**Российская академия наук**
**Сибирское отделение**
**Институт систем информатики**
**им. А. П. Ершова**

**А. П. Стасенко**

# ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ СИСТЕМЫ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ SISAL 3.0

**Препринт**
**110**

**Новосибирск 2004**

В работе описывается система COM (Component Object Model) интерфейсов, называемая внутренним представлением IR1, которые задают язык промежуточного графового представления IF1 для системы функционального программирования (SFP) языка SISAL 3.0, являющегося расширением функционального языка SISAL-90. Объясняется использование IR1 для задания модуля SISAL 3.0 программы. Описаны «вершины» и «составные вершины» IF1 графа вместе с обоснованием изменений в некоторых «составных вершинах» оригинальной спецификации IF1. Для нетривиальных синтаксических конструкций языка SISAL 3.0 поясняется способ их задания во внутреннем представлении IR1.

**Siberian Division of the Russian Academy of Sciences**
**A. P. Ershov Institute of Informatics Systems**

**A. P. Stasenko**

# INTERNAL REPRESENTATION OF FUNCTIONAL PROGRAMMING SYSTEM SISAL 3.0

**Preprint**
**110**

**Novosibirsk 2004**

This work describes COM (Component Object Model) interface system, called Internal Representation IR1, representing Intermediate Form graph language IF1, created for System of Function Programming (SFP) SISAL 3.0 (successor of SISAL-90 functional language). The way to use IR1 for describing SISAL 3.0 program module is shown. Nodes and compound nodes used by IF1 graph are presented along with description and basing of introduced differences in IF1 specification for some compound nodes. For non-trivial syntactic constructions of SISAL 3.0 language the way of their representation in IR1 is explained.

## ВВЕДЕНИЕ[1]

Целью данной работы является описание графового внутреннего представления IR1, реализованного в виде COM (Component Object Model) интерфейсов, пригодного для задания модуля программы на языке SISAL 3.0. Работа выполняется в рамках проекта по созданию системы функционального программирования (SFP) SISAL 3.0, позволяющей прикладному программисту разрабатывать и отлаживать параллельную программу на своем рабочем месте с последующим ее исполнением на супервычислителе, доступном ему по сети. В таком проекте, состоящем из многих независимых частей, часто связанных только при помощи внутреннего представления, аспект продуманности структуры и способа реализации внутреннего представления становится весьма значимым.

При разработке внутреннего представления учитывались следующие существенные требования.

1. Машинная независимость как для представления параллелизма (нет явного разбиения вычислений на несколько потоков), так и для значений (независимость от разрядности машинной архитектуры) типов данных.
2. Полнота внутреннего представления, позволяющая произвести его ретрансляцию в семантически эквивалентную программу на исходном языке. Следует заметить, что это требование не влечёт за собой необходимость взаимно однозначного соответствия между синтаксическими конструкциями языка и объектами внутреннего представления. Две различные синтаксические конструкции языка со схожей семантикой могут быть выражены с помощью одинаковых объектов внутреннего представления. Это позволяет ввести некий базовый набор объектов внутреннего представления для задания неких обобщенных операций, вроде объекта, задающего обобщенную условную операцию и подходящего для описания выбирающих выражений с различным синтаксисом. Такой базовый набор операций значительно упрощает преобразования (машинно-независимые оптимизации) внутреннего представления. Но в то же самое время затруднятся задача ретрансляции внутреннего представления в эффек-

[1] Работа выполнена при финансовой поддержке Научной программы «Университеты России» (грант УР.04.01.027).

тивную программу на исходном языке программирования, поскольку часть информации в обобщенных объектах внутреннего представления, которая выражена явным образом в синтаксисе исходных конструкций языка, может быть представлена неявным образом. Например, может быть утеряно явно выделенное управляющее выражение синтаксической конструкции, типа оператора **switch** в языке C.

3. Возможность ретрансляции в синтаксически корректную программу после преобразований внутреннего представления, сохраняющих его корректность для исходного языка (см. разд. 4 для языка SISAL 3.0).
4. Лёгкость интерпретации (исполнения заданных внутренним представлением вычислений) без проведения дополнительных преобразований структуры внутреннего представления.
5. Структурированность объектов внутреннего представления для задания естественной вложенности одних конструкций исходного функционального языка в другие.
6. Все неявные действия над данными, например, преобразования типов, должны быть выражены явным образом с помощью объектов внутреннего представления.
7. Расширяемость, в смысле лёгкого введения новых объектов внутреннего представления для задания новых конструкций языков программирования и типов данных.

Существует несколько способов задания внутреннего представления, например, в этом качестве можно рассматривать дерево разбора транслируемой программы. Однако в нашем случае оно не удовлетворяет требованию интерпретируемости, так как это требование подразумевает наличие контекстной (семантической) информации. Вместе с этим, требование машинно-независимого параллелизма и функциональность представляемых языков программирования приводит нас к использованию потоковых графов как естественной основе структуры внутреннего представления.

Действительно, потоковые графы обладают рядом следующих полезных для требуемого внутреннего представления свойств.

1. Явно заданные информационные (семантические) связи (дуги) между операциями (вершинами) делают процесс интерпретации осуществимым без дополнительных преобразований. Это влечет отсутствие побочных эффектов вычислений (ввиду отсутствия понятия переменной) — естественного свойства чисто функциональных языков.

2. Параллелизм на уровне отдельных информационно независимых операций, не зависящий от машинной архитектуры.

Разд. 1 посвящён краткому описанию сути графового языка IF1 как основы внутреннего представления IR1, а также особенностям реализации IR1. Разд. 2 описывает COM интерфейсы внутреннего представления IR1, тогда как разд. 3 целиком отведён под особенности использования интерфейсов IR1 при задании модуля языка SISAL 3.0. В разд. 4 и 5 описаны семантики вычислений вершин IF1 графа, используемых во внутреннем представлении IR1.

## 1. ОТ IF1 К IR1

В связи с потребностью разработки транслятора функционального языка SISAL 3.0 [1, 2] появилась необходимость выработки удобного внутреннего представления для представления потока вычислений программ этого языка. Еще в 1985 г. был разработан язык IF1 (Intermediate Form) [3]. Этот язык основывается на ациклических иерархических графах и идеально подходит для поставленной цели в рамках языка SISAL 1.0.

При разработке языка IF1 основной задачей являлось создание графового языка, который мог бы стать результатом трансляции (front-end) компиляторов нескольких функциональных языков. С помощью такого графового языка становится возможным объединение и единые преобразования для программ, написанных с помощью разных функциональных языков. Однако, конечно, язык IF1 не достаточен для представления программ всех функциональных языков. Он в большой степени зависит от особенностей языков SISAL и VAL, но предполагается, что язык IF1 является достаточной основой для более обобщенных промежуточных представлений, требуемых для описания более широкого множества функциональных языков.

Программа на языке IF1 состоит из множества графов специального вида, среди которых выделено подмножество графов, соответствующих множеству функций исходной SISAL программы. Граф в IF1 состоит из объектов трёх видов: вершин (node, в английской терминологии IF1), дуг и рамки графа. Причём вершинам и рамке графа приписаны упорядоченные множества входов (в которые входят дуги, но не более одной для одного входа) и выходов (из которых выходят дуги). Входы рамки графа рассматриваются как выходы (результаты вычислений) графа, а выходы рамки графа как входы (параметры) графа. Тем самым, рамку графа можно рассматривать как «вывернутую наизнанку» вершину. В дальнейшем, как и в оригиналь-

ной терминологии IF1 входы могут называться входными портами, а выходы — выходными портами.

Каждой дуге графа приписан тип (если IF1 задаёт строго типизированный язык) пересылаемого значения. Существует, однако, специальный вид дуги, обозначающий литерал (константу) любого типа. Эти дуги не исходят ни из какого выхода, но имеют дополнительное свойство, передающее непосредственное значение литерала в какой-либо входной порт.

Вершины обозначают операции над своими входами (аргументами), результаты которых находятся на выходах вершины. Вершины бывают простыми и составными. Простые вершины (или просто вершины) не имеют внутренней структуры помимо ассоциированной с ними операции. Составные вершины дополнительно содержат упорядоченное множество графов. Их количество и все связи между входами (выходами) составной вершины и входами (выходами) этих графов задаются типом (или семантикой) операции составной вершины. Структурированность вычислений, задаваемых IF1 графом, основывается на иерархичности IF1 графов, заданной посредством графов составной вершины.

Тем самым, IF1 задаёт поток вычислений без какого-либо дополнительного управления и исключительных ситуаций. Поэтому предполагается наличие дополнительного ошибочного значения для каждого используемого типа данных, возвращаемого вершинами операций, определённых не для всех значений своих аргументов (например, деление на нуль).

Очевидно, что вычисления, заданные подобным образом, определяют частичный порядок над общей последовательностью выполнения вершин-операций одного IF1 графа. Не сравнимые между собой операции можно выполнить параллельно, что позволяет естественным (не зависящим от машинной архитектуры) образом задать модель параллельных вычислений, причём на очень низком уровне — уровне отдельной операции. К тому же IF1 граф допускает лёгкую интерпретируемость (исполнение), так как он задаёт потоковую модель вычислений, для исполнения которой (с некоторыми ограничениями) можно даже использовать суперкомпьютеры с потоковой архитектурой.

Реализация рассмотренных сущностей графового языка IF1 с помощью интерфейсов классов C++ (реализованных с помощью технологии COM [4]) получила название IR1 (Internal Representation). В этих интерфейсах были учтены моменты, необходимые для реализации нововведений, появившихся в SISAL 3.0 (далее просто SISAL). Прежде всего, к ним относятся модульная структура и задаваемые пользователем редукции. Однако сразу

надо отметить некоторые особенности системы введенных интерфейсов, реализующих объекты графа IF1.

Во-первых, было принято решение отказаться от выделения дуги в качестве отдельной сущности (интерфейса) и переложить связи между портами на интерфейс порта. Типы передаваемых дугами значений также поддерживаются при помощи интерфейса порта. Такое решение позволило упростить реализацию обхода графа, так как для любого взятого порта мы сразу можем определить порт, поставляющий ему значения, и множество портов, которые, в свою очередь, принимают от него значения. Это, например, позволило легко реализовать операцию удаления порта из внутреннего представления. В дальнейшем, термин дуга, однако, будет употребляться при описании взаимосвязей между портами.

Во-вторых, было принято решение не делать различных интерфейсов для реализации входных и выходных портов. Это позволило рассматривать единый интерфейс порта с этих двух точек зрения одновременно, что в свою очередь позволило унифицировать сущности графа и рамки графа с точки зрения того, какую группу портов считать входными и выходными портами. Теперь один и тот же порт может служить выходом графа и в то же время являться входом для дуг, принадлежащих этому графу (и наоборот). Такая унификация сделала возможным рассматривать граф в качестве вершины в «объемлющем» графе, ничего не знающем о внутреннем устройстве такой вершины.

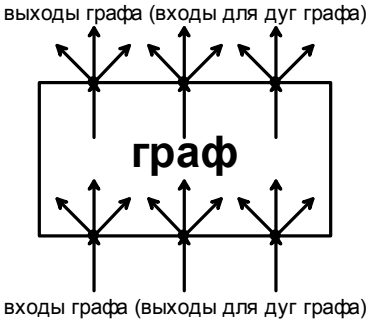



*Рис. 1.* Входы и выходы графа

Тем самым, можно определить IR1 граф как объект, содержащий упорядоченные множества входных (выходов для дуг графа) и выходных (входов для дуг графа) портов и множество (с задаваемым в некоторых случаях порядком) IR1 графов, являющихся вершинами этого графа. Причем эти графы иногда будут называться подграфами относительно родительского графа, их содержащего.

Представим составную вершину как IR1 граф без дуг, содержащий в качестве вершин IR1 графы, соответствующие графам составной вершины, а вместо специального типа дуги, обозначающей литерал в IF1, введём вершину с одним выходом, которая имеет специальное свойство — значение литерала. Тогда с помощью одного интерфейса IR1 графа можно реализовать четыре сущности языка IF1: граф, вершина, составная вершина и специальный тип дуги для литерала.

Таким образом, внутреннее представление IR1 модуля языка SISAL можно представить в виде множества IR1 графов с пометками, соответствующих множеству функций и редукций исходной SISAL программы, и множества именованных типов, которые будут экспортированы из модуля SISAL программы. К пометкам IR1 графа относятся имя функции или редукции, которую он задаёт, и флаг её экспортируемости. Внутреннее представление всей программы на языке SISAL состоит из множества внутренних представлений IR1 для отдельных модулей программы.

Далее, если не указано обратного, под термином граф подразумевается IR1 граф.

## 2. ОПИСАНИЕ СИСТЕМЫ ИНТЕРФЕЙСОВ IR1 (IR1.DLL)

В этом разделе будет рассмотрена система интерфейсов IR1, необходимая для задания модуля программы. Вся специфика, свойственная представлению модуля SISAL программы, рассматривается в следующем разделе.

При разработке системы интерфейсов для представления модуля программы, представленной ниже, преследовалась цель сделать её максимально унифицированной, что должно значительно облегчить её преобразование к другим формам (включая визуализацию) и упростить процесс трансляции с других языков функционального программирования (кроме SISAL), а также уменьшить общее количество необходимых интерфейсов.

Унификация, во-первых, заключается в использовании динамического множества строковых и числовых свойств, присущих каждому интерфейсу

внутреннего представления вместо введения фиксированных свойств, доступных через методы более широкого множества фиксированных интерфейсов. Это позволяет унифицировать многие операции над объектами внутреннего представления, например, поиск объекта в коллекции объектов по его свойству. Также это позволяет пользователю внутреннего представления вводить свои свойства (пометки) объектов. Такие пометки могут оказаться удобными при обходе объектов внутреннего представления без необходимости введения дополнительных структур данных.

Самый существенный недостаток динамического списка свойств — это логарифмическая временная сложность (от количества свойств во множестве) доступа к значению свойства по идентификатору свойства. Однако в нашем случае данная оценка временной сложности улучшается тем, что по построению транслятором количество свойств каждого объекта внутреннего представления ограниченно заданной константой. Эта константа зависит от специализации данного внутреннего представления при задании программ на конкретном языке программирования (для языка SISAL 3.0 описание специализации приведено в разд. 3).

Во-вторых, вместо отдельных интерфейсов для литералов, вершин и составных вершин используется интерфейс графа, что вместе с динамическим списком свойств позволяет также, например, отказаться от таких более ёмких, чем граф, интерфейсов, как функция или редукция.

Внутреннее представление IR1 реализуется с помощью интерфейсов, объявленных в библиотеке типов (IDL type library), которая находится в динамически подключаемой библиотеке ir1.dll вместе с реализацией этих интерфейсов. Использованная для реализации интерфейсов технология COM совместно со стандартным языком (IDL) их описания позволяет абстрагироваться от языка программирования и окружения, на котором будут написаны программы, работающие с данным внутренним представлением. В частности, интерфейсы внутреннего представления полностью совместимы со скриптовыми языками, вроде JavaScript и VBScript.

### 2.1. Основные интерфейсы внутреннего представления

На рис. 2 приведена схема основных интерфейсов внутреннего представления. Причём у них указаны только существенные с точки зрения дальнейшего рассмотрения методы. Методы интерфейсов, расположенные в первой части интерфейса, называются атрибутами. Атрибут может иметь флаг «[read-only]» означающий то, что значение атрибута доступно только для чтения и не может быть изменено непосредственным присваиванием.

Интерфейсы *IUknown* и *IDispatch* являются стандартными и здесь не рассматриваются.

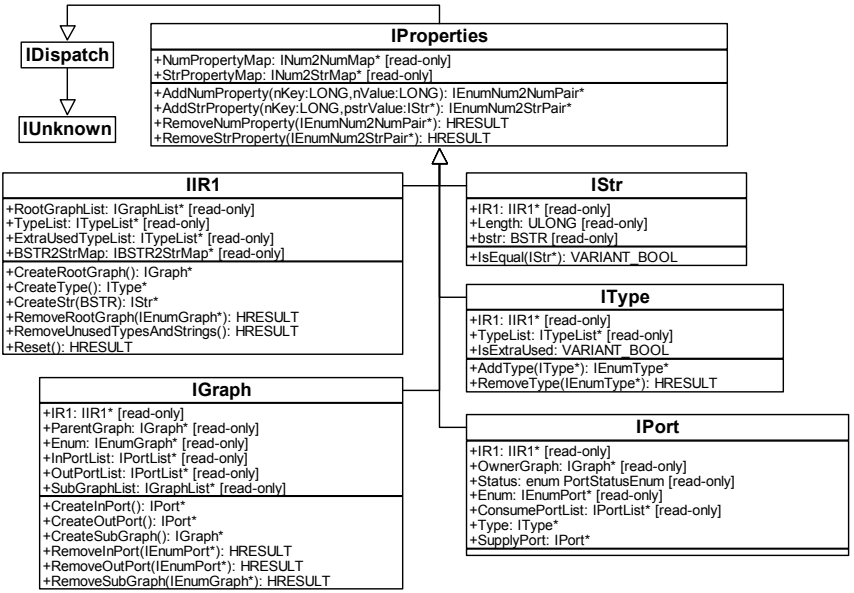



*Рис. 2.* Диаграмма интерфейсов IR1

### *2.1.1. Интерфейс числовых и строковых свойств*

Остановимся подробнее на интерфейсе *IProperties*, который является базовым для всех остальных интерфейсов объектов внутреннего представления. С помощью атрибутов *IProperties→NumPropertyMap* и *IProperties → StrPropertyMap* можно получить доступ к множествам числовых и строковых свойств объекта внутреннего представления соответственно. Множество строковых свойств задаётся интерфейсом *INum2StrMap*, а числовых свойств — *INum2NumMap*. Свойство задаётся с помощью пары {ключ свойства, значение свойства}, где ключ свойства уникален в одном множестве свойств. Строковым свойствам соответствует интерфейс *INum2StrPair*, а числовым — *INum2NumPair*. Пары во множестве при их перечислении упорядочены по неубыванию значения ключа свойства.

Тем самым, множество свойств можно рассматривать как отображение из множества ключей свойств во множество их значений. В дальнейшем, при обозначении имен ключей строковых свойств будем использовать префикс *sk* (string key), а для числовых свойств — *nk* (numerical key). Интерфейс *IProperties* также позволяет добавлять новые свойства (методы *IProperties → AddNumProperty()* и *IProperties → AddStrProperty()*) и удалять уже существующие (методы *IProperties → RemoveNumProperty()* и *IProperties → RemoveStrProperty()*).

Может возникнуть вопрос о машинной независимости внутреннего представления ввиду наличия ключей свойств и значений числовых свойств типа LONG, являющегося, строго говоря, машинно-зависимым. Однако тот факт, что числовые свойства не используются для задания значений типов языка программирования, программы которого задаются внутренним представлением, позволяет использовать данный тип данных (при условии, что его длина не менее 32 бит). Скалярные значения языка программирования, задаваемого внутренним представлением, хранятся в машинно-независимом строковом виде.

Строки строковых свойств объектов внутреннего представления также имеют определённую машинную независимость, ввиду их задания с помощью последовательностей UNICODE символов и выделенной длине строки (для возможности задания произвольных строк, не имеющих выделенного конечного символа).

### *2.1.2. Интерфейс внутреннего представления*

Рассмотрим основной интерфейс внутреннего представления *IIR1*, служащий «фабрикой классов» для всех остальных интерфейсов объектов внутреннего представления: корневых графов (создаются с помощью метода *IIR1→CreateRootGraph()*), типов (*IIR1→CreateType()*) и строк (*IIR1→CreateStr()*).

Корневым графом называется граф, не являющийся подграфом другого графа, например, функция или редукция для языка SISAL. Все внутренние объекты графа (порты и подграфы) создаются и локально хранятся с помощью методов интерфейса родительского графа, который будет рассмотрен ниже. Это позволяет упростить процесс навигации по ним, а также упростить процесс их последующего освобождения при освобождении родительского графа.

В то же время объект интерфейса *IIR1* хранит список созданных им корневых графов (атрибут *IIR1→RootGraphList*), типов (атрибут *IIR1→TypeList*) и множество созданных строк (атрибут

*IIR1→BSTR2StrMap*). Элементом множества строк является пара {ключ строки, указатель на интерфейс строки}, задаваемая интерфейсом *IBSTR2StrPair*. Где ключ строки — это строка типа BSTR, которая однозначно идентифицирует пару в этом множестве. Если строка, указанная методом *IIR1→CreateStr()*, уже существует во множестве, то новая *IStr* строка не создаётся и клиенту возвращается указатель на уже существующую *IStr* строку.

Пары множества *IIR1→BSTR2StrMap* при перечислении упорядочены в лексикографическом порядке значений ключей свойств. Организация коллекции уже созданных строк в виде множества, а не списка, хотя и приводит к логарифмической сложности операции добавления новой строки, но гарантирует отсутствие повторяющихся строк, что позволяет для сравнения *IStr* строк одного внутреннего представления сравнивать всего лишь указатели на их интерфейсы.

С помощью метода *IIR1→RemoveRootGraph()* допускается удаление указанного уже созданного корневого графа, в то время как удалить конкретный уже созданный тип или строку невозможно. Допускается лишь массовое удаление неиспользуемых типов и строк методом *IIR1→RemoveUnusedTypesAndStrings()*. Тип считается неиспользуемым, если он не используется каким-либо портом графа внутреннего представления (и не является подтипом используемого типа) и не принадлежит списку *IIR1→ExtraUsedTypeList*. Строка считается неиспользуемой, если она не используется в каком-либо строковом свойстве объекта внутреннего представления и не является строковым свойством используемой строки.

Метод *IIR1→Reset()* удаляет все созданные объекты внутреннего представления, возвращая объект интерфейса *IIR1* в первоначальное состояние. При удалении объекта внутреннего представления все его методы, доступные через оставшиеся указатели на его интерфейс, ничего не делают и возвращают код ошибки E_FAIL.

### *2.1.3. Интерфейс строки*

Далее рассмотрим интерфейс *IStr*, служащий для хранения строки внутреннего представления, который используется как значение строкового свойства в объектах внутреннего представления. Указатель на объект этого интерфейса для нужной строки типа BSTR можно получить при помощи метода *IIR1→CreateStr()*. После этого изменить строку, хранящуюся в *IStr* строке, невозможно. Однако можно узнать её длину через атрибут *IStr→Length* и получить её в виде BSTR строки через атрибут *IStr→bstr*. Через атрибут *IR1* можно узнать указатель на интерфейс *IIR1*, создавший

данную строку. Метод *IStr→IsEqual()* служит для сравнения двух *IStr* строк. Для строк одного внутреннего представления временная сложность данного метода равна константе.

### *2.1.4. Интерфейс типа*

Интерфейс *IType* используется для задания произвольного типа, который применяется для типизации передаваемых портами значений. Для передачи структуры агрегатных типов (например: массив, запись, и. т. д.) этот интерфейс использует очень простое решение — напрямую редактируемый список типов, на которых основывается данный тип. Для него определены методы добавления в конец нового типа (*IType→AddType()*) и удаления указанного типа (*IType→RemoveType()*).

Сам список можно получить с помощью атрибута *IType→TypeList*. Элементы этого списка далее иногда называются подтипами данного типа. Это вместе с механизмом динамического множества свойств интерфейса *IProperties* даёт достаточно мощное средство для описания широкого класса типов, например, так могут быть описаны все типы языка SISAL.

Интерфейс *IType* допускает рекурсивно определяемые типы, например, множества типов языка SISAL:

**type** integer_arrays = [integer, **array**[integer_arrays]].

Рекурсивно определенные типы не являются чем-то исключительным для чисто функционального программирования, ведь отсутствие понятия указателя вынуждает, например, задавать такие распространенные объекты, как списки, не с помощью указателя на следующий элемент списка, а с помощью рекурсии.

Возможность рекурсивного определения типа повлекла за собой реализацию списка подтипов в виде списка «слабых» указателей на интерфейс *IType*, не увеличивающих счетчик ссылок интерфейса *IUnknown*. Иначе, если бы хранились «сильные» указатели, циклические последовательности типов при освобождении всех ссылок на её элементы извне приводили бы к «утечке» памяти. Возможным решением, конечно, было бы отслеживание таких циклов и соответствующая модификация методов

*IUnknown→Release()* и *IUnknown→AddRef()*

для подсчёта количества внешних ссылок на цикл. Но подобное отслеживание циклических зависимостей привело бы к не оправданным временным расходам при вызове метода *IType→AddType()*.

Решение хранить «слабые» указатели на подтипы, в свою очередь, определило отсутствие метода удаления указанного типа в интерфейсе *IIR1*.

Это невозможно, ввиду вероятности наличия «слабых» указателей на удаляемый тип, содержащихся в списке подтипов других типов. После удаления эти указатели утратили бы смысл. Единственное решение этой проблемы, заключающееся в поддержании интерфейсом типа множества типов, в отношении которых данный тип является подтипом, было признано нерациональным. Поэтому было решено ввести общее удаление "неиспользуемых" типов и строк (по аналогичным соображениям)

*IIR1→RemoveUnusedTypesAndStrings()*.

Булевский атрибут *IType→IsExtraUsed* равен VARIANT_FALSE после создания типа. Установка значения этого атрибута в VARIANT_TRUE гарантирует, что данный тип не будет удален в результате работы метода *IIR1→RemoveUnusedTypesAndStrings()*. Список типов внутреннего представления с атрибутами *IType→IsExtraUsed*, равными VARIANT_TRUE, находится в атрибуте *IIR1→ExtraUsedTypeList*. Атрибут *IType→IsExtraUsed*, равный VARIANT_TRUE, может использоваться для типов экспортируемых из модуля — они могут и не применяться в портах графов внутреннего представления.

### *2.1.5. Интерфейс порта*

Интерфейс порта *IPort* служит для организации потоков данных между операциями, задаваемыми графами внутреннего представления. Порт характеризуется своим типом (атрибут *IPort→Type*) и другим портом, от которого он получает данные (атрибут *IPort→SupplyPort*). В то же время реализация этого интерфейса постоянно поддерживает список портов (атрибут *IPort→ConsumePortList*), которым данный порт поставляет данные. Это множество значительно упрощает некоторые действия над портами и связями между ними. Объект интерфейса порта порождается с помощью методов интерфейса графа *IGraph→CreateInPort()* и *IGraph→CreateOutPort()*.

Атрибут *IPort→Enum* возвращает итератор списка входных или выходных портов графа-владельца (атрибут *IPort→OwnerGraph*) рассматриваемого порта. Причём возвращаемый итератор находится на текущем порте. Атрибут *IPort→Status* позволяет определить, является ли порт входным или выходным с точки зрения его графа-хозяина.

Унифицировав вход и выход графа в одном интерфейсе порта, мы, однако, не избавились от необходимости знать, является ли он входом или выходом графа (атрибут *IPort→Status*). Эта информация требуется для поддержания корректности создаваемой дуги графа (атрибута *IPort→SupplyPort*), которую можно провести только в следующих случаях.

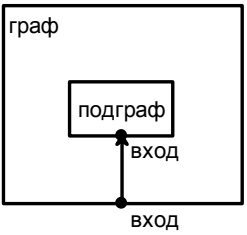
<table>
<tr><th rowspan="2">Текущий порт это —</th><th colspan="2">Порт-поставщик (IPort→SupplyPort) это —</th></tr>
<tr><th>вход</th><th>выход</th></tr>
<tr><td>вход</td><td> 

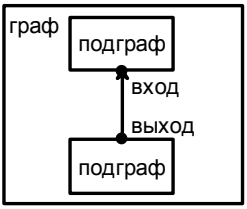
Рис. 3a<br>Граф IPort → OwnerGraph → IGraph → ParentGraph текущего порта должен равняться графу IPort→OwnerGraph порта-поставщика.</td><td> 

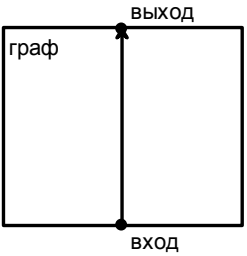
Рис. 3b<br>Графы IPort → OwnerGraph → IGraph → ParentGraph обоих портов должны совпасть, но не равняться NULL.</td></tr>
<tr><td>выход</td><td> 

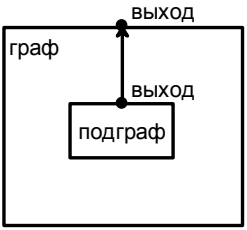
Рис. 3c<br>Графы IPort→OwnerGraph обоих портов должны совпасть.</td><td> 

Рис. 3d<br>Граф IPort→OwnerGraph текущего порта должен равняться графу IPort → OwnerGraph → IGraph → ParentGraph порта-поставщика.</td></tr>
</table>

Без этой проверки были бы возможны, например, следующие ситуации, изображенные на рис. 4a, 4b и 4c.

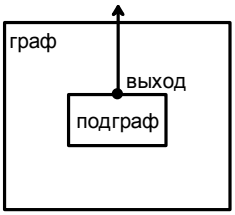



*Рис. 4a*

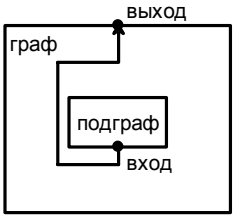



*Рис. 4b*

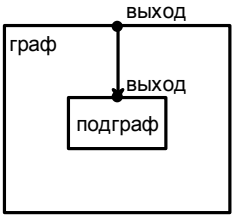



*Рис. 4c*

### *2.1.6. Интерфейс графа*

Интерфейс графа *IGraph* характеризуется своими входными (атрибут *IGraph→InPortList*) и выходными портами (атрибут *IGraph→OutPortList*), а также своими подграфами (атрибут *IGraph→SubGraphList*). Граф может как создавать свои входные (методом *IGraph → CreateInPort()*) и выходные (методом *IGraph → CreateOutPort()*) порты и свои подграфы (методом *IGraph → CreateSubGraph()*), так и удалять их.

При удалении порта из графа методами *IGraph→RemoveInPort()* и *IGraph→RemoveOutPort()* все порты, которым он поставлял данные, разрывают с ним связь (их атрибут *IPort→SupplyPort* становится равным NULL). При удалении подграфа методом *IGraph→RemoveSubGraph()* его входные и выходные порты разрывают связи аналогичным образом.

Атрибут *IGraph→Enum* возвращает итератор списка подграфов (атрибут *IGraph→SubGraphList*) родительского графа (атрибут *IGraph→ParentGraph*), находящийся на текущем графе. В случае если граф является корневым (его атрибут *IGraph→ParentGraph* равен NULL), то возвращается итератор списка корневых графов (атрибут *IIR1→RootGraphList*) внутреннего представления, которому принадлежит текущий граф (атрибут *IGraph→IR1*).

С помощью интерфейса графа *IGraph* можно реализовать широкий класс объектов внутреннего представления, например, в терминологии IF1 v1.0: графы, вершины, составные вершины, литералы, а также новые объекты — графы задаваемых пользователем редукций.

## 2.2. Интерфейсы коллекций

Далее рассмотрим интерфейсы коллекций объектов внутреннего представления, упомянутые выше. Их общая диаграмма приведена на рис. 5.

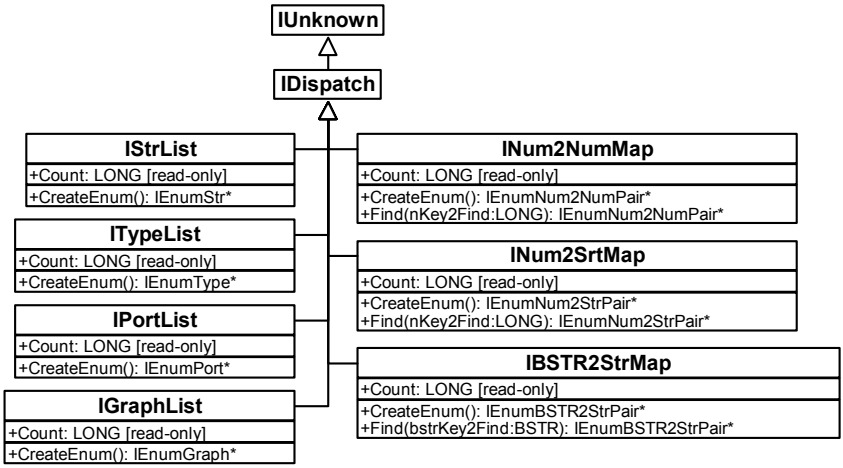



*Рис. 5.* Диаграмма интерфейсов коллекций объектов IR1

Видно, что интерфейсы списков строк (*IStrList*), типов (*ITypeList*), портов (*IPortList*) и графов (*IGraphList*) позволяют лишь узнать количество элементов в списке (атрибут *Count*) и получить указатель на интерфейс итератора (атрибут *CreateEnum*), который находится на первом элементе коллекции, или за ним, если такового нет.

Интерфейсы конечных отображений чисел в числа (*INum2NumMap*), чисел в *IStr* строки (*INum2StrMap*) и BSTR строк в *IStr* строки (*IBSTR2StrMap*) дополнительно определяют метод *Find()*. Этот метод осуществляет поиск элемента в области значений отображения по заданному элементу из множества определения отображения. Причём данный метод имеет логарифмическую временную сложность, которая зависит от мощности множества определения отображения.

Интерфейсы рассмотренных коллекций не имеют методов прямого доступа к своим элементам по их порядковому номеру. Эту операцию невозможно реализовать за константное время для применяемых реализаций коллекций в виде списков и отображений. Отсутствие подобного метода по-

зволяет избежать неэффективных ситуаций, например, перебора элементов коллекции с его помощью.

Операция добавления (равно как и удаления) элемента не присутствует в интерфейсах коллекций. Методы добавления и удаления элемента коллекции вынесены в интерфейсы внутреннего представления, являющиеся хозяевами этих коллекций. Это было сделано с целью проведения дополнительных проверок поступающих данных и обновления дополнительных внутренних структур реализации этих интерфейсов.

Следует отметить, что операция удаления элемента коллекции с помощью итератора имеет константную временную сложность и приводит к порче всех других итераторов этой коллекции, указывающих на удаляемый элемент. Итератор, с помощью которого было произведено удаление, переводится на следующий элемент коллекции.

### 2.3. Вспомогательные интерфейсы

На рис. 6 приведена диаграмма всех интерфейсов итераторов и пар, использованных выше.

Интерфейсы итераторов не соответствуют стандартным ATL [5] интерфейсам семейства *IEnumXXX*. С помощью метода *Next()* теперь можно получить только один следующий элемент коллекции. Это «ограничение» связано с неприспособленностью стандартного метода *Next()* для скриптовых языков.

Представляется целесообразным введение атрибута *Item*, равного текущему элементу коллекции. Его использование в некоторых случаях весьма уместно и позволяет избежать не интуитивного дублирования итератора методом *Clone()*. Предположим, что после анализа элемента коллекции, полученного методом *Next()*, потребуется удалить его из коллекции. В этом случае его итератор, с помощью которого указывается удаляемый элемент, можно эффективно получить, только если предварительно он был продублирован методом *Clone()*. Гораздо естественнее не переходить к следующему элементу коллекции, пока всё ещё требуется итератор текущего элемента.

Синтаксис и семантика методов *Skip()*, *Reset()* и *Clone()* соответствуют аналогичным методам стандартных итераторов.

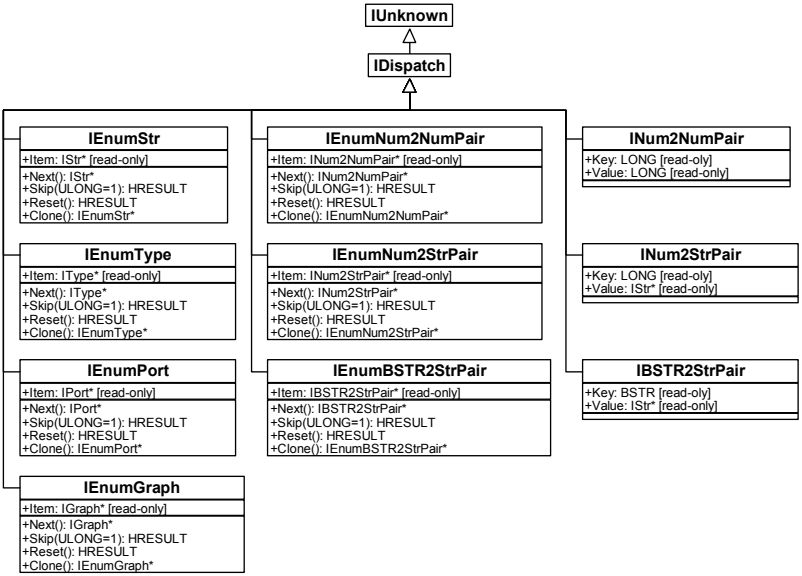



***Рис. 6.*** Диаграмма интерфейсов итераторов для коллекций объектов IR1 и пар {LONG, LONG}, {LONG, IStr*}, {BSTR, IStr*}

## 3. СПЕЦИАЛИЗАЦИЯ IR1 ДЛЯ ЯЗЫКА SISAL 3.0 (IR1_NAMES.DLL)

В этом разделе будут рассмотрены числовые и строковые свойства интерфейсов внутреннего представления, их возможные значения, а также другие соглашения, необходимые для задания внутреннего представления модуля SISAL программы.

Любой тип, порт или граф внутреннего представления может иметь строковое свойство *skFileName*, равное пути к файлу, в котором находилось определение данного объекта в момент трансляции текущего внутреннего представления. Если такое строковое свойство присутствует, то данный объект внутреннего представления может дополнительно иметь числовое свойство *nkFileLine*, равное номеру строки, который уточняет местоположение объекта. Местоположение объекта может быть дополнительно уточ-

нено с помощью числового свойства *nkFileLinePos*, равного номеру позиции в указанной строке.

Тип языка SISAL 3.0 задаётся с помощью интерфейса *IType*, объекты которого дополнительно имеют числовое свойство *nkTypeTag*. Значения этого числового свойства расшифровываются в следующей таблице.

| **Значение числового свойства *nkTypeTag*** | **Тип языка SISAL 3.0** | **Содержимое списка подтипов *IType→TypeList*** |
|---|---|---|
| ***ttNull*** | *null* | Список подтипов пуст. |
| ***ttBoolean*** | *boolean* | Список подтипов пуст. |
| ***ttCharacter*** | *character* | Список подтипов пуст. |
| ***ttInteger*** | *integer* | Список подтипов пуст. |
| ***ttReal*** | *real* | Список подтипов пуст. |
| ***ttDouble*** | *double* | Список подтипов пуст. |
| ***ttAnyRecord*** | *Всевозможные записи* **record**. | Список подтипов пуст. |
| ***ttAnyUnion*** | *Всевозможные союзы* **union**. | Список подтипов пуст. |
| ***ttAnyFunction*** | *Всевозможные функции* **function**. | Список подтипов пуст. |
| ***ttNamed*** | *Переименование типа*. Дополнительно имеется строковое свойство *skTypeName*, равное имени типа. Задаётся синтаксической конструкцией: «**type** <имя типа> = <базовый тип>». | Содержит один тип, который является базовым типом переименованного типа. |
| ***ttCustom*** | *Пользовательский тип*. Дополнительно имеется строковое свойство *skTypeName*, равное имени типа. Задаётся синтаксической конструкцией: «**type** <имя типа> := <базовый тип>». | Содержит один тип, который является базовым типом пользовательского типа. |

| | | |
|---|---|---|
| ***ttArray*** | *Массив* **array**[…]. | Содержит один тип элемента массива. |
| ***ttStream*** | *Поток* **stream**[…]. | Содержит один тип элемента потока. |
| ***ttRecord*** | *Запись* **record**[…]. | Содержит типы полей записи в естественном порядке их следования. Каждый тип поля записи имеет дополнительное строковое свойство *skRecordFieldName*, обозначающее имя этого поля. |
| ***ttUnion*** | *Союз* **union**[…]. | Содержит типы тегов союза в естественном порядке их следования. Каждый тип тега союза имеет дополнительное строковое свойство *skUnionTagName*, обозначающее имя этого тега. |
| ***ttFunction*** | *Функция* **function**[… **returns** …]. | Содержит два указателя на интерфейс типа *IType* без числового свойства *nkTypeTag*. Типы аргументов функции находятся в списке подтипов первого типа, а типы результатов функции находятся в списке подтипов второго типа. Типов аргументов и результатов функции перечисляются в естественном порядке. |
| ***ttReduction*** | *Редукция*. Используется только в типе выходного порта графа литерала. | Содержит три указателя на интерфейс типа *IType* без числового свойства *nkTypeTag*. Типы начальных параметров редукции находятся в списке подтипов первого типа. Типы циклических параметров редукции находятся в списке подтипов второго типа, а типы результатов редукции находятся в списке подтипов третьего типа. Типы параметров и результатов редукции перечисляются в естественном порядке. |
| ***ttSet*** | *Множество типов* {…}. | Содержит типы, принадлежащие множеству типов. |

Порты графов внутреннего представления могут иметь строковое свойство *skPortName*, содержащее имя значения, приходящего в данный порт из порта поставщика, который можно получить через атрибут *IPort→SupplyPort*.

Как было уже неоднократно сказано, с помощью интерфейса графа реализуется широкий спектр понятий IF1: литералы, вершины, составные вершины и IF1 графы. Понять, какого рода объект реализован с помощью этого интерфейса, можно, анализируя значение его числового свойства *nkGraphTag*, расшифровка которого приведена в следующей таблице.

| **Значение числового свойства *nkGraphTag*** | **Описание объекта внутреннего представления** | **Содержимое списка подграфов *IGraph → SubGraphList*** |
|---|---|---|
| ***gtLiteral*** | *Литерал*. Граф имеет ровно один выходной порт и ни оного входного порта. Тип литерала определяется по типу выходного порта. Алгоритм определения значения литерала объясняется после текущей таблицы. | Список подграфов пуст. |
| ***gtFunction*** | *Функция*. Граф имеет строковое свойство *skFunctionName*, задающее имя функции. Типы входных портов графа соответствуют типам аргументов функции. Типы выходных портов — типам результатов функции. Входные порты графа имеют строковое свойство *skPortName*, значение которого равно имени соответствующего формального параметра функции. | Список подграфов задаёт неупорядоченное (порядок, инициированный списком, несущественен) множество вершин графа функции. |

| | | |
|---|---|---|
| ***gtReduction*** | *Редукция*. Граф имеет строковое свойство *skReductionName*, задающее имя редукции. Типы входных портов графа соответствуют типам начальных и циклических параметров редукции. Типы выходных портов — типам результатов редукции. Входные порты графа имеют строковое свойство *skPortName*, значение которого равно имени соответствующего параметра редукции. | Список подграфов содержит три упорядоченных подграфа, соответствующих инициализации, телу и предложению возврата редукции. Эти графы не имеют числового свойства *nkGraphTag*, а список их подграфов задаёт неупорядоченное множество их вершин. |
| ***gtSelect*, *gtLoopA*, *gtLoopB*, *gtLoopA1*, *gtLoopB1* и *gtForAll*** | *Составная вершина*. | Список подграфов задает упорядоченную последовательность IF1 графов составной вершины. Каждый из этих графов имеет числовое свойство *nkSubGraphOrd*, значение которого равно порядковому номеру графа в последовательности. Граф, принадлежащий данной последовательности, не имеет числового свойства *nkGraphTag*. Список подграфов графа задаёт неупорядоченное множество его вершин. |
| ***все остальное*** | *Вершина графа*. | Список подграфов пуст. |

Если с помощью интерфейса графа реализуется литерал, то его значение определяется приведенной ниже последовательностью действий.

1. Если граф имеет числовое свойство nkErrorLiteral (равное произвольному значению), то литерал считается ошибочным значением своего типа.
2. Если тип литерала является функциональным или переименованием функционального типа, то у графа есть строковые свойства skFunctionName и skModuleName, содержащие имя функции и имя модуля, в котором эта функция определена, соответственно.
3. Если тип литерала является редукцией, то у графа есть строковые свойства skReductionName и skModuleName, содержащие имя ре-

дукции и имя модуля, в котором эта редукция определена, соответственно.

4. Если тип литерала является булевским или переименованием булевского типа, то граф имеет числовое свойство *nkBooleanLiteral*. Значение свойства не равно нулю, если булевское значение равно **true**, и равно нулю иначе.
5. Если ни один из предыдущих случаев не выполняется, то значение литерала задаётся строковым свойством *skLiteralValue*.
    - Для символьного литерала это будет строка, состоящая из одного символа.
    - Целочисленные литералы заданы последовательностью шестнадцатеричных цифр с возможным знаком.
    - Вещественные числа представлены строкой в экспоненциальной форме.
    - Строковые литералы заданы непосредственно.

Внутреннее представление разобранного модуля языка SISAL представляет собой указатель на интерфейс IIR1, содержащий в качестве корневых графов (атрибут *IIR1→RootGraphList*) графы функций и редукций. Граф импортируемой в данный модуль функции или редукции не содержит подграфов (атрибут *IGraph→SubGraphList*) и имеет дополнительное строковое свойство *skModuleName*, означающее имя модуля, из которого производилось импортирование. Графы экспортируемых функций и редукций имеют числовое свойство *nkExtern*, которое может быть равно произвольному значению.

Аналогично функциям, типы модуля (атрибут *IIR1→TypeList*) имеют строковое свойство *skModuleName*, если они были импортированы из модуля с таким именем, и числовое свойство *nkExtern*, если они определяются в текущем модуле и предполагаются быть экспортированы из него. Во втором случае типы модуля также присутствуют в списке *IIR1→ExtraUsedTypeList*. Нужно также отметить, что экспортируемый тип всегда является именованным или пользовательским типом. Поэтому данный тип содержит строковое свойство *skTypeName*, равное имени типа, которое используется для экспорта этого типа.

Внутреннее представление имеет строковое свойство *skModuleName*, содержащее имя модуля, которое оно задаёт. Имеется также строковое свойство *skSourceLanguageName*, равное «SISAL 3.0».

## 4. ОПИСАНИЕ «ВЕРШИН» IR1 ДЛЯ SISAL 3.0

В этом разделе будут использованы обозначения последовательности портов вершин, которые приведены в следующей таблице.

| Обозначение | Описание обозначения |
|---|---|
| (**T**)*$^{n}$ | n портов типа **T**, где n≥0. |
| (**T**)+$^{n}$ | n портов типа **T**, где n≥1. |
| **array**$^{n}$ | Массив размерности n. |
| [**T**] | Нуль или один порт типа **T**. |
| {$\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_n}$} | Порт с типом $\mathbf{T_i}$, где i от 1 до n. |
| $\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_n}$ | Последовательность портов с типами $\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_n}$. |

Перед рассмотрением семантики вершин стоит отметить, что любые два порта, соединенные дугой, имеют одинаковый тип (атрибут *IPort→Type*). Все допустимые преобразования типов выполняются явно с помощью соответствующих вершин.

Ниже приведён шаблон описания вершины.

| | |
|---|---|
| **Тип вершины** | Здесь находится значение числового свойства *gtGraphTag* объекта интерфейса графа *IGraph*. |
| **Порты** | <типы входных портов вершины> → <типы выходных портов вершины> |
| **Семантика** | Здесь находится описание преобразования значений типов, присущих входным портам вершины, в значения типов выходных портов вершины. |

### 4.1. Преобразование типов

| **Тип вершины** | *gtCustomType* |
|---|---|
| **Порты** | $\mathbf{T_1} \rightarrow \mathbf{T_2}$,<br>где $\mathbf{T_1}$ является пользовательским типом, основанным на типе $\mathbf{T_2}$ (или наоборот). |
| **Семантика** | Вершина выполняет приведение типа к пользовательскому типу, имеющему в своей основе исходный тип. Никакого реального действия над значением типа не производится, как и в операции *:Radian* следующего примера.<br>`type Radian := double;`<br>`function Pi(returns Radian)`<br>`  3.14d0 : Radian`<br>`end function`<br>Также эта вершина может выполнять приведение пользовательского типа к его базовому типу, как преобразование *:double* в следующем примере:<br>`type Degree := double;`<br>`function deg2rad(deg: Degree returns Radian)`<br>`  3.14d0 / 180.0d0 * deg:double`<br>`end function` |

| **Тип вершины** | *gtCharacter* |
|---|---|
| **Порты** | integer → character |
| **Семантика** | Вершина генерирует значение символьного типа с кодом символа, равным входному целому числу. |

| **Тип вершины** | *gtInteger* |
|---|---|
| **Порты** | {character, real, double} → integer |
| **Семантика** | Вершина преобразует указанные значения в целое число, равное коду символа, если на входе находится значение символьного типа, иначе, в случае вещественного числа, отбрасывается его дробная часть. |

| **Тип вершины** | *gtRoundInteger* |
|---|---|
| **Порты** | {real, double} → integer |
| **Семантика** | Вершина округляет вещественное число до целого значения. |

| **Тип вершины** | *gtReal* |
|---|---|
| **Порты** | {integer, double} → real |
| **Семантика** | Вершина генерирует значение вещественного числа одинарной точности, равное, с учётом потери точности, заданному целому или вещественному числу двойной точности. |

| **Тип вершины** | *gtDouble* |
|---|---|
| **Порты** | {integer, real} → double |
| **Семантика** | Вершина генерирует значение вещественного числа двойной точности, равное заданному целому или вещественному числу одинарной точности. |

## 4.2. Арифметические операции

| **Тип вершины** | *gtAdd, gtSub, gtMul, gtDiv* |
|---|---|
| **Порты** | integer, integer → integer или<br>real, real → real или<br>double, double → double |
| **Семантика** | Вершина *gtAdd* складывает значения входных портов. Вершина *gtSub* вычитает значение второго входного порта от значения первого входного порта. Вершина *gtMul* умножает значения входных портов. Вершина *gtDiv* делит значение первого входного порта на значение второго входного порта. В любом случае результат помещается в выходной порт. |

| **Тип вершины** | *gtMod* |
|---|---|
| **Порты** | integer, integer → integer |
| **Семантика** | Вершина выполняет вычисление остатка от целочисленного деления значения первого входного порта на значение второго входного порта. |

| **Тип вершины** | *gtPow* |
|---|---|
| **Порты** | real, real → real или<br>double, double → double |
| **Семантика** | Вершина возводит значение первого порта в степень, задаваемую значением второго входного порта. |

| **Тип вершины** | *gtEqual, gtNotEqual* |
|---|---|
| **Порты** | boolean, boolean → boolean или<br>character, character → boolean или<br>integer, integer → boolean или<br>real, real → boolean или<br>double, double → boolean |
| **Семантика** | Вершина *gtEqual* сравнивает значения первого и второго входных портов и возвращает значение **true**, если эти значения равны. Иначе возвращается значение **false**. Вершина *gtNotEqual* возвращает значение **true**, когда значения портов не равны, и значение **false** в случае равенства этих значений. |

| **Тип вершины** | *gtTypeEqual, gtTypeNotEqual* |
|---|---|
| **Порты** | $\mathbf{T_1}$, $\mathbf{T_2}$ → boolean,<br>где значение второго порта может быть опущено. |
| **Семантика** | Вершина *gtTypeEqual* сравнивает типы значений первого и второго входных портов и возвращает значение **true**, если эти значения равны. Иначе возвращается значение **false**. Вершина *gtTypeNotEqual* возвращает значение **true**, когда типы значений портов не равны, и значение **false** в случае равенства этих типов.<br>Применение данных вершин необходимо тогда, когда транслятор не в состоянии определить тип значения первого или второго портов. Это может произойти, если тип $\mathbf{T_1}$ или тип $\mathbf{T_2}$ заданы с помощью множества типов. |

| **Тип вершины** | *gtLess, gtLessOrEqual* |
|---|---|
| **Порты** | character, character → boolean или<br>integer, integer → boolean или<br>real, real → boolean или<br>double, double → boolean |
| **Семантика** | Вершина *gtLess* возвращает значение **true**, если значение первого входного порта строго меньше значения второго входного порта. В противном случае возвращается значение **false**. Вершина *gtLessOrEqual* действует также, только в случае равенства сравниваемых значений возвращается значение **true**. |

| **Тип вершины** | *gtAnd, gtOr* |
|---|---|
| **Порты** | boolean, boolean → boolean |
| **Семантика** | Вершина *gtAnd* возвращает значение **true** в случае, если оба значения входных портов равны **true**. Вершина *gtOr* возвращает значение **true** в случае, если значение хотя бы одного входного порта равняется **true**. |

| **Тип вершины** | *gtNeg* |
|---|---|
| **Порты** | integer → integer или<br>real → real или<br>double → double |
| **Семантика** | Вершина меняет знак значения входного порта. |

| **Тип вершины** | *gtNot* |
|---|---|
| **Порты** | boolean → boolean |
| **Семантика** | Вершина возвращает значение **true**, если значение входного порта равно **false**, и возвращает значение **false** в противном случае. |

| **Тип вершины** | *gtIsError* |
|---|---|
| **Порты** | **T** → boolean |
| **Семантика** | Вершина возвращает значение **true**, если значение входного порта равно ошибочному значению типа **T**. Иначе возвращается значение **false**. |

### 4.3. Операции с массивами

| **Тип вершины** | *gtABuild* |
|---|---|
| **Порты** | integer, (**T**)*$^{n}$ → **array**[**T**] |
| **Семантика** | Вершина создаёт массив с нижней границей, равной значению первого входного порта, который состоит из n элементов, перечисленных на остальных входных портах. |

| **Тип вершины** | *gtAFill* |
|---|---|
| **Порты** | integer, integer, **T** → **array**[**T**] |
| **Семантика** | Вершина создаёт массив с нижней границей, равной значению первого входного порта, который состоит из повторений значения, заданного на третьем входном порте. Количество повторений задаётся значением второго входного порта. |

| **Тип вершины** | *gtAElement* |
|---|---|
| **Порты** | **array**$^{n}$[**T**], (integer)+$^{n}$ → **T** |
| **Семантика** | Вершина осуществляет выборку элемента массива, который задан на первом входном порте. Последующие входные порты определяют n-мерный индекс, с помощью которого осуществляется выборка. |

Более сложные операции выборки над массивами реализуются с помощью выражений **for** с многомерными диапазонами. Например, пусть A — n-мерный массив. Представим выражение выборки его элементов в виде «A [ <выражение выбора> ]». Выражение выбора можно представить как $D_1$ … $D_2$ … $D_n$, где $D_{1\ldots n}$ — это синглет, дуплет, триплет либо список индексов.

Такое выражение выборки порождает массив размерности m, равной размерности выражения выбора. Обозначим сумму количества дуплетов,

триплетов и индексных массивов, участвующих в выражении выбора, как m1. Тогда размерность выражения выбора определяется как разность m1 и количества операторов **dot** в выражении выбора.

Пусть $f_1$, …, $f_{m1}$ — это номера дуплетов, триплетов и индексных массивов выражения выбора. Тогда выражение выборки можно представить следующим образом:

```
for x_f1 in D_f1 returns array of
  for x_f2 in D_f2 returns array of
    …
      for x_fm1 in D_fm1 returns array of
        A [ d_1, …, d_n ]
      end for
    …
  end for
end for
```

Здесь $d_i = x_i$, если i совпадает с каким-нибудь номером $f_1$, …, $f_{m1}$, иначе $d_i = D_i$, т. е. константе (синглету). Для непрерывных цепочек **dot** операций «$D_{fi}$ **dot** $D_{fi+1}$ … **dot** $D_{fi+j}$» в предыдущем выражении надо произвести замену следующей последовательности операторов **for**:

```
for x_fi in D_fi returns array of
  for x_fi+1 in D_fi+1 returns array of
    …
      for x_fi+j in D_fi+j returns array of
```

Эта последовательность операторов **for** заменяется следующим оператором:

```
for x_fi in D_fi dot
    x_fi+1 in D_fi+1 dot
    …
    x_fi+j in D_fi+j returns array of
```

Тем самым, произвольное выражение выборки элементов массива сведено к операции выборки элемента массива по его индексу.

| **Тип вершины** | *gtAReplace* |
|---|---|
| **Порты** | **array**$^n$[**T**], (integer)+$^n$, T → **array**$^n$[**T**] |
| **Семантика** | Вершина замещает элемент массива заданного значением первого входного порта. Замещаемый элемент указывается с помощью индекса, расположенного на входных портах со второго до n+2. Замещающий элемент указан значением последнего входного порта. |

Операцию замещения элементов массива общего вида можно представить как «A [ <выражение выбора> ! <выражение замещения> ]». Далее используются обозначения, введённые при описании вершины *gtAElement*.

Случай, когда выражение выбора задаётся списком синглетов, непосредственно описывается с помощью вершины *gtAReplace*. Предположим, что в выражении выбора имеется хотя бы один дуплет, триплет или список индексов.

Если выражение замещения задано значением типа элемента n-мерного массива A, то операцию замещения можно представить в следующем виде:

```
for xf1 in Df1 cross
    xf2 in Df2 cross
    …
    xfm1 in Dfm1
  A := old A [ d1, …, dn ! <выражение замещения> ]
  returns value of A
end for
```

Здесь в соответствующих местах вместо оператора **cross** находится оператор **dot**. А если выражение замещения — это массив размерности m от элементов, имеющих тип элемента n-мерного массива A, то операция замещения выражается следующим образом:

```
let i1 := 1;
in for xf1 in Df1
  i1 := old i1 + 1;
  i2 := 1;
  returns value of for xf2 in Df2
    i2 := old i2 + 1;
    i3 := 1;
    …
      returns value of for xfm1 in Dfm1
        im1 := old im1 + 1;
        A := old A [d1, …, dn ! <выраже-
ние замещения> [c1, …, cm]]
        returns value of A
      end for
    …
  end for
end for end let
```

Для непрерывных цепочек **dot** операций «$D_{fi}$ **dot** $D_{fi+1}$ … **dot** $D_{fi+j}$» в предыдущем выражении надо произвести замену следующей последовательности операторов **for**:

```
for x_fi in D_fi
  i_i := old i_i + 1;
  i_i+1 := 1;
  returns value of for x_fi+1 in D_fi+1
    i_i+1 := old i_i+1 + 1;
    i_i+2 := 1;
    …
      returns value of for x_fi+j in D_fi+j
```

Эта последовательность операторов **for** заменяется следующим оператором:

```
for x_fi in D_fi dot
    x_fi+1 in D_fi+1 dot
    …
    x_fi+j in D_fi+j
```

После такой замены останется ровно m значений $i_{1\ldots m1}$. Использованная выше последовательность индексов «$c_1$, …, $c_m$» определяется как последовательность оставшихся значений «$i_1$, …, $i_{m1}$».

| **Тип вершины** | *gtAAddH, gtAAddL* |
|---|---|
| **Порты** | **array**[**T**], **T** → **array**[**T**] |
| **Семантика** | Вершина *gtAAddH* добавляет в конец массива, заданного значением первого входного порта, элемент, находящийся на втором входном порте. Вершина *gtAAddL* добавляет этот элемент в начало массива. |

| **Тип вершины** | *gtARemH, gtARemL* |
|---|---|
| **Порты** | **array**[**T**] → **array**[**T**] |
| **Семантика** | Вершина *gtARemH* удаляет последний элемент массива, заданного значением первого входного порта. Вершина *gtARemL* удаляет первый элемент этого массива. |

| **Тип вершины** | *gtALimL, gtASize* |
|---|---|
| **Порты** | **array**[**T**] → integer |
| **Семантика** | Вершина *gtALimL* возвращает нижнюю границу массива, который находится на первом входном порте. Вершина *gtASize* возвращает количество элементов в этом массиве. |

| Тип вершины | *gtASetL* |
|---|---|
| **Порты** | **array**[**T**], integer → **array**[**T**] |
| **Семантика** | Вершина возвращает массив, равный массиву, который находится на первом входном порте. Только значение нижней границы нового массива равно значению второго входного порта. |

| Тип вершины | *gtACatenate* |
|---|---|
| **Порты** | **array**[**T**], (**array**[**T**])$^{+n}$ → **array**[**T**] |
| **Семантика** | Вершина возвращает массив, полученный последовательным склеиванием массивов, находящихся на входных портах вершины. Нижняя граница нового массива полагается равной единице. |

## 4.4. Операции с потоками

| Тип вершины | *gtSBuild* |
|---|---|
| **Порты** | (**T**)$^{*n}$ → **stream**[**T**] |
| **Семантика** | Вершина создаёт поток, который состоит из n элементов, перечисленных на входных портах вершины. |

| Тип вершины | *gtSFill* |
|---|---|
| **Порты** | integer, **T** → **stream**[**T**] |
| **Семантика** | Вершина создаёт поток, который состоит из повторений значения, заданного на втором входном порте. Количество повторений задаётся значением первого входного порта. |

| Тип вершины | *gtSRange* |
|---|---|
| **Порты** | integer, integer, integer → **stream**[integer] |
| **Семантика** | Вершина генерирует поток целых чисел, состоящий из последовательности чисел, начинающейся со значения, заданного первым входным портом, и заканчивающейся значением, которое меньше либо равно значению второго входного порта. Шаг чисел в этой последовательности задаётся значением третьего входного порта. Если значение шага отрицательно, то в последовательность входят числа большие или равные значению второго входного порта. |

| **Тип вершины** | *gtSFirst* |
|---|---|
| **Порты** | **stream**[**T**] → **T** |
| **Семантика** | Вершина возвращает значение первого элемента потока, находящегося на первом входном порте. |

| **Тип вершины** | *gtSRest* |
|---|---|
| **Порты** | **stream**[**T**] → **stream**[**T**] |
| **Семантика** | Вершина возвращает поток, полученный из потока, заданного на первом входном порте, путём удаления его первого элемента. |

| **Тип вершины** | *gtSSize* |
|---|---|
| **Порты** | **stream**[**T**] → integer |
| **Семантика** | Вершина *gtSSize* возвращает количество элементов в потоке, находящемся на первом входном порте. |

| **Тип вершины** | *gtSCatenate* |
|---|---|
| **Порты** | **stream**[**T**], (**stream**[**T**])$+^{n}$ → **stream**[**T**] |
| **Семантика** | Вершина возвращает поток, полученный последовательным склеиванием потоков, находящихся на входных портах вершины. |

## 4.5. Операции с записями

| **Тип вершины** | *gtRBuild* |
|---|---|
| **Порты** | $\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_n}$ → **record**[ :$\mathbf{T_1}$; :$\mathbf{T_2}$; …; :$\mathbf{T_n}$] |
| **Семантика** | Вершина создаёт новую запись с полями, равными значениям соответствующих входных портов. |

| **Тип вершины** | *gtRElements* |
|---|---|
| **Порты** | **record**[ :$\mathbf{T_1}$; :$\mathbf{T_2}$; …; :$\mathbf{T_n}$] → $\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_n}$ |
| **Семантика** | На соответствующих выходных портах вершина возвращает значения полей записи, которая находится на первом входном порте. |

| **Тип вершины** | *gtRReplace* |
|---|---|
| **Порты** | **T = record**[ :$\mathbf{T_1}$; :$\mathbf{T_2}$; …; :$\mathbf{T_n}$], $\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_n}$ → **T** |
| **Семантика** | Вершина возвращает запись, полученную из записи, которая задана на первом входном порте, после замены значений её нескольких полей. Значение поля замещается, если соответствующий ему входной порт не пуст. Входные порты, соответствующие полям записи, располагаются со второго входного порта. |

## 4.6. Операции с союзами

| **Тип вершины** | *gtUBuild* |
|---|---|
| **Порты** | $\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_n}$ → **union**[ :$\mathbf{T_1}$; :$\mathbf{T_2}$; …; :$\mathbf{T_n}$] |
| **Семантика** | Вершина создаёт новый союз, содержащий значение, заданное на одном из входных портов, которые обозначают соответствующие теги союза. Остальные входные порты вершины пусты. |

| **Тип вершины** | *gtUElements* |
|---|---|
| **Порты** | **union**[ :$\mathbf{T_1}$; :$\mathbf{T_2}$; …; :$\mathbf{T_n}$] → $\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_n}$ |
| **Семантика** | На одном из выходных портов, соответствующих тегам союза, вершина возвращает значение союза, который находится на первом входном порте. Остальные выходные порты содержат ошибочные значения соответствующих типов тегов союза. |

| **Тип вершины** | *gtUTestTag* |
|---|---|
| **Порты** | **union**[ :$\mathbf{T_1}$; :$\mathbf{T_2}$; …; :$\mathbf{T_n}$], (boolean)+$^{n}$ → boolean |
| **Семантика** | Вершина возвращает значение **true**, если значение союза, находящееся на первом входном порте, соответствует заданному тегу. Входные порты, соответствующие тегам союза, располагаются, начиная со второго входного порта. Тег задан, если соответствующий ему порт не пуст и его значение равно **true**. |

## 4.7. Операции с функциями

| **Тип вершины** | *gtCall* |
|---|---|
| **Порты** | **function**[$\mathbf{A_1}$, $\mathbf{A_2}$, …, $\mathbf{A_n}$ **returns** $\mathbf{R_1}$, $\mathbf{R_2}$, …, $\mathbf{R_n}$], $\mathbf{A_1}$, $\mathbf{A_2}$, …, $\mathbf{A_n}$ → $\mathbf{R_1}$, $\mathbf{R_2}$, …, $\mathbf{R_n}$ |
| **Семантика** | Вершина осуществляет вызов функции, заданной на первом входном порте, с аргументами, заданными на последующих входных портах. Выходные порты вершины соответствуют результатам вызова функции. |

| **Тип вершины** | *gtBindArguments* |
|---|---|
| **Порты** | **function**[$\mathbf{A_1}$, $\mathbf{A_2}$, …, $\mathbf{A_n}$ **returns** $\mathbf{R_1}$, $\mathbf{R_2}$, …, $\mathbf{R_n}$], $\mathbf{A_1}$, $\mathbf{A_2}$, …, $\mathbf{A_n}$ → **function**[$\mathbf{A_{i(1)}}$, $\mathbf{A_{i(2)}}$, …, $\mathbf{A_{i(j)}}$ **returns** $\mathbf{R_1}$, $\mathbf{R_2}$, …, $\mathbf{R_n}$] |
| **Семантика** | Вершина возвращает функцию от меньшего числа аргументов, чем функция, которая задана на первом входном порте. Аргументами новой функции ($\mathbf{A_{i(1)}}$, $\mathbf{A_{i(2)}}$, …, $\mathbf{A_{i(j)}}$) остаются те аргументы заданной функции, чьи входные порты пусты. Остальные аргументы заданной функции полагаются равными значениям соответствующих им входных портов. Входные порты ($\mathbf{A_1}$, $\mathbf{A_2}$, …, $\mathbf{A_n}$), соответствующие аргументам заданной функции, располагаются, начиная со второго входного порта. |

## 4.8. Операции с индексами

Следующие две вершины могут встречаться только в графе «генератора диапазона» составной вершины *gtForAll*. Основная их особенность заключается в том, что им присущ неявный параметр, определяемый только во

время интерпретации внутреннего представления. Далее он будет обозначаться именем ITERATION.

| **Тип вершины** | *gtScatter* |
|---|---|
| **Порты** | **array**[**T**] → **T**, integer или<br>**stream**[**T**] → **T**, integer |
| **Семантика** | Вершина возвращает элемент (в качестве первого выходного порта) и его индекс (в качестве второго выходного порта), в списке индексов. Список индексов находится на первом входном порте. Индекс возвращаемого элемента списка индексов равен нижней границе массива плюс значение ITERATION, если список индексов задан массивом. Если список индексов задан потоком, то значение индекса элемента равно значению ITERATION. Данная вершина определена для значений параметра ITERATION от 1 до количества элементов списка индексов. |

| **Тип вершины** | *gtDot* |
|---|---|
| **Порты** | {**array**, **stream**}[**T**], ({**array**, **stream**}[**T**])$+^{n}$ → **T**, (**T**)$+^{n}$ |
| **Семантика** | Вершина определяется для значений параметра ITERATION от 1 до количества элементов в самом длинном из списков индексов, находящихся на входных портах вершины. Вершина помещает элемент с номером ITERATION каждого списка индексов в соответствующий этому списку выходной порт. Если список индексов задан массивом, то его нижняя граница не учитывается. В случае если элемента с таким номером в списке индексов нет, то значение соответствующего выходного порта равняется ошибочному значению типа **T**. |

### 4.9. Операции с редукциями

Все вершины редукций могут встречаться только в графе «предложения возврата» составной вершины, задающей циклическое выражение. Вершины редукций имеют общее назначение первого входного порта: он имеет булевский тип, и его значение управляет включением текущих циклических параметров редукции в редуцируемую последовательность. Значение по умолчанию для предопределённых редукций используется тогда, когда редуцируемая последовательность пуста.

| **Тип вершины** | *gtReduceValue* |
|---|---|
| **Порты** | boolean, **T** → **T** |
| **Семантика** | Вершина обозначает редукцию «value **of**», которая возвращает последнее значение редуцируемой последовательности.<br>Значение, по умолчанию возвращаемое этой редукцией, определяется несколько специфическим образом. Если редуцируемое значение является константным выражением или константным циклическим именем, тогда редукция по умолчанию возвращает это значение. Если редуцируемое значение является циклически локальным и не стоит в тесте, тогда возвращается ошибка типа **T**. Если редуцируемое значение является циклически локальным, стоящим в тесте, или является циклически зависимым, тогда возвращается значение, связанное с редуцируемым значением во внешней области действия. |

| **Тип вершины** | *gtReduce* |
|---|---|
| **Порты** | boolean, **reduction**[$\mathbf{I_1}$, $\mathbf{I_2}$, …, $\mathbf{I_n}$ **repeat** $\mathbf{L_1}$, $\mathbf{L_2}$, …, $\mathbf{L_n}$ **returns** $\mathbf{R_1}$, $\mathbf{R_2}$, …, $\mathbf{R_n}$], $\mathbf{I_1}$, $\mathbf{I_2}$, …, $\mathbf{I_n}$, $\mathbf{L_1}$, $\mathbf{L_2}$, …, $\mathbf{L_n}$ → $\mathbf{R_1}$, $\mathbf{R_2}$, …, $\mathbf{R_n}$ |
| **Семантика** | Вершина обозначает пользовательскую редукцию, заданную на втором входном порте. Начиная с третьего входного порта идут начальные значения редукции, а после располагаются её циклические значения. Результаты редукции находятся на выходных портах. |

| **Тип вершины** | *gtReduceSum, gtReduceProduct, gtReduceGreatest, gtReduceLeast* |
|---|---|
| **Порты** | boolean, integer → integer или<br>boolean, real → real или<br>boolean, double → double |

| | |
|---|---|
| **Семантика** | Вершина *gtReduceSum* обозначает редукцию «sum **of**», которая суммирует редуцируемые значения. Значение по умолчанию равно 0.<br>Вершина *gtReduceProduct* обозначает редукцию «pruduct **of**», которая перемножает редуцируемые значения. Значение по умолчанию равно 1.<br>Вершина *gtReduceGreatest* обозначает редукцию «greatest **of**», которая выбирает наибольшее среди редуцируемых значений. Значение по умолчанию равно минимально возможному значению возвращаемого типа.<br>Вершина *gtReduceLeast* обозначает редукцию «least **of**», которая выбирает наименьшее среди редуцируемых значений. Значение по умолчанию равно максимально возможному значению типа возвращаемого типа. |

| **Тип вершины** | *gtReduceArray* |
|---|---|
| **Порты** | boolean, integer, **T** → **array**[**T**] |
| **Семантика** | Вершина обозначает редукцию «**array of**». Второй вход вершины задаёт нижнюю границу массива, состоящего из редуцируемых значений. По умолчанию подразумевается пустой массив. |

| **Тип вершины** | *gtReduceStream* |
|---|---|
| **Порты** | boolean, **T** → **stream**[**T**] |
| **Семантика** | Вершина обозначает редукцию «**stream of**», которая собирает редуцируемые значения в поток. По умолчанию подразумевается пустой поток. |

| **Тип вершины** | *gtReduceACatenate* |
|---|---|
| **Порты** | boolean, integer, **array**[**T**] → **array**[**T**] |
| **Семантика** | Вершина обозначает редукцию «catenate **of**». Второй вход вершины задаёт нижнюю границу массива, который получается последовательным склеиванием массивов, являющихся редуцируемыми значениями. По умолчанию подразумевается пустой массив. |

| **Тип вершины** | *gtReduceSCatenate* |
|---|---|
| **Порты** | boolean, **stream**[**T**] → **stream**[**T**] |
| **Семантика** | Вершина обозначает редукцию «catenate **of**», которая последовательно склеивает потоки, являющиеся редуцируемыми значениями. По умолчанию подразумевается пустой поток. |

## 5. ОПИСАНИЕ «СОСТАВНЫХ ВЕРШИН» IR1 ДЛЯ SISAL 3.0

### 5.1. Составная вершина выбора *gtSelect*

Данная составная вершина имеет произвольное число входов и выходов. Обозначим число выходов именем N. Количество графов составной вершины *gtSelect* больше или равно трём. Обозначим их количество числом M.

Первый граф имеет M-2 выходов и N входов соответствующих (числом, порядком и типом) входам составной вершины. Все остальные графы этой составной вершины имеют такие же входы. Некоторые входы могут в графе не использоваться, но они всё равно присутствуют, чтобы не менять нумерацию, позволяющую отождествить их с входными портами составной вершины.

Каждый выход первого графа составной вершины *gtSelect* имеет булевский тип. Последовательно вычисляем их значения до первого значения **true**. Предположим, что это случилось и значение **true** вычислилось на выходном порте с номером m, где 1 <= m <= M - 2. В этом случае дальше рассматриваем граф с номером m + 1. Этот граф и будет определять значение выходных портов составной вершины *gtSelect*.

Если же ни один из выходных портов первого графа не возвращает значения **true**, то для определения значений выходных портов составной вершины *gtSelect* будет использоваться её последний граф с номером M, соответствующий ветви **else**.

В свою очередь, каждый граф с номером от 2 до M, используемый для генерации выходных значений, имеет в точности N выходов, которые однозначно соответствуют выходам составной вершины *gtSelect*.

Структура первого (управляющего) графа данной составной вершины отличается от структуры, предложенной в IF1. Там этот граф имеет один выход целого типа. Согласно значению этого выхода происходит выбор графа для генерации выходных значений.

Однако генерация такого числа связана с использованием дополнительных операций сложения, «логического И» и «логического НЕ», что неоправданно усложняет граф, понижая его читаемость.

На рис. 7 приведён пример такого управляющего графа для следующего выражения.

```
if     b1 then …
elseif b2 then …
elseif b3 then …
          else …
end if
```

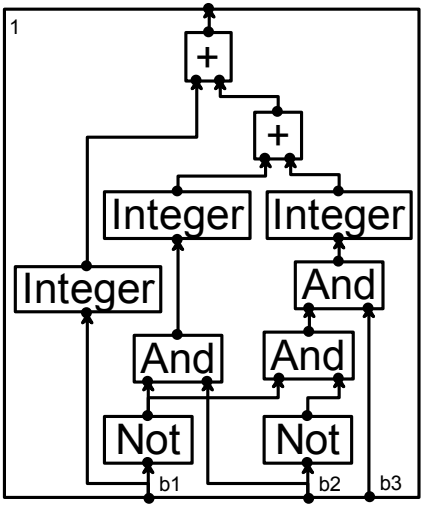



*Рис. 7.* Управляющий граф составной вершины выбора (Select) в IF1

Из рис. 7 видна необходимость, которая до сих пор не возникала, в преобразовании (integer) от булевского типа к целому типу. Также неприятным моментом является нарастание в арифметической прогрессии количество вершин *gtAnd* для каждого последующего условия **elseif**.

Для интерпретатора этого графа также существенным было бы использование вычислений с коротким замыканием при обработке вершины *gtAnd*, которое заключается в запрете вычисления значения второго входа этой вершины, если значение на первом входе равняется **false**. Иначе на

результаты вычисления условий **elseif** влияли бы ошибочные значения последующих условий. Тем самым, предложенный вариант управляющего графа с несколькими выходами можно также рассматривать как разновидность операции, требующей для своего вычисления правила короткого замыкания.

Управляющий граф составной вершины *gtSelect* в IF1 допускает модель вычислений, основанную на готовности данных (data-driven), в отличие от предложенного нами варианта управляющего графа с несколькими выходами, требующего модель вычислений по запросу (demand-driven). Однако указанные выше сложности структуры управляющего графа *gtSelect* в спецификации IF1 приводят к использованию предложенного варианта. Требование использования demand-driven модели его вычисления не является чем-то исключительным, так как смешанный подход, основанный на применении различных моделей вычисления по соображениям эффективности, является общепринятым для не потоковых машинных архитектур.

Может возникнуть закономерный вопрос: «Почему бы не разбить предложенный управляющий граф с несколькими выходами для составной вершины *gtSelect* на несколько управляющих графов, имеющих один выход, для которых можно использовать data-driven модель вычислений?». Это было бы оправданно, если бы составная вершина *gtSelect* не использовалась для реализации синтаксической конструкции «**case-of-then[-else]**». У этой конструкции есть единое управляющее выражение, используемое во всех её условных ветвях **of-then**. Тем самым, при разбиении каждого её условия по отдельным графам возникает необходимость вычислять управляющее выражение несколько раз.

Использование же одной составной вершины для реализации разных синтаксических конструкций «**if-then-elseif-else**» и «**case-of-then[-else]**» связано с унификацией их схожего семантического назначения для облегчения последующих преобразований (оптимизаций). Хотя, конечно, потеря явно выделенного управляющего выражения «**case-of-then[-else]**» конструкции создаёт дополнительные сложности при необходимости получения эффективной программы на языке SISAL при ретрансляции данной составной вершины.

Составная вершина *gtSelect* используется для представления следующих конструкций языка SISAL.

- Выражение «**if-then-elseif-else**».
- Выражение «**case-of-then[-else]**». В случае отсутствия ветви **else**, в качестве последнего графа генерируется граф, возвращающий ошибочные значения типов, соответствующих выходам составной вершины *gtSelect*.
- Выражение «**case type-of-then[-else]**». В управляющем графе используется простая вершина *gtTypeEqual*.
- Выражение «**case tag-of-then[-else]**». В управляющем графе используется простая вершина *gtUTestTag*.
- Выражение «**where-is-in**».

Выражение «**where-is-in**» имеет следующую синтаксическую форму:

```
where <унарное выражение типа массив where>
  is <идентификатор where> in <n-арное выраже-
ние where>
end where,
```

где <идентификатор *where*> обозначает элемент массива <унарное выражение типа массив *where*> в <n-арное выражение *where*>. Если предположить, что <унарное выражение типа массив *where*> является n-мерным массивом, то выражение «**where-is-in**» при помощи других конструкций представляется следующим образом:

```
for A1 in <унарное выражение типа массив where>
returns array of
  for A2 in A1 returns array of
    …
      for <идентификатор where> in An-1 returns
array of
        <n-арное выражение where>
      end for
    …
  end for
end for
```

Эта синтаксическая форма выражения «**where-is-in**» заменяет приведенный ниже синтаксис данного оператора в языке SISAL-90.

```
where <унарное выражение от одного массива>
then <унарное выражение>
  <список elsewhere>
  else <унарное выражение>
end where
```

Новая синтаксическая форма была введена для облегчения процесса выделения управляющего массива для программиста и компилятора. К тому же снимается ограничение на применение булевских условий. Также, вместо имени массива теперь возможно использовать выражение с результатом типа массив. Тем самым, новая синтаксическая форма более функциональна и удобна.

На рис. 8 приведён пример составной вершины *gtSelect*, порождаемой приведенным ниже выражением.

```
if     A+B < E+F then A, B
elseif C+D < E+F then C, D
                     else E, F
end if
```

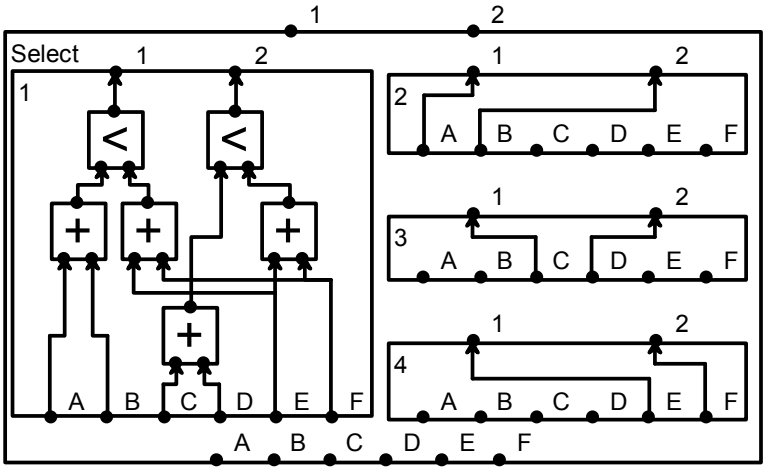



*Рис. 8.* Составная вершина *gtSelect*

### 5.2. Составные вершины циклов

Порты составной вершины циклов и её графов можно разбить на следующие общие группы:

- класс **K** — это значения, импортируемые в составную вершину;
- класс **R** — это значения, экспортируемые из составной вершины;
- класс **L** — это циклические значения;

- класс **OL** — это циклические значения с предыдущей итерации (имена с префиксом **old**).

Значения класса **OL** числом, порядком и типами соответствуют значениям класса **L**.

### *5.2.1. Цикл с постусловием gtLoopA*

Данная составная вершина используется для циклов с постусловием **for-repeat** языка SISAL. Составная вершина *gtLoopA* имеет четыре графа.

| **Номер графа в списке подграфов графа составной вершины** | **Название графа** | **Классы входных портов графа** | **Классы выходных портов графа** |
|---|---|---|---|
| 1 | Граф *инициализации*. | **K** | **L** |
| 2 | Граф *тела цикла*. | **OL**, **K** | **L** |
| 3 | Граф *теста цикла*. | **L**, **OL**, **K** | Один порт булевского типа. |
| 4 | Граф *предложения возврата*. | **L**, **OL**, **K** | **R** |

Граф *инициализации* это не первая итерация цикла (см. составные вершины *gtLoopA1* и *gtLoopB1*). Он служит для связывания значений внешних имён с «**old** именами» для первого исполнения тела цикла. Так как иначе не понятно, где брать значения «**old** имён» в первой итерации тела цикла, задаваемой графом *тела цикла*. Ведь если взять эти значения из группы портов **K**, то станет неправильным граф потока вычислений для последующих итераций. Фактически, граф *инициализации* состоит только из входных и выходных портов и дуг, их соединяющих.

В графе *тела цикла*, изображенном на рис. 9, нет необходимости во входных портах из группы **L**, как можно было бы подумать, рассматривая следующий пример:

```
for repeat
  A := old A + 1;
  B := A * 2
returns value of B
end for
```

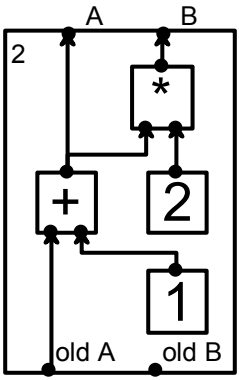



*Рис. 9.* Граф *тела цикла* составной вершины *gtLoopA*

Граф *предложения возврата* содержит уникальные вершины, описывающие вызов пользовательской или предопределенной редукции. Причём эти вершины сразу поставляют значения выходам графа предложения возврата, которые в свою очередь получают значения только от вершин редукций.

### *5.2.2. Цикл с предусловием gtLoopB*

Данная составная вершина используется для циклов с предусловием **for-while** языка SISAL. Составная вершина *gtLoopB* имеет четыре графа.

| **Номер графа в списке подграфов графа составной вершины** | **Название графа** | **Классы входных портов графа** | **Классы выходных портов графа** |
|---|---|---|---|
| 1 | Граф *инициализации*. | **K** | **L** |
| 2 | Граф *теста цикла*. | **L**, **K** | Один порт булевского типа. |
| 3 | Граф *тела цикла*. | **OL**, **K** | **L** |
| 4 | Граф *предложения возврата*. | **L**, **OL**, **K** | **R** |

### *5.2.3. Циклы с выделенной первой итерацией gtLoopA1 и gtLoopB1*

Эти составные вершины, по сравнению с составными вершинами *gtLoopA* и *gtLoopB*, имеют дополнительный граф *первой итерации* тела цикла. Этот граф располагается непосредственно за графом инициализации, увеличивая номера других графов на единицу. Конфигурация портов графа *первой итерации* полностью аналогична конфигурации портов в графе *тела цикла*.

На рис. 10 приведена составная вершина, соответствующая приведенному ниже выражению цикла с постусловием и первой итерацией.

```
let
  A  := 11;
  A0 := 8
in for initial
  A := old A + A0;
  B := A * 2
repeat
  A := old A + 1;
  B := A * 3
while B < 100 or old A < 200
returns sum of A * B
        value of B when B mod 2 = 0
end for
end let
```

Данная составная вершина применяется для циклов с генератором диапазона вместо условия («выражение for» в SISAL терминологии). Для этой составной вершины вводится дополнительная группа портов класса **D**. Класс **D** — это значения, генерируемые графом *генератора диапазона*. Имена портов этой группы не могут иметь префикса **old**. Данная составная вершина содержит четыре графа.

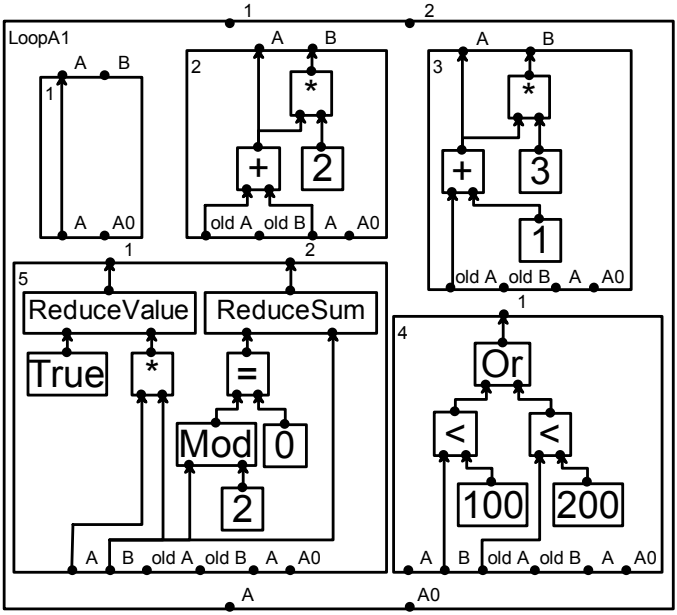



*Рис. 10.* Граф составной вершины *gtLoopA1*

### *5.2.4. Выражение цикла gtForAll*

| **Номер графа в списке подграфов графа составной вершины** | **Название графа** | **Классы входных портов графа** | **Классы выходных портов графа** |
|---|---|---|---|
| 1 | Граф *инициализации*. | **K** | **L** |
| 2 | Граф *генератора диапазона*. | **K** | **D** |
| 3 | Граф *тела цикла*. | **OL**, **D**, **K** | **L** |
| 4 | Граф *предложения возврата*. | **L**, **OL**, **D**, **K** | **R** |

Граф *генератора диапазона* содержит уникальные вершины *gtScatter* и *gtDot*. Эти и только эти вершины поставляют значения портам группы **D** этого графа. Предполагается, что простые вершины на этапе интерпретации *gtScatter* и *gtDot* имеют дополнительный неявный параметр.

### 5.3. Пользовательская редукция *gtReduction*

Данная составная вершина служит для представления пользовательских редукций. IR1 граф данной составной вершины может быть только корневым (его атрибут *IGraph→ParentGraph* равен NULL).

Входные порты составной вершины неявно разбиты на два класса: **IN** (начальные значения) и **LN** (потоковые значения). Для их разделения можно использовать количество входных портов (группы **IN**) первого графа данной составной вершины. Данная составная вершина имеет три графа.

| **Номер графа в списке подграфов графа составной вершины** | **Название графа** | **Классы входных портов графа** | **Классы выходных портов графа** |
|---|---|---|---|
| 1 | Граф *первой итерации.* | **IN** | **L** |
| 2 | Граф *тела цикла.* | **OL**, **IN**, **LN** | **L** |
| 3 | Граф *предложения возврата.* | **L**, **IN**, **LN** | **R** |

Выходные порты составной вершины *gtReduction* принадлежат группе **R**, как и выходные порты графа *предложения возврата*.

В графе *первая итерация* некоторые значения выходных портов группы **L** могут остаться не проинициализированным, однако в этом случае соответствующие им входные порты группы **OL** также не будут использоваться в следующих графах этой составной вершины. Это замечание, впрочем, касается также и составных вершин цикла, только применительно к их графу *инициализации*.

В графе *предложения возврата*, в отличие от графа *предложения возврата* составных вершин цикла, из вершин редукций используется только вершина *gtReduceValue*. На рис. 11 приведен пример графа пользовательской редукции, описание которой находится ниже.

```
reduction my_sum(x: integer repeat z:
integer returns integer)
  for repeat
    x := old x + z;
  returns value of x
  end for
end reduction
```

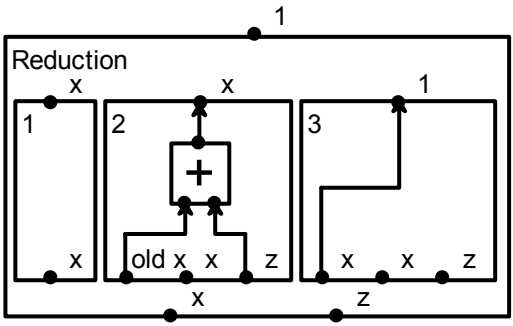



*Рис. 11.* Граф пользовательской редукции

## ЗАКЛЮЧЕНИЕ

В данной работе были рассмотрены реализованные интерфейсы внутреннего представления IR1 и их применение для задания модуля программы языка SISAL 3.0. Была описана семантика вершин внутреннего представления, используемых для представления операций языка SISAL 3.0.

Разработанное внутреннее представление удовлетворяет требованиям, изложенным во введении.

В результате стала возможной реализация других частей (в частности, транслятора) системы функционального программирования (SFP) SISAL 3.0, основывающихся на рассмотренном внутреннем представлении IR1.

Использование технологии COM для реализации интерфейсов IR1 упрощает применение внутреннего представления для компонентов SFP, написанных не на Microsoft Visual C++.

В заключение благодарю аспиранта Хана Юрия Владимировича за множество дельных советов по поводу COM интерфейсов и способов их реализации с помощью ATL.

## СПИСОК ЛИТЕРАТУРЫ

1. **Бирюкова Ю. В.** SISAL 90 руководство пользователя. — Новосибирск, 2000. — 84 с. — (Препр. / РАН. Сиб. отд–е. ИСИ; № 72).
2. **Касьянов В. Н., Бирюкова Ю. В., Евстигнеев В. А.** Функциональный язык Sisal 3.0. // Поддержка супервычислений и интернет–ориентированные технологии. — Новосибирск, 2001. — С. 54–67.
3. **IF1**: an intermediate form for applicative languages / Skedzielewski S., Glauert J. — Livermore, CA, 1985. — (Lawrence Livermore National Laboratory; M-170).
4. **Бокс Д.** Сущность технологии COM. Библиотека программиста. — СПб.: Питер, 2001. — 400 с.: ил.

**Rector B., Sells C.** ATL Internals. — Addison-Wesley Pub Co, 1999. — 635pp.

**А. П. Стасенко**

# ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ СИСТЕМЫ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ SISAL 3.0

**Препринт**
**110**

Рукопись поступила в редакцию 10.12.03
Рецензент В. А. Евстигнеев
Редактор З. В. Скок

---

Подписано в печать 20.08.04
Формат бумаги 60 × 84 1/16 Объем 3.2 уч.-изд.л., 3.5 п.л.
Тираж 60 экз.

---

ЗАО РИЦ «Прайс-курьер»
630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383-2) 34-22-02