семантика.

- *Успешное сопоставление*: значение, представленное именем name, должно быть равно тестовому значению.
- *Результирующие определения*: нет.

10. Имена

10.1. Общие замечания

Синтаксис.

```
name ::=
  id |
  ( op )
```

Атрибуты. Если имя name представляет некоторую схему, то для него определен *максимальный класс*, т.е. максимальный класс ассоциируемого с данным именем классового выражения. В случае параметризованной схемы у этого имени есть также некоторый формальный параметр схемы. Если имя name представляет какой-либо тип, значение, переменную или канал, то с ним (с именем) ассоциируется некоторый максимальный тип. Ниже для каждой из указанных альтернатив определены свои конкретные атрибуты.
Семантика. Имя name представляет некоторую сущность, которая является схемой, типом, значением, переменной или каналом.

10.2. Идентификаторы (Identifiers)

Атрибуты. Если имя name представляет собой идентификатор id, то его атрибутами являются атрибуты данного идентификатора id.
Семантика. Идентификатор id представляет сущность, с которой он связан соответствующим определением.

10.3. Операции (Operators)

Контекст и правила видимости. В конструкции name, являющейся операцией op, видны все предопределенные значения операций.
Атрибуты. Максимальным типом имени name, представляющим собой операцию, является максимальный тип этой операции op.
Семантика. Операция op представляет сущность, с которой она связана соответствующим определением.

Скобки превращают операцию op в функцию, которую можно применять в префиксной нотации как аппликативное выражение (см. раздел 6.10). С учетом того, что операция op может быть префиксной prefix_op или инфиксной infix_op, это означает выполнение следующих эквивалентностей:

prefix_op value_expr ≡ (prefix_op)(value_expr)
$value\_expr_1$ infix_op $value\_expr_2$ ≡ (infix_op)($value\_expr_1$, $value\_expr_2$)

11. Идентификаторы и операции

11.1. Общие замечания

Синтаксис.

```
id_or_op ::=
    id |
    op

op ::=
    infix_op |
    prefix_op
```

Терминология. Любое встречающееся в тексте вхождение идентификатора или операции (id_or_op, id или op) может быть связано либо с их определением — *точка определения* (*defining* occurrence), либо с их применением или использованием — *точка использования* (*applied* occurrence).

Точками определения считаются следующие вхождения:

- идентификатора (идентификаторов) id, встречающегося непосредственно внутри определений scheme_def, sort_def, variant_def, union_def, short_record_def, abbreviation_def, prefix_application, infix_application, single_variable_def, multiple_variable_def, single_channel_def, multiple_channel_def, axiom_naming или id_or_wildcard;
- идентификатора или операции id_or_op, встречающихся непосредственно внутри конструктора constructor, деструктора destructor, реконструктора reconstructor или связывания binding;
- идентификатора или операции id_or_op, которые являются внутренним образцом inner_pattern.

Все остальные вхождения считаются точками использования данного идентификатора или операции.

Каждая точка определения идентификатора или операции является частью некоторой декларативной конструкции, представляющей, по крайней мере, определение, с помощью которого задается данный идентификатор или операция.

Точка использования идентификатора или операции называется *видимой*, если есть какое-либо видимое определение, с помощью которого задается указанный идентификатор или операция.

Допустимая точка использования некоторого идентификатора или операции имеет *соответствующее определение* (или *интерпретацию*). Для любой точки использования идентификатора или операции, возможны три случая:

1. Видимого определения, с помощью которого вводится данный идентификатор или операция, нет, т.е. указанная точка использования не является видимой. В этом случае точка использования считается недопустимой и, следовательно, данный идентификатор или операция не имеют соответствующего определения.
2. Есть в точности одно видимое определение, с помощью которого вводится данный идентификатор или операция. Это определение и является соответствующим определением указанного идентификатора или операции.
3. Есть два или более видимых определений, с помощью которых вводится данный идентификатор или операция. Согласно правилам видимости и контекстным условиям для декларативных конструкций это возможно только для идентификаторов и операций, представляющих значения. В этом случае единственное соответствующее определение находится (если оно вообще может быть найдено) по правилам перегрузки имен. Если невозможно найти единственное соответствующее определение, то данная точка использования считается недопустимой.

Контекст и правила видимости. У всех операций есть одно или более предопределенное значение, контекстом которого служит вся спецификация целиком и которое не может быть скрыто, поэтому операции всегда являются видимыми.

Контекстные условия. Точка использования идентификатора или операции должна быть видимой, и при этом должно существовать единственное соответствующее определение данного идентификатора или операции. Отсюда следует, что максимальный тип точки определения операции должен различаться с максимальным типом (типами) предопределенных значений данной операции.

Атрибуты. Для идентификатора, представляющего схему, его максимальный класс и возможный формальный параметр схемы определяются соответствующим определением этого идентификатора (см. раздел 3.2). Максимальный тип идентификатора или операции, представляющей некоторый тип, значение, переменную или канал, определяется соответствующим определением данного идентификатора или операции.

11.2. Инфиксные операции (Infix Operators)

Синтаксис.

infix_op ::=
= | ≠ | > | < | ≥ | ≤ | ⊃ | ⊂ | ⊇ | ⊆ | ∈ | ∉ | + | – | \ | ^ | ∪ | † | * | / |
° | ∩ | ↑

Контекстные условия. Контекстные условия для инфиксных операций описаны в разделе 11.1.

Атрибуты. Максимальный тип инфиксной операции infix_op определяется ее соответствующим определением (см. раздел 11.1). Максимальным типом инфиксной операции infix_op, представляющей свое предопределенное значение, является максимальный тип соответствующего типового выражения, указанного ниже.

Семантика. Семантика инфиксной операции infix_op определяется ее соответствующим определением. Предопределенные значения (семантика) применения инфиксных операций infix_op перечислены ниже.

Инфиксные операции infix_op оперируют с парой значений, являющихся их аргументами. У некоторых инфиксных операций infix_op могут быть предусловия, которые должны выполняться для аргументов этих операций. При нарушении предусловия результирующее воздействие инфиксного выражения value_infix_expr, в составе которого встречается данная инфиксная операция infix_op, считается недоспецифицированным.

Далее приведены значения конкретных инфиксных операций (*T* обозначает типовую переменную, представляющую произвольный тип).

- **Равенство:**

  $= : T \times T \to \textbf{Bool}$

  Результат равен **true** тогда и только тогда, когда значения обоих аргументов равны.

- **Неравенство:**

  $\neq : T \times T \to \textbf{Bool}$

  Результат равен **true** тогда и только тогда, когда значения обоих аргументов не равны.

- **Целочисленное сложение:**

  $+ : \textbf{Int} \times \textbf{Int} \to \textbf{Int}$

  Результат равен сумме двух целочисленных значений.

- **Вещественное сложение:**

  $+ : \textbf{Real} \times \textbf{Real} \to \textbf{Real}$

  Результат равен сумме двух вещественных значений.

- **Целочисленное вычитание:**

  $- : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$

  Результат равен разности первого и второго целочисленных значений.
- **Вещественное вычитание:**

  $- : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$

  Результат равен разности первого и второго вещественных значений.
- **Целочисленное умножение:**

  $* : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$

  Результат равен произведению двух целочисленных значений.
- **Вещественное умножение:**

  $* : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$

  Результат равен произведению двух вещественных значений.
- **Целочисленное возведение в степень:**

  $\uparrow : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$

  Предусловие: второе целочисленное значение должно быть неотрицательным, и оно не должно быть равным 0, если первое целочисленное значение равно 0.
  Результатом является первое целочисленное значение, возведенное в степень второго целочисленного значения.
- **Вещественное возведение в степень:**

  $\uparrow : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Real}$

  Предусловие: если второе вещественное значение отрицательно или равно нулю, то первое вещественное значение должно быть отличным от нуля (0.0); если второе вещественное значение не является целым числом, то первое должно быть неотрицательным.
  Результатом является первое вещественное значение, возведенное в степень второго вещественного значения.
- **Композиция функций:**

  $\circ : (T_2 \xrightarrow{\sim} \mathsf{a}\ T_3) \times (T_1 \xrightarrow{\sim} \mathsf{a}'\ T_2) \to (T_1 \xrightarrow{\sim} \mathsf{a}''\ T_3)$

  где *a″* представляет собой объединение *a* и *a′*.
  Результат равен композиции двух функций, определяемой следующим образом:

  $f_1 \circ f_2 \equiv \lambda x : T_1 \bullet f_1(f_2(x))$
- **Композиция отображений:**

  $\circ : (T_2 \xrightarrow{\mathbf{m}} T_3) \times (T_1 \xrightarrow{\mathbf{m}} T_2) \to (T_1 \xrightarrow{\mathbf{m}} T_3)$

Результат равен композиции двух отображений, определяемой следующим образом:

$$m_1 \circ m_2 \equiv [x \mapsto m_1(m_2(x)) \mid x : T_1 \bullet x \in \mathbf{dom}\ m_2 \wedge m_2(x) \in \mathbf{dom}\ m_1]$$

- **Целочисленное деление:**

  $/ : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$

  Предусловие: второе целочисленное значение должно быть отличным от нуля (0).
  Абсолютным значением (без учета знака) результата является количество раз, которое абсолютное значение второго целочисленного значения содержится в абсолютном значении первого целочисленного значения. Знак результата равен произведению знаков аргументов.
- **Вещественное деление:**

  $/ : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Real}$

  Предусловие: второе вещественное значение должно быть отличным от нуля (0.0).
  Результат получается путем деления первого вещественного значения на второе вещественное значение.
- **Ограничение отображения на подмножестве (Map restriction to):**

  $/ : (T_1 \xrightarrow[\mathbf{m}]{} T_2) \times T_1\text{-}\mathbf{infset} \to (T_1 \xrightarrow[\mathbf{m}]{} T_2)$

  Результатом является отображение, полученное из исходного путем ограничения его домена элементами указанного множества.
- **Остаток от целочисленного деления:**

  $\backslash : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$

  Предусловие: второе целочисленное значение должно быть отличным от нуля (0).
  Абсолютным значением результата является остаток от деления абсолютного значения второго целочисленного значения на абсолютное значение первого целочисленного значения. Знак результата равен знаку первого аргумента. Справедливо следующее соотношение:

  $$a = (a / b) * b + (a \backslash b)$$

  где *a* и *b* — целочисленные значения, *b* отлично от нуля.
- **Разность множеств:**

  $\backslash : T\text{-}\mathbf{infset} \times T\text{-}\mathbf{infset} \to T\text{-}\mathbf{infset}$

  Результатом является множество, все элементы которого входят в первое множество и не входят во второе.

- **Усечение отображения (Map restriction by):**

  $\setminus : (T_1 \xrightarrow[\mathbf{m}]{} T_2) \times T_1\text{-}\mathbf{infset} \to (T_1 \xrightarrow[\mathbf{m}]{} T_2)$

  Результатом является отображение, полученное из исходного путем исключения из его домена элементов указанного множества.
- **Целочисленное больше:**

  $> : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}$

  Результат равен **true** тогда и только тогда, когда первое целочисленное значение больше второго целочисленного значения.
- **Вещественное больше:**

  $> : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Bool}$

  Результат равен **true** тогда и только тогда, когда первое вещественное значение больше второго вещественного значения.
- **Целочисленное меньше:**

  $< : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}$

  Результат равен **true** тогда и только тогда, когда первое целочисленное значение меньше второго целочисленного значения.
- **Вещественное меньше:**

  $< : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Bool}$

  Результат равен **true** тогда и только тогда, когда первое вещественное значение меньше второго вещественного значения.
- **Целочисленное больше или равно:**

  $\geq : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}$

  Результат равен **true** тогда и только тогда, когда первое целочисленное значение больше или равно второго целочисленного значения.
- **Вещественное больше или равно:**

  $\geq : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Bool}$

  Результат равен **true** тогда и только тогда, когда первое вещественное значение больше или равно второго вещественного значения.
- **Целочисленное меньше или равно:**

  $\leq : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}$

  Результат равен **true** тогда и только тогда, когда первое целочисленное значение меньше или равно второго целочисленного значения.
- **Вещественное меньше или равно:**

  $\leq : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Bool}$

Результат равен **true** тогда и только тогда, когда первое вещественное значение меньше или равно второго вещественного значения.

- **Включение в качестве подмножества (строгое):**

  $\supset$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  Результат равен **true** тогда и только тогда, когда второе множество является строгим подмножеством первого множества (т.е. совпадение множеств не допускается).
- **Вхождение в качестве подмножества (строго):**

  $\subset$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  Результат равен **true** тогда и только тогда, когда первое множество является строгим подмножеством второго множества (т.е. совпадение множеств не допускается).
- **Включение в качестве подмножества:**

  $\supseteq$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  Результат равен **true** тогда и только тогда, когда второе множество является подмножеством первого множества.
- **Вхождение в качестве подмножества:**

  $\subseteq$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

  Результат равен **true** тогда и только тогда, когда первое множество является подмножеством второго множества.
- **Принадлежность множеству:**

  $\in$: T $\times$ T-**infset** $\rightarrow$ **Bool**

  Результат равен **true** тогда и только тогда, когда первый аргумент является элементом указанного множества.
- **Отрицание принадлежности множеству:**

  $\notin$: T $\times$ T-**infset** $\rightarrow$ **Bool**

  Результат равен **true** тогда и только тогда, когда первый аргумент не является элементом указанного множества.
- **Пересечение множеств:**

  $\cap$: T-**infset** $\times$ T-**infset** $\rightarrow$ T-**infset**

  Результатом является множество, состоящее из всех элементов, которые входят одновременно в оба указанные множества.
- **Объединение множеств:**

  $\cup$: T-**infset** $\times$ T-**infset** $\rightarrow$ T-**infset**

  Результатом является множество, состоящее из всех элементов, которые входят хотя бы в одно из указанных множеств.

- **Объединение отображений:**

  $\cup : (T_1 \xrightarrow{\mathbf{m}} T_2) \times (T_1 \xrightarrow{\mathbf{m}} T_2) \to (T_1 \xrightarrow{\mathbf{m}} T_2)$

  Результатом является отображение, состоящее из всех пар первого отображения и всех пар второго отображения.
- **Конкатенация списков:**

  $\hat{} : T^{\omega} \times T^{\omega} \xrightarrow{\sim} T^{\omega}$

  Предусловие: первый список должен быть конечным.
  Результатом является конкатенация указанных списков, т.е. результирующий список состоит из всех элементов первого и второго списков, взятых в своем порядке, причем элементы первого списка стоят первыми.
- **Перекрытие отображений:**

  $\dagger : (T_1 \xrightarrow{\mathbf{m}} T_2) \times (T_1 \xrightarrow{\mathbf{m}} T_2) \to (T_1 \xrightarrow{\mathbf{m}} T_2)$

  Результатом является первое отображение, перекрытое вторым отображением. Для всех элементов, принадлежащих одновременно доменам обоих указанных отображений, второе отображение перекрывает (замещает) первое.

11.3. Префиксные операции (Prefix Operators)

Синтаксис.

prefix_op ::=
abs | **int** | **real** | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**

Контекстные условия. Контекстные условия для префиксных операций описаны в разделе 11.1.
Атрибуты. Максимальный тип префиксной операции prefix_op определяется ее соответствующим определением (см. раздел 11.1). Максимальным типом префиксной операции prefix_op, представляющей свое предопределенное значение, является максимальный тип соответствующего типового выражения, указанного ниже.
Семантика. Семантика префиксной операции prefix_op определяется ее соответствующим определением. Предопределенные значения (семантика) применения префиксных операций prefix_op перечислены ниже.
Префиксные операции prefix_op оперируют со значениями (аргументами). У некоторых префиксных операций prefix_op могут быть предусловия, которые должны выполняться для аргументов этих операций. При нарушении предусловия результирующее воздействие префиксного выражения value_prefix_expr, в составе которого встречается данная префиксная операция prefix_op, считается недоспецифицированным, если не указано ничего другого.

Далее приведены значения конкретных префиксных операций (*T* обозначает типовую переменную, представляющую произвольный тип).

- **Целочисленное абсолютное значение:**

  abs : **Int** → **Int**

  Результатом является абсолютное значение целочисленного аргумента. Для отрицательных целых возвращается значение с обратным знаком. Операция является тождественной для неотрицательных целочисленных значений.

- **Вещественное абсолютное значение:**

  abs : **Real** → **Real**

  Результатом является абсолютное значение вещественного аргумента. Для отрицательных вещественных значений возвращается значение с обратным знаком. Операция является тождественной для неотрицательных вещественных значений.

- **Преобразование вещественного в целое:**

  int : **Real** → **Int**

  Абсолютным значением результата является наибольшее целое, которое меньше или равно абсолютного значения вещественного аргумента. Знаком результата является знак аргумента.

- **Преобразование целого в вещественное:**

  real : **Int** → **Real**

  Результат идентичен аргументу, изменяется только его тип. Для функций преобразования типов справедливо соотношение:

  int real a = a

  где *a* — произвольное целочисленное значение.

- **Кардинальность множества:**

  card : T-**infset** $\xrightarrow{\sim}$ **Int**

  Результат равен количеству элементов указанного множества. Эффект применения операции **card** к бесконечному множеству представляет собой расходящийся процесс.

- **Длина списка:**

  len : T^{ω} $\xrightarrow{\sim}$ **Int**

  Результат равен длине указанного списка. Эффект применения операции **len** к бесконечному списку представляет собой расходящийся процесс.

- **Индексы списка:**

  inds : $T^{\omega} \to$ **Int-infset**

  Результатом является множество индексов указанного списка. Пусть *f_list* обозначает некоторый конечный список и *i_list* — бесконечный список, тогда:

  inds f_list = { i | i : **Int** • i ≥ 1 ∧ i ≤ **len** f_list }
  inds i_list = { i | i : **Int** • i ≥ 1 }

- **Элементы списка:**

  elems : $T^{\omega} \to$ T-**infset**

  Результатом является множество элементов указанного списка.

- **Головной элемент списка:**

  hd : $T^{\omega} \xrightarrow{\sim} T$

  Предусловие: список должен быть непустым.
  Результатом является первый элемент указанного списка.

- **Исключение головного элемента из списка:**

  tl : $T^{\omega} \xrightarrow{\sim} T^{\omega}$

  Предусловие: список должен быть непустым.
  Результатом является список, который получается из исходного списка путем удаления его первого элемента.

- **Домен отображения:**

  dom : $(T_1 \xrightarrow[\mathbf{m}]{} T_2) \to T_1$-**infset**

  Результатом является домен указанного отображения: значения, для которых определено это отображение.

- **Область значений отображения:**

  rng : $(T_1 \xrightarrow[\mathbf{m}]{} T_2) \to T_2$-**infset**

  Результатом является область значений указанного отображения: значения, которые могут быть получены путем применения этого отображения к значениям из его домена.

12. Связки

12.1. Инфиксные связки (Infix Connectives)

Синтаксис.

infix_connective ::=
⇒ | ∨ | ∧

Контекстно-независимые расширения. Инфиксные связки infix_connective предназначены для составления новых булевских выражений из уже имеющихся булевских выражений.

При определении результирующего воздействия аксиоматического инфиксного выражения axiom_infix_expr, в котором встречается инфиксная связка infix_connective, используется общее правило, заключающееся в том, что второе выражение вычисляется только в том случае, когда значение первого выражения не определяет однозначно значение всего выражения axiom_infix_expr. Это позволяет обойти в ряде случаев возможное расхождение или тупиковую ситуацию во втором выражении. Значение аксиоматических инфиксных выражений axiom_infix_expr задается в терминах if-выражений, сокращениями которых и являются аксиоматические инфиксные выражения.

- **Логическое и:**

  $value\_expr_1 \wedge value\_expr_2$

  является сокращенной формой записи выражения:

  if $value\_expr_1$ **then** $value\_expr_2$ **else false end**

- **Логическое или:**

  $value\_expr_1 \vee value\_expr_2$

  является сокращенной формой записи выражения:

  if $value\_expr_1$ **then true else** $value\_expr_2$ **end**

- **Импликация:**

  $value\_expr_1 \Rightarrow value\_expr_2$

  является сокращенной формой записи выражения:

  if $value\_expr_1$ **then** $value\_expr_2$ **else true end**

12.2. Префиксные связки (Prefix Connectives)

Синтаксис.

prefix_connective ::=
~

Контекстно-независимые расширения. Префиксная связка prefix_connective предназначена для построения новых булевских выражений из уже имеющихся булевских выражений.

- **Логическое отрицание:**
  Аксиоматическое префиксное выражение axiom_prefix_expr вида:

  ~ value_expr

  является сокращенной формой записи выражения:

  if value_expr **then false else true end**

13. Инфиксные комбинаторы (Infix Combinators)

Синтаксис.

infix_combinator ::=
□ |
|⌝ |
‖ |
╫ |
;

Семантика. Инфиксные комбинаторы infix_combinator применяются для построения выражений, которые либо взаимодействуют, либо оказывают воздействие на переменные. С каждым инфиксным комбинатором infix_combinator связан ряд простых правил доказательства, поясняющих семантику соответствующего инфиксного комбинатора.

- **Внешний выбор (External choice):**

  $value\_expr_1$ □ $value\_expr_2$

  Внешний выбор производится между результирующими воздействиями двух указанных выражений *value_expr*. На выбор может повлиять возможный эффект от параллельного выполнения некоторого третьего выражения.
  Единичным элементом (единицей) внешнего выбора является **stop**, нулевым элементом (нулем) — **chaos**. Внешний выбор обладает свойствами идемпотентности, коммутативности, ассоциативности и дистрибутивности по отношению к внутреннему выбору:

  value_expr □ **stop** ≡ value_expr

  value_expr □ **chaos** ≡ **chaos**

  value_expr □ value_expr ≡ value_expr

  $value\_expr_1$ □ $value\_expr_2$ ≡ $value\_expr_2$ □ $value\_expr_1$

  $value\_expr_1$ □ ($value\_expr_2$ □ $value\_expr_3$) ≡
  ($value\_expr_1$ □ $value\_expr_2$) □ $value\_expr_3$

$value\_expr_1 \ \square \ (value\_expr_2 \ \sqcap \ value\_expr_3) \equiv$
$(value\_expr_1 \ \square \ value\_expr_2) \ \sqcap \ (value\_expr_1 \ \square \ value\_expr_3)$

- **Внутренний выбор (Internal choice):**

  $value\_expr_1 \ \sqcap \ value\_expr_2$

  Внутренний — недетерминированный — выбор производится между результирующими воздействиями двух указанных выражений *value_expr*. Возможный эффект от параллельного выполнения некоторого третьего выражения не может оказать влияние на внутренний выбор.
  Нулевым элементом (нулем) внутреннего выбора является **chaos**. Внутренний выбор обладает свойствами идемпотентности, коммутативности и ассоциативности:

  $value\_expr \ \sqcap \ \mathbf{chaos} \equiv \mathbf{chaos}$

  $value\_expr \ \sqcap \ value\_expr \equiv value\_expr$

  $value\_expr_1 \ \sqcap \ value\_expr_2 \equiv value\_expr_2 \ \sqcap \ value\_expr_1$

  $value\_expr_1 \ \sqcap \ (value\_expr_2 \ \sqcap \ value\_expr_3) \equiv$
  $(value\_expr_1 \ \sqcap \ value\_expr_2) \ \sqcap \ value\_expr_3$

- **Параллельная композиция (Concurrent composition):**

  $value\_expr_1 \ \| \ value\_expr_2$

  Два указанных выражения *value_expr* выполняются параллельно друг с другом до тех пор, пока одно из них не завершится, в то время как другое будет продолжать свое выполнение. Эти выражения могут предложить произвести взаимодействие посредством каналов, в частности они могут предложить взаимодействие друг с другом (если одно из них производит ввод по какому-то каналу, а другое вывод по тому же каналу). Если выражения в состоянии взаимодействовать друг с другом, они могут сделать внешний выбор, который предписывает им произвести это взаимодействие. Альтернативной возможностью является внешний выбор, который оставляет указанным выражениям свободу произвести взаимодействие друг с другом или с другими параллельно выполняющимися выражениями. Если выражения не могут взаимодействовать друг с другом, но в состоянии осуществить взаимодействие с другими параллельно выполняющимися выражениями, им предоставляется возможность выполнить это взаимодействие.
  Единичным элементом (единицей) параллельной композиции является **skip**, нулевым элементом (нулем) — **chaos**. Параллельная композиция обладает свойствами коммутативности, ассоциативности и

дистрибутивности по отношению к внутреннему выбору:

value_expr ‖ **skip** ≡ value_expr

value_expr ‖ **chaos** ≡ **chaos**

$value\_expr_1$ ‖ $value\_expr_2$ ≡ $value\_expr_2$ ‖ $value\_expr_1$

$value\_expr_1$ ‖ ($value\_expr_2$ ‖ $value\_expr_3$) ≡
($value\_expr_1$ ‖ $value\_expr_2$) ‖ $value\_expr_3$

$value\_expr_1$ ‖ ($value\_expr_2$ ⌉⌈ $value\_expr_3$) ≡
($value\_expr_1$ ‖ $value\_expr_2$) ⌉⌈ ($value\_expr_1$ ‖ $value\_expr_3$)

Если выражение *value_expr* сходится и не вызывает взаимодействия и если $c_1 \neq c_2$, то выполняются следующие две эквивалентности:

x := c_1? ‖ c_2! value_expr ≡
(x := c_1? ; c_2! value_expr) □ (c_2! value_expr ; x := c_1?)

x := c? ‖ c! value_expr ≡
(((x := c? ; c! value_expr) □ (c! value_expr ; x := c?))
□ (x := value_expr))
⌉⌈ (x := value_expr)

- **Interlocked-композиция (Interlocked composition):**

  $value\_expr_1$ ⫲ $value\_expr_2$

  Два указанных выражения *value_expr* выполняются, блокируя друг друга по отношению к внешним взаимодействиям (т.е. взаимодействиям с другими какими-то выражениями), до тех пор, пока одно из них не завершится, в то время как другое будет продолжать свое выполнение. Эти выражения могут предложить осуществить взаимодействие посредством каналов, в частности они могут предложить взаимодействие друг с другом (если одно из них производит ввод по какому-то каналу, а другое вывод по тому же каналу). Если выражения будут в состоянии взаимодействовать друг с другом, они выполнят это взаимодействие. Если выражения будут в состоянии взаимодействовать с другими параллельно выполняющимися выражениями, но только не друг с другом, они попадут в тупиковую ситуацию, где и будут пребывать, пока одно из них не сможет завершиться.
  Единицей interlocked-композиции является **skip**, нулем — **chaos**. Interlocked-композиция обладает свойствами коммутативности и дистрибутивности по отношению к внутреннему выбору:

  value_expr ⫲ **skip** ≡ value_expr

  value_expr ⫲ **chaos** ≡ **chaos**

$value\_expr_1 \between value\_expr_2 \equiv value\_expr_2 \between value\_expr_1$

$value\_expr_1 \between (value\_expr_2 \mid\urcorner value\_expr_3) \equiv$
$(value\_expr_1 \between value\_expr_2) \mid\urcorner (value\_expr_1 \between value\_expr_3)$

Заметим, что в отличие от параллельной композиции ‖ interlocked-композиция ⫳ не является ассоциативной.
Если выражение *value_expr* сходится и не вызывает взаимодействия и если $c_1 \neq c_2$, то справедливы следующие две эквивалентности:

$x := c_1? \between c_2! value\_expr \equiv$ **stop**

$x := c? \between c! value\_expr \equiv x := value\_expr$

Комбинатор interlocked-композиции иллюстрирует различие между внешним и внутренним выборами. Если выражения $value\_expr_1$ и $value\_expr_2$ сходятся и не вызывают взаимодействия и если $c_1 \neq c_2$, то имеют место следующие две эквивалентности:

$(x := c_1? \ \square \ c_2! value\_expr_2) \between c_1! value\_expr_1 \equiv x := value\_expr_1$

$(x := c_1? \ \square \ c_2! value\_expr_2) \between c_1! value\_expr_1 \equiv (x := value\_expr_1) \mid\urcorner$ **stop**

- **Последовательная композиция:**

$value\_expr_1$ **;** $value\_expr_2$

Второе выражение *value_expr* принуждается к выполнению после выполнения первого выражения *value_expr*. В качестве результата выполнения последовательной композиции возвращается значение второго выражения *value_expr*.
Единицей последовательной композиции является **skip**, кроме того, данная композиция ассоциативна и обладает свойством дистрибутивности по первому аргументу по отношению к внутреннему выбору:

$value\_expr$ **;** **skip** $\equiv value\_expr$

skip **;** $value\_expr \equiv value\_expr$

$value\_expr_1$ **;** $(value\_expr_2$ **;** $value\_expr_3) \equiv$
$(value\_expr_1$ **;** $value\_expr_2)$ **;** $value\_expr_3$

$(value\_expr_1 \mid\urcorner value\_expr_2)$ **;** $value\_expr_3 \equiv$
$(value\_expr_1$ **;** $value\_expr_3) \mid\urcorner (value\_expr_2$ **;** $value\_expr_3)$

Литература для дополнительного чтения

[ADL]	http://www.sun.com/960201/cover/language.html
[Eiffel]	http://www.eiffel.com
[iContract]	R.Kramer. iContract – The Java Design by Contract Tool. Fourth conference on OO technology and systems (COOTS), 1998.
[IFAD]	http://www.ifad.dk
[ISPRAS]	http://www.ispras.ru/
[Jones]	C.B. Jones. Systematic Software Development Using VDM. Prentice Hall International, 1986.
[Larch]	J.Guttag et al. The Larch Family of Specification Languages. IEEE Software, Vol. 2, No.5, September 1985, pp.24-36.
[OMG]	http://www.omg.org
[RAISE-language]	The RAISE Language Group. The RAISE Specification Language. Prentice Hall Europe, 1992.
[RAISE-method]	ftp://ftp.iist.unu.edu/pub/RAISE/method_book
[RUP]	http://www.rational.com
[VDM_SL]	N.Plat, P.G.Larsen. An Overview of the ISO/VDM-SL Standard. SIGPLAN Notes, Vol. 27, No. 8, August 1992.
[VDM++]	http://www.csr.ncl.ac.uk/vdm/
[MSC]	Message Sequence Charts. ITU recommendation Z.120.
[SDL]	Specification and Design Language. ITU recommendation Z.100.
[Z]	J.M.Spivey. The Z Notation: A Reference Manual. Prentice Hall, 2nd edition, 1992.

Приложение 1. Приоритет операций

Приоритет операций над значениями (по возрастанию)		
Приоритет	Операции	Ассоциативность
14	□ λ ∀ ∃ ∃!	Правая
13	≡ **post**	
12	□ ⫿ ‖ ╫	Правая
11	;	Правая
10	:=	
9	⇒	Правая
8	∨	Правая
7	∧	Правая
6	= ≠ > < ≥ ≤ ⊂ ⊆ ⊃ ⊇ ∈ ∉	
5	+ – \ ^ ∪ †	Левая
4	* / ° ∩	Левая
3	↑	
2	:	
1	~ prefix_op	

Приоритет типовых операций (по возрастанию)		
Приоритет	Operator(s)	Ассоциативность
3	$\xrightarrow{\mathbf{m}}$ $\xrightarrow{\sim}$ →	Правая
2	×	
1	**-set -infset** * ω	

Приложение 2. Таблица соответствия RSL символов и их ASCII представления

ASCII	RSL	ASCII	RSL	ASCII	RSL
><	×	isin	∈	~isin	∉
||	‖	++	╫	-\	λ
|=|	□	|^|	⫿	-list	*
**	↑	-inflist	ω	~=	≠
/\	∧	\/	∨	+>	↦
>=	≥	exists	∃	all	∀
<=	≤	union	∪	!!	†
inter	∩	<<	⊂	always	□
-m->	$\xrightarrow{\mathbf{m}}$	<<=	⊆	=>	⇒
-~->	$\xrightarrow{\sim}$	>>	⊃	is	≡
->	→	>>=	⊇	<->	↔
#	°	<.	⟨	.>	⟩
:-	•				

Используемые в ASCII представлении символов RSL слова all, exists, union, inter, isin, always являются зарезервированными и не могут использоваться в качестве идентификаторов.

Приложение 3. Ключевые слова RSL

Ключевые слова RSL			
Bool	**class**	**inds**	**skip**
Char	**do**	**initialise**	**stop**
Int	**dom**	**int**	**swap**
Nat	**elems**	**len**	**then**
Real	**else**	**let**	**tl**
Text	**elsif**	**local**	**true**
Unit	**end**	**object**	**type**
abs	**extend**	**of**	**until**
any	**false**	**out**	**use**
as	**for**	**post**	**value**
axiom	**forall**	**pre**	**variable**
card	**hd**	**read**	**while**
case	**hide**	**real**	**with**
channel	**if**	**rng**	**write**
chaos	**in**	**scheme**	

Указанные в таблице ключевые слова не могут использоваться в качестве идентификаторов.

Содержание