The HAC Languge

A boring technical document

David Fang

This manual describes the HAC language specification.

This document can also be found online at http://www.csl.cornell.edu/~fang/hackt/hac.

The main project home page is http://www.csl.cornell.edu/~fang/hackt/.

Short Contents

Table of Contents

Preface

```
                                    You have acquired a scroll entitled
                                         'irk gleknow mizk'(n).--More--

                                   This is an IBM Manual scroll.--More--

                                           You are permanently confused.
```
– *Dave Decot*

Why must every document have a preface?

Goals

Designing a new language is not a task for the faint-hearted. There has to be sufficient motivation to justify such labor:

- No existing language (or composition thereof) meets the requirements demanded.
- A workaround atop of existing languages is deemed to be insufficient.
- For fun.

As selfish as it may seem, the HAC language is designed to meet a very specific requirement of the author. The author's desire is to provide a language to facilitate:

- Ease and effectiveness of (asynchronous) VLSI design automation
- Ease of automatic design space exploration (at high architecture level and low level circuits)

Design Automation

What do other languages lack? Asynchrony? Heirarchical information.

When charged with the task of designing a circuit component, one is usually given a functional specification to meet. The difficulty often lies with coming up with fitting functional specification — knowing *a priori* the context in which a component is used. Without the context in which a component is used, it is futile to optimize the design of that component. Often, one functional specification for a component is really meant to be used in multiple contexts, in which case, one would design a different version for each context.

"Heavy-tree" of definition uses (like calling context stacks).

Ramble ramble ramble...

Shape and form factors...

Optimization using dynamic performance information...

Design Space Exploration

The ultimate ambition for this project is to be able to automatically explore the design space of implementations of significantly complex functional specifications.

Program transformations.

Design choices fall into two major categories: 1) Quantitative (How many? How wide? How large?) and 2) Qualitative (What *kind*?).

Quantitative parameters are simply traits that can be described quantitatively: How many buffers? How wide a datapath?

Examples of qualitative parameters: What kind of buffer? What protocol or reshuffling? Which kind of adder? Whether or not to use a speculative path? Whether or not to introduce resource arbitration? Parallel or serial?

Objectives. Area, energy, performance, efficiency. How about aesthetics and simplicity (design time)?

Asynchrony

So what does *asynchronous* VLSI have to do with deciding to use a new language?

Verilog and VHDL shortcomings... Other existing (public) async. synthesis tools.

1 Introduction

> But in our enthusiasm, we could not resist a radical overhaul of the system, in which all of its major weaknesses have been exposed, analyzed, and replaced with new weaknesses.
>
> *Bruce Leverett, "Register Allocation in Optimizing Compilers"*

In the beginning, there was CSP.

This document describes the language specification for the HAC Language.

Should be largely implementation-independent. But we discuss some of the issues.

Our implementation is a multi-phase compiler with four phases:

- Compile – like precompiling modules
- Link
- Unroll – hierarchical expansion of top-level instantiations.
- Finalize – *unique* instantiation after connections.

Compile-time is ...

Link-time is ...

Unroll-time is ...

Finalize is ...

1.1 Purpose

What is the purpose of HAC? What is the meaning of life?

What this IS:

- a hardware description language

What this IS NOT:

- a sequential programming language

1.2 Roots

The HAC language is based on the CAST (Caltech Asynchronous Synthesis Tool) language.

Discuss limitations. Hierarchical information was lost, as a result of flattening identifiers into strings.

Religious differences, much like the `vi` versus `emacs` holy war, led to a divergence of implementations and interpretations.

1.3 Overview

In Chapter blah we cover blah. In Chapter foo we cover foo.

2 Types

In HAC, there are four classes of types to use: parameters, datatypes, channels, and processes.

2.1 Parameters

Parameters differ from the others in that they don't correspond to any physical instances; they are merely user-manipulated values. Currently, there are only two (built-in) parameter types: `pint`, which is an integer value, and `pbool`, which is a boolean value.

Parameters may be set no more than once, and they must be initialized before they are used. Using uninitialized parameters is an error.

We cover templates more in-depth in Chapter 9 [Templates], page 25. HAC supports parametric types, or templates, much like C++. Terminology: a template definition (or just template) is a parameterized definition. User-defined datatypes, channels, and processes may be parameterized. Currently the only types that may appear in template signatures are parameters and arrays thereof. Later we may add support for template template arguments, muahahaha.

2.2 Definitions

random notes, please pardon the lack of order... we really need to re-organize this document, I know.

Consider introducing templates up front...

Describe definition and type-system, bottom-up, from instantiations?

TODO: make the following distinctions: A physical definition corresponds to the complete description of a particular implementation. Physical definition has its own *concrete layout map* that translates name (possibly with index) to offset. (Users never directly deal with concrete layout maps.) Non-template definitions have only one trivially-generated concrete layout map. Template definitions, however, may have multiple *layout map templates*, among which one is selected to generate a concrete layout map for each instantiation. Layout templates are just value- and type-parameterized layout descriptions. (Multiple layout templates arise from partial and full specializations; a generic template definition contributes one layout template.)

A *type* can either be a reference to a non-template definition (which may be built-in) or a reference to a template definition with a set of parameters.

STALE: Two (non-template) types are equivalent if and only if they refer to the same non-template definition. In Chapter 9 [Templates], page 25 we extend the notion of type-equivalence to cover types with templates.

2.2.1 Physical Layout

Definitions of physical entities invariably contain some table mapping a logical member or port name to the location of the desired member. Such is the typical implementation of data structures in traditional programming languages. Non-template definitions have only one table.

Chapter 9 [Templates], page 25 discusses how templates affects this view of structures' layout tables. Template definitions introduce *table-templates* (necessary when members' types and sizes depend on template parameters). Each complete template definition (all arguments supplied) uses follows one table-template to generate a final layout table.

2.2.2 Names

The *name* of a plain non-template definition refers to the one definition only. (OK, that made no sense whatsoever.) The name of a template definition, however, refers to a *family* of definitions.

2.3 Completeness and Usability

In HAC, there are three levels of completeness for definitions.

1. A definition is *declared* if its template signature (which may be empty) has been defined. Forward declarations of a type only specify the name and template parameters, and no other information.
2. A definition is *signed* if it is declared and its public ports interface (which may be empty) has been defined.
3. A definition is *defined* if it is signed and its body has been defined.

NOTE: what about definition bindings?

Since type information only refers to template parameters, a type may refer to any declared definition.

2.3.1 Instantiation

An instantiation is the creation of definition into a physical object.

2.3.2 Connection

See Chapter 10 [Connections], page 35.

2.3.3 Typedefs

See Chapter 12 [Typedefs], page 39.

2.4 Future extensions

Support '`typeof()`' operator and '`sizeof()`' operator.

3 Expressions

Expressions live in the realm of parameters and datatypes. Compile-time meta language.

Boolean.

Integer.

3.1 Range expressions

Limits: interpretation of negative ranges? Are we allowing negative indices?

Implicit ranges.

Explicit ranges. Q: does $x <= y$? If we allow negative indices then yes, because we need to be able to express empty ranges, when $x > y$.

Compile-time (meta-language) interpretation.

See chapter on compile-time flow-control (loops and conditionals).

4 Arrays

In many languages, arrays are useful for collective or repetitive constructs. In HAC, arrays come in two flavors: sparse and dense.

4.1 Dense arrays

Dense arrays, which may be multidimensional, have the constraint that each dimension is precisely covered by a set of contiguous indices, expressible in the form `[a..x][b..y]`. The lower index of each dimension need not start at 0.

The syntax for dense arrays is similar to that of C declarations, with a few extensions discussed in Section 4.4 [Array Declarations], page 12. The following examples are all dense declarations, resulting in densely packed collections:

Dense array declarations:

```
pint y[2][2];      // y is dense 2D
pint N = 5;
int bar[4][5][N];  // bar is dense 3D
int z[2..6][3];    // z is dense 2D
```

4.2 Sparse arrays

A sparse array, on the other hand, is a generalization of a (possibly multidimensional) set, whose indices need not be continuous.

One feature of HAC is that one may arbitrarily extend arrays, as sets of indices, with multiple declarations. A dense array can be made sparse by adding indexed instances that break the dense condition.

This example declare collection `q` with two statements, resulting in a sparsely poplulated (sparse) collection.

Sparse array from declarations:

```
int q[2][2][2];
int q[1][1][3..3];   // result is sparse
```

A sparse array can be populated densely by filling in indices to satisfy the dense condition,

Dense array from declarations:

```
pint q[1..1][0..1][2..3];
pint q[1][1][2..3];
pint q[1][1..1][2..3]; // result is dense
```

The only constraint is that one cannot re-instantiate an index that has been previously instantiated.

4.3 Size-equivalence

Two arrays are *size-equivalent* if the following are true:

1. The number of dimensions match.
2. Both are densely packed.
3. The size of each dimension is equal.

The range of indices covered by each dimension need not be equal. *Range-equivalence* is a stronger relationship that requires that the respective upper and lower bounds of two arrays match.

Any array that is sparse cannot be size-equivalent to any other array, even if the set of indices contained are identical!

When we refer to instances as being type-equivalent, we also mean that they are size-equivalent. A connection between two instances is legal if and only if they are type-equivalent and size-equivalent. In Chapter 10 [Connections], page 35, we discuss connections more in-depth.

4.4 Declarations

Arrays of any type (parameters, channels, processes, datatypes) are declared the same way. Two syntaces are available for declarations. For example, a 1-D array may be declared using:

- *type identifier* `[` *range* `]` (where *range* : *pint* .. *pint*)
- *type identifier* `[` *pint* `]` (≡ *type identifier* `[` 0..*eval(pint)*-1 `]`)

The first form explicitly specifies the range to instantiate, whereas the second form implicitly starts instantiating from 0 and ends at one less than the evaluated integer argument. The second form is just a convenient syntax for a common construct.

Multidimensional arrays are declared with multiple dimension specifiers. Each dimension of a multidimensional array may be specified using either style.

Instance declarations that extend an existing array are no different, as long as the added range doesn't overlap with previous declarations.

Object implementation detail: Ranges of the second form are expanded out in the intermediate format.

Tracking the state of instantiations w.r.t. references.

4.5 References

Referencing arrays can be a little tricky. One may reference dense subsets of either sparse or dense arrays by specifying the precise range of each dimension. An error occurs if not every instance referenced has been instantiated.

For example, given the declaration `int z[100]`, the reference `z[0..9]` would refer to the 1-D array of length 10 containing z[0] through z[9]. `z[0..0]` refers to the 1-D array of length 1 containing z[0], which is *not* size-equivalent to `z[0]`, a 0-D (single instance) reference to z[0], because the number of dimensions do not match.

Indexing an array with a single `pint` results in a size-type of one less dimension, in other words, every dimension singly indexed is *collapsed*. Dimensions indexed with an explicit range are *preserved*. Consider the following examples, given that `y` is a 2-D array:

- `y[i][j]` is a 0-D array, or single instance
- `y[i..i][j]` is a 1-D array of size 1
- `y[i][j..j]` is a 1-D array of size 1
- `y[i..i][j..j]` is a 2-D array of size 1 x 1

4.5.1 Implicit size

For convenience, one may refer to the entire collection of an array with just the name of the instance. However, such an *implicit* collection reference is valid if and only if the implied collection (or sub-array) is densely packed.

some valid and invalid examples References with collapsing dimension:

```
int y[3][4][5];
int z[2][3];
z[0..1][0..2] = y[2][2..3][0..2];  // y ref. collapses 1st dim.
z[0..1][0..2] = y[1..2][1..3][3];  // y ref. collapses 3rd dim.
z = y[0][2..3][0..2];
int x[2][3];
x = z;
```

(TODO: split the above example into several small ones...)

The idea of implicit collections extends to higher dimensions as well. Supposing the first m out of n dimensions are indexed, with k unspecified dimensions, then the number of dimensions of the reference is $m - d + k$ or $n - d$, where d is the number of collapsed dimensions in m. In addition, for such a reference to be valid, the set of subarrays rooted at the nodes indexed by the first m dimensions must all be *range-equivalent*, not just size-equivalent. Implicit array references that do not satisfy this are considered errors.

[tons of examples]

4.6 Aggregates

For convenience, we often want to reference a collection of instances or expressions, and group them into the same entity. This section describes the various modes of *aggregate* references supported by the HAC language.

complex-expr-term : *array-construction* | *loop-concatenation* | *simple-expr*

(Here, *simple-expr* is actually a *shift-expr* in the grammar, to eliminate ambiguity with repsect to the < and > operators, which also wrap around template arguments.)

The following subsections explain the various array-related constructs.

4.6.1 Array concatenation

The syntax for concatenating arrays:

array-concatenation : *array-concatenation* # *complex-expr-term* | *complex-expr-term*

Semantics: The result is an array of the same dimensionality as its constituents, but the size of the first dimension is the sum of the first dimension of its constituents. The constituents, therefore, cannot be scalars. In 2 or higher dimensions, the sizes of all trailing dimensions (past the first) must match to form a valid concatenation. This construction is valid for both meta-expressions and instance-references.

Example: 1D-arrays of size [M], [N], [P] would be concatenated to form a 1D-array of size [M+N+P]. 2D-arrays of size [M][Q], [N][Q], [P][Q] would be concatenated to form a 2D-array of size [M+N+P][Q].

Status: implemented

4.6.2 Array construction

The syntax for building higher-dimension arrays:

array-construction : `{` *construction-list* `}` *construction-list* : *construction-list* `,` *complex-expr-term* | *complex-expr-term*

Semantics: This construct takes N-dimension references and produces an $N + 1$-dimension array of references. Technically, each element is first promoted one dimension (creating an $N + 1$-dimension array with leading dimensions size [1]) and the results are then concatenated. The following are equivalent: `{ x, y }` vs. `{x} # {y}`. The dimension constraint for matching trailing dimensions also applies here. This construction is valid for both meta-expressions and instance-references.

Example: This is most commonly used for grouping scalars into a 1D-array.

Status: implemented

4.6.3 Loop concatenation

The syntax for loop-style concatenation:

`(` `#` `:` *identifier* `:` *range* `:` *complex-expr-term* `)`

Semantics:

Example:

Status: not yet implemented, low priority.

4.7 Issues

Interpreter vs. compiler.

Tracking what instances are available at the point-of-reference.

5 Processes

Processes are the building blocks of concurrent programs. In HAC, processes describe the execution of a single type of entity. The behavioral description can be very high-level, or it may be as detailed as transistor netlists.

5.1 Declarations

One may declare a new type of process without specifying its definition, like a prototype in C/C++. A process declaration contains only the name of the process type, and a port specification with an (optional) list of formal instances.

5.1.1 Ports

A process declaration may be repeated any number of times as long as the port formal instances are equivalent.

Two port formal instance lists are equivalent if and only if the following are true:

1. The list contains the same number of formal instances.
2. Each formal instance (in order of each list) is type-equivalent (and size-equivalent).
3. Each formal instance has the same name.

Unlike C, where formal identifiers are optional in prototypes, port formal lists require names for each instance. This allows one to reference a process's ports individually before the process is defined.

Unlike normal instantiations found in a namespace or definition body, formal instance arrays may not be extended with re-declarations. Since they may only be declared once, they must be densely packed.

5.1.2 Forward declarations

Only the name of the process type is declared.

Also include template signature, covered later.

Not yet supported.

5.2 Definitions

A process definition specifies a body in addition to a port specification.

If a process definition is preceded by a declaration with the same name, then the definition's port specification must match those of the prototype, i.e. each port formal instance must be type-equivalent between the declaration and the definition. Likewise, declarations that follow a definition must also declare the same port formal instances.

5.2.1 Process definition body

The body of a process definition describes a sequence of actions taken by the process.

Refer to CHP chapter, appendix.

Also HSE, PRS.

6 Channels

Processes communicate to each other via channels. Channels are an abstract notion of a point-to-point medium of communication between sender and receiver. (HAC does not yet support multi-sender or multi-receiver communication primitives.)

6.1 Sending and Receiving

Without going into the details of channels, we can define a notion of directionality for channels. Suppose we have some channel type `chan(bool)`.

```
chan(bool) X;
chan?(bool) Y;
chan!(bool) Z;
```

This declares a channel `X` with unspecified direction (nondirectional), `Y` as a read-only (receive-only) directional channel, and `Z` as a send-only directional channel. Send-only and receive-only channels are the most useful in process port declarations — what use are uni-directional channels in the global or local scopes[1]?

6.1.1 Connections and Directions

What are legal connections between channel instance references? Channels have directional connection semantics. Two value producers cannot be aliased, nor can two consumers.

Q: Should legal programs be restricted to connecting at most one-receiver to one-sender?

A: Yes, until we support shared channels (see below).

Q: Should dangling channels (one-way only) be allowed, rejected, or warned? Does not count definition ports, which are assumed to be connected.

Clarification: Channel directions indicated in port declarations only dictate which direction one *cannot* connect to locally.

We use the following example program template to answer questions.

```
defproc inner(chan(bool) A; chan?(bool) B; chan!(bool) C) { ... }
defproc outer(chan(bool) P; chan?(bool) Q; chan!(bool) R) {
  inner x(...), y(...);
}
```

- `x(P,Q,R), y;` – legal
- `x, y; x.A = P; x.B = Q; x.C = R` – legal, equivalent to the previous
- `x, y(x.A, x.B, x.C);` – error: two receive ports being aliased
- `x(P,R,Q), y;` – error: connecting send-only to receive-only channel
- `x, y(x.A, x.C, x.B);` – error: equivalent to above

Goal: precise set of rules for channel connections. Q: does it depend on whether or not referenced channel is local vs. port?

Proposal: a read-only port can take a directionless or read-only port (forwarded) channel as an argumnent. (Likewise for send-only ports.) (Update: accepted.)

[1] Should multi-module linkage ever be specified and implemented, a receive-only channel in one module could connect to a send-only channel in another module.

Should we allow alias connection syntax for channels? Aliasing is a symmetric relation, whereas port connections *need not* be. Yes, but with the following additional semantic constraints:

Proposal: Unidirectional channels should not be referenceable as aliases, only connected through ports. Only nondirectional channels may use alias syntax. Thus, directional channels may only be connected by passing port arguments. Consequences: every physical channel must be connected to at least one nondirectional channel. A send-receive pair must be connected using a nondirectional channel of the same type. (Status: obsolete, in favor of the proposals below)

Proposal: A nondirectional channel may be connected to only one read-only channel and only one send-only channel. (Update: accepted, with possible exception of explicitly shared channels, proposed below.) The following example would be rejected:

```
defproc bucket(chan?(bool) S) { ... }
chan(bool) R;
bucket a(R), b(R);
```

Implementation detail: As aliases are built using a union-find, make sure the canonical node always knows what direction of channels have been connected (propagation). We will track this with a set of flags indicating whether a channel is already connected to a producer or consumer. Note, however, that send/receive use of a channel in CHP body counts as connecting a consumer/producer, respectively. Thus, channel connection checking should include a final pass over the CHP's unrolled footprint.

Proposal: Reject local channel declarations with directional qualifiers. Rationale: It doesn't make sense to have a uni-directional channel in a local scope because any connection or use thereof would result in a block. (Could this be useful for debugging, e.g. causing intentional deadlock?)

Proposal: Reject dangling channels. Channels that are missing connection to a producer or consumer should be rejected. Basically, when a channel is deduced as dangling, at least a diagnostic should be issued. **Resolution:** issue warning diagnostic, without rejecting outright. Eventually, it is up to the final tool to accept or reject dangling connections.

Proposal: Shared channels: Thus far, we've described one-to-one channels where producers and consumers are exclusively paired. In some exceptional circumstances one might desire to share a channel among multiple senders or multiple receivers, where exclusive access is to be guaranteed by the programmer. The following semantics are proposed for sharing channels: A channel is allowed to be connected to multiple receivers, if and only if all participating receivers agree to share, by some implementation of 'agree.' Likewise, multiple senders may share a channel, if and only if all participating senders agree to share. One end of communication on a channel is indifferent to whether or not the other end is shared; a non-shared sender however may connect to shared receivers, and a non-shared receiver may connect to shared senders; To mix shared and non-shared uses on the same end is considered an error.

Rationale: Since channel sharing is exceptional, we want to prevent inadvertent sharing of channels. Non-shared channels expect exclusive use of the channel, so to share them would violate the fundamental assumption.

Syntax: a port declared with `??` or `!!` indicates that the channel may be shared (by receiving or sending). A `??` port channel may be connected locally to multiple receivers, and

a `!!` port channel may be connected locally to multiple senders. When referenced externally, as a member of process, `??` ports may share the same channel as multiple receivers, and `!!` ports may share the same channel as multiple senders.

TODO: examples.

Q: should process aliases be allowed, if processes declare directional channel ports? (Without further modification, they are rejected when attempting to connect ports recursively.)

```
defproc sink(chan?(bool) X) { }
sink x, y;
x = y; // results in x.X = y.X
```

Q: What is the initial 'connected' state of a non-directional channel port (inside definition)? A: a read-only port is assumed to be connected to a producer, and a write-only port is assumed to be connected to a consumer.

Q: Should local 'connected' state information be propagated from formal collections' aliases to actual collections' aliases? With this, channels that are locally dangling can be properly connected hierarchically. A: Yes, we propagate local connectivity summaries up through formals to initialize the connectivity state of substructure actuals for more precise hierarchical alias analysis. This may help check connectivity of non-directional channels. NOTE: this would also apply to relaxed actuals, and is a necessary step towards properly implementing them (a general mechanism to propagate local information externally). TODO: illustrative examples.

Q: Do we allow a channel (member of an array) to be referenced by both meta-parameter and nonmeta-parameters? No. Rationale: Nonmeta indexed channels introduce another difficulty, as they cannot be analyzed at compile time... Effectively, this restricts mixing of meta-aliasing and nonmeta-referencing, at least for channels, variables may be another issue. This issue is related to alias disambiguation.

6.2 Fundamental Channel Types

Until now, we've used `chan(bool)` without discussing its meaning.

6.3 User-defined Channel Types

User-defined channel types are dsecriptions of physical implementations of abstract channels and interfaces.

6.4 Issues

This section is the most of asked (but not necessarily answered) questions pertaining to the channel aspects of the HAC language.

6.4.1 Typedefs

Q: Should channels be typedef-able?

6.4.2 Channel Relaxed Templates

Q: Should channels ever involve relaxed template arguments? A: No. Can't see a good reason for allowing channel type to vary within a higher dimension collection. This applies

recursively to data-types which fall into the channel's type specification. Can processes every be involved in channel type? Probably not.

7 Datatypes

Datatypes are physical representations of information. Or not.

TODO: this must spell out structs and user-defined datatype implementations as distinct notions.

7.1 Built-in datatypes

Currently, HAC has two built in datatypes: `bool` and `int`. These are not to be confused with the parameter types `pbool` and `pint`. A `bool` represents the state of a physical or logical node. An `int` is simply an array of `bool`s with an integer interpretation.

7.1.1 Booleans

In the nonmeta language, assigning a `pbool` value to a `bool` is common and legal. The compiler should eventually resolve all meta-parameter values to constants, which would result in assigning a boolean constant to a `bool` data instance.

In the following example, the `constant<true>` and `constant<false>` types would unroll as one-time assignments to different values. (See Chapter 14 [CHP], page 43, for the description of the CHP language.)

```
template <pbool B>
defproc constant(bool b) {
  chp { b := B }
}
```

Nonmeta operators: e.g. in CHP.

Defect: Currently `bool` is overloaded to mean different things in different contexts. It is (tentatively) both a physical type representing a single node or net, and abstract data type representing a boolean value (whose implementation may be multi-node, such as dual-rail.) In a data type context, `bool` is synonymous with `int<1>`. There is an ambiguity with '`defproc foo(bool b) {...}`', where it is not known without additional context whether `bool` refers to the physical type or the data type. Until implementation-interface semantics are better defined and implemented, we must allow CHP to manipulate physical `bool`s directly. Eventually, we would like an unambiguous use of `bool`, which may require introducing a typedef for `int<1>`, such as `dbool`, either built-in or library-defined.

7.1.2 Integers

The `int` type can take an optional *width* parameter that specifies the *physical* number of bits used to represent the integer. An integer value need not necessarily be encoded in two's-complement; one may use more bits to encode more abstract values (like "not-an-number"), or employ error-correcting codes. The default width of an `int` is 32[1]. Technically speaking, `int` is a built-in templated (parameterized) datatype definition. Templates are discussed in more detail in Chapter 9 [Templates], page 25. The specify an `int`'s width, one can write `int<`*pint*`>`.

For integer type-checking in the nonmeta language, the assignment of a `pint` value or constant to an `int` of *any* width is legal. Implementation detail: this is accomplished by using `int<0>` as a magic width type for meta-valued integers and constants.

[1] 32 was chosen arbitrarily.

Arithmetic and relational operators: In CHP, the standard arithmetic operations on `int` types interprets the bits as signed two's-complement integers. Operator overloading is not yet supported for user-defined datatypes, but may be in the future.

Update: `int<1>` is equivalent to `bool` in a data type context.

7.2 Enumerations

Enumerations are sets of values associated with user-specified names. The value members of an enumeration represent a set of logical values that only have meaning in the enumeration's context, i.e. they are not publicly observable values. (This is unlike C, in which enumerations can take integer values that can be passed to and from integer variables.) Enumerations are particularly useful for specifying control and data interfaces between communicating processes. Think of enumerations as tags that can be understood by the sender and receiver of the enumerated type.

The only values that an enumerated instance can take are those specified in the enumerated type. Thus one can never assign an integer or boolean value to an enumeration, nor can one assign an enumerated value to an `int` or \bool\ or other user-defined type. One can only compare enumerated values of the same enumerated type.

There is no notion of equivalence between enumerated types (outside of typedefs, Chapter 12 [Typedefs], page 39).

7.3 User-defined datatypes

In HAC, one can define arbitrarily complex datatypes. User-defined datatypes resemble structs in C.

Actually, the user-defined datatype implementations are NOT the same as plain structs. They describe an *implementation* of an abstract data type with physical types.

7.3.1 Declarations

7.3.2 Definitions

7.3.3 Views

Views are a way of sub-typing datatypes.

Views are simply specific interpretations or refinements of a datatype.

8 Namespaces

To avoid name collisions, HAC supports namespaces. Namespaces are also useful for organizing closely related definitions and instantiations. HAC allows namespaces to be nested, although one-level (flat) namespaces may be emulated by name-mangling.

The default namespace is the *global* namespace, the ultimate root of all user-defined namespaces. The global namespace has no parent namespace.

See Appendix C.10 (Technicalities:Namespaces) of Stroustrup's *The C++ Programming Language*.

8.1 Identifiers

Identifiers may be prefixed with namespaces using the scope (`::`) operator, e.g. '`zoo::aquatic::fish`'. Any identifier that begins with `::` is an absolute identifier, one whose namespace path is specified from the global namespace. Identifiers fall into one of several categories:

- unqualified, in which the parent namespace is implicit
- qualified-relative, where an identifier is prefixed with a partial namespace path (e.g. '`dinner::food::salad`')
- qualified-absolute, where the full path to a given entity is specified in the identifier's prefix. (e.g. '`::plane`' or '`::farm::horse`')

8.2 Importing

To make all public definitions within a namespace available using unqualified identifiers, one can import an entire namespace.

Aliasing is like psuedo subnamespace.

Each time a namepace is closed, all of its imported namespaces are discarded, which means that the next time that namespace is opened, it loses its previous imports. This can be useful for resetting imports, should the need or desire arise.

Since the global namespace cannot be closed, import directives in the global namespace last until the end-of-file.

8.3 Resolution

In this section, we describe the order in which namespaces are searched to resolve identifiers. Given an unqualified identifier, references are resolved as follows:

1. Lookup the identifier in the current namespace (where the reference is made). If a match is found, it is guaranteed to be unique, otherwise, continue searching.
2. Lookup the identifier in each imported and aliased namespace. If a unique match is found, return it. In the case of multiple matches (in different namespaces), report ambiguity as an error. If no matches are found, continue searching.
3. If not already in the global namespace, continue searching using steps 1 and 2 in the *parent* namespace. Return the first unique match found or an error if zero or more than one match is found.

8.4 Issues

Interpretation:

What does it mean for a parameter assignment or connection to appear in a namespace? Doesn't make any sense. Therefore forbid assignments and connections (actions) inside namespaces. Namespaces may include instantiations and definitions and other namespaces.

Implementation:

9 Templates

Your quote here.

– Bjarne Stroustrup

9.1 Terminology

Arity of a template signature is the number of parameters, or degrees of freedom. Non-template definitions are said to have arity zero. Later we will also refer to the arity of template specializations. We will use $|A|$ to denote the arity of a template (including specialization).

9.2 Forward declarations

Not yet supported. The idea is to declare only the template signature of an identifier, without declaring its ports. Much like the following in C++:

```
template <template <int, class> class>
class my_template_class;
```

'`my_template_class`' is a class that takes a class that takes an integer and a class as a template argument as a template argument.

9.2.1 Template signature equivalence

The formal parameters of a forward signature are allowed to have identifiers, which facilitates latter parameters depending on former parameters. Forward declarations are equivalent if the prototype name matches (in the same namespace) and their template signatures are equivalent. The following forward declarations are equivalent (in C++):

```
template <int> class foo;
template <int P> class foo;
template <int Q> class foo;
```

However, only the identifiers used in template class *definitions* may be referenced from within the definition.

```
template <int R>
class foo {
  /* R may be referenced as the first parameter */
};
```

Note that nowhere outside of the definition, can template parameters be referenced (just as in C++). In \hac, one may declare an instance of a declared but undefined type, which may not necessarily contain any named parameters.

```
foo<7> bar;
int N = bar.R; // ERROR: no public member named R
```

Here is an example of equivalent template signatures:

```
template <int N, int [N]> ...
template <int N, int A[N]> ...
template <int N, int B[N]> ...
template <int M, int A[M]> ...
```

```
template <int M, int B[M]> ...
```

In all cases, the first parameter must be named because the second parmeter depends on the first. Since nothing else depends on the second parameter, its name is optional. Again, only the parameter names used in the definition may be referenced from within the definition.

The same rules in this section (ripped off of C++) pertain to process, channel, and datatype template definitions in HAC.

9.3 Default values

In C++, default values can only appear as a suffix formal parameter list. In HAC, we allow default values in any position of the formal parameter list. However, each defaulting argument position in a type reference must be given a placeholder (a blank space between commas), even if it is at the end of the argument list.

9.4 Type-equivalence

NEW: In HAC, there are two subclasses of template parameters: *strict parameters* are required to be equivalent for type-matching, whereas *relaxed parameters* are ignored when type matching. We refer to the *strict type* of an entity as the underlying template type with fully-specified strict parameters, disregarding its relaxed parameters. The rationale for making this distinction is that frequently, one wishes to declare sparse or dense collections of the same logical type while allowing some internal variation. For example, ROM cells hard-wired to 0 or 1 are permitted in the same collection, and would be templated with one relaxed parameter for the cell's value[1]. The user has the freedom to decide what parameters considered relaxed.

The proposed syntax for distinguishing the two types of parameters is:

template-signature:

- `template <` *strict-parameter-list* `>`
- `template <` *strict-parameter-list*$_{opt}$ `> <` *relaxed-parameter-list* `>`

Both parameter lists are syntactically identical. When there are no relaxed parameters, the second set of angle-brackets may be omitted. If there are only relaxed parameters, the first set of angle-brackets are still required but empty.

There are two levels of type equivalence. Two entities are *collection-equivalent* or *collectible* if their underlying template type is the same and their strict parameters are equivalent, but not necessarily their relaxed parameters. Two entities are *connection-equivalent* or *connectible* if they are collectible, their relaxed parameters are equivalent, and their public interface (port-for-port) is connectible, i.e., the dimensions, sizes, and strict types of the ports are themselves, connection-equivalent. (Note the recursive definition.) Connectibility implies collectibility, but not vice versa.

TODO: denotational semantics

Definition: a type is *complete* is it is either a strict type, or a relaxed type with its relaxed parameters bound. Connectible-equivalence is the same as equivalence between complete types.

[1] The motivation for this comes from the fact that in CAST, environment sources could not be arrayed if they differed in values.

DEBATE: (Resolved, see above: NEW) Should the fourth criterion be required for type-equivalence, or can we allow non-equivalent template parameters as long as the ports interface is equivalent?

Implementation requirement: type errors must be caught by the end of the create-phase of compilation.

Definition requirement: For a template definition to be well-formed, all instances (aliases) local to that definition, including ports, must have complete type. Note that a given definition template may instantiate a definition for each set of unique template parameters, some of which may be meet this requirement, others which may not. [Quick examples of valid and invalid definition?]

Syntax and semantics: The basic syntax for explicitly binding instance alias type parameters is:

- *instance-reference* `<` *relaxed-parameter-list* `>`

The instance-reference may reference a collection of aliases (with ranged specification), in which case the same relaxed parameters are bound to each referenced alias. (Technically, member references could be rejected in this context because all ports are required to have complete type by construction.)

Application: Arrays. The rationale for introducing relaxed types is to be able to declare an array of elements whose members are not identical, though their interfaces remain compatible. The following valid example shows how relax-typed arrays are declared:

```
template <pint X><pbool B>
defproc foo(...) { ... }

foo<2> bar[2];
bar[0]<true>;  // bind relaxed parameter
bar[1]<false>; // bind different relaxed parameter
```

Had one of the parameter-binding statements been omitted, compilation would eventually result in a error indicating that the unbound instance had incomplete type.

Syntactic sugar.

`foo<1> bar<true>;` is equivalent to `foo<1> bar; bar<true>;`.

`foo<1> bar<true>(...);` is equivalent to `foo<1> bar; bar<true>; bar(...)`.

`foo<1> bar<true>[N];` is equivalent to `foo<1> bar[N]; bar[0..N-1]<true>;`.

Aliasing and collection strictness. Consider the following declarations: `foo<1><true> bar;` and `foo<1> car<true>;`. `bar` cannot be aliased to `car` because the collections (even though they are scalar) have different strictness. Strictly speaking, their respective array types are `foo<1><true>` and `foo<1>`, which are not equivalent. For a well-formed connection between two aliases, the collections of the respective aliases must be equivalent, *match in strictness*, and the relaxed types bound to each alias must be equivalent or compatible. In other words, members of strictly typed collections (arrays) cannot alias members of relaxed typed collections.

Implicit type binding though connections. When two aliases are connected to each other, the connection is valid if, in addition to their parent collections' types being equivalent, one of the following holds: both aliases are bound to equivalent parameters or at least one

alias has unbound parameters. When an alias is formed, the relaxed type parameters are automatically 'synchronized,' which effectively binds the type of aliases through other aliases of bound type. [Examples from test suite needed.]

Implementation detail: As soon as an instance alias is bound to a type, it is instantiated. All (reachable) aliases thereof are also instantiated and recursively connected. Attempts to reference members of type-unbound aliases *can* be rejected as errors. (Theoretically, since the public ports may not depend on relaxed parameters, such references may not be treated as errors in the future. It is possible to instantiate the ports before the type is bound, however, internal aliases cannot be replayed until the type is bound, either explicitly or implicitly. The current policy is just a conservative approximation of the eventual operational semantics.)

Type propagation through ports. Since definition members and ports are required to have complete type, the relaxed parameters must be propagated from the formal definition to each instantiating context. Consider the example:

```
defproc wrap(foo<1> x) {
x<false>;
}
wrap Z;
```

upon instantiating `Z` in the top-level, `Z.x` has type `foo<1><false>`.

Restrictions. Where are relaxed parameters forbidden? In port specifications, types of ports and their array sizes may not depend on relaxed parameters, i.e. they may only depend on strict template parameters. This guarantees that the port interfaces remain the same among members of a relax-typed array. In typedef templates, the canonical type's strict template arguments may not depend on relaxed parameters. (Why not? Allowing so would give a means of subverting template parameter strictness.) Anywhere else in the body of a definition template, relaxed parameters may be used freely, within the constraints of the fundamental typing rules.

When should relaxed template parameters be used? General guideline: when the port interfaces do not depend on the said parameters.

9.4.1 Template Examples

The following example illustrates how one might describe a ROM array using relaxed template arguments.

```
template <> <pbool VAL>
defproc ROMcell(...) { ... }

template <pint X, Y> <pbool V[X][Y]>
defproc ROMarray(...) {
  ROMcell<> x[X][Y];  // <> is optional
  // can't reference x[i][j].member yet
  // because instance types are incomplete
  (i:X:
    (j:Y: x[i][j] <V[i][j]> (...); )
  )
  // from here, may reference x[i][j].member
```

```
}

ROMarray <2,3>
    a <{{false,true,false},
      {true,false,true}}>;
```

This following example shows how one would declare an array of sources (like for a test environment):

```
template <pint N>
defchan e1of (...) {...}

template <pint N> <pint M; pint V[M]>
defproc source_e1of (e1of<N> c) {...}

e1of<4> C[10];
source_e1of<4> CS[10];  // instances' types are incomplete
CS[0] <1, {2}> (C[0]);  // can complete types and connect
CS[1] <3, {2,0,1}> (C[1]);
CS[2] <2, {1,0}> (C[2]);
...
```

We could have also declared the array of sources with sparse instantiation, as long as the strict template arguments match:

```
source_e1of<4> CS[0..0];
  // this determines the entire collection's strict parameters
  // but sets the relaxed parameters for only the indexed range
CS[0]<1, {2}>(C[0]);
  // This binds the relaxed template parameters and connects ports.
source_e1of<4> CS[1..1];
CS[1]<3, {2,0,1}>(C[1]);
source_e1of<4> CS[2..2];
CS[2]<2, {1,0}>(C[2]);
...
```

[TODO: write section on connection examples]

9.5 Type Parameters

TODO: This section is in consideration for future extension.

Up to this point, the template parameters covered are *valued* parameters. We now introduce *type* parameters.

Some examples:

- `template <datatype D> defproc ...`
- `template <chan C> defproc ...`
- `template <proc P> defproc ...`

9.6 Template Template Parameters

Punt, I mean it. (What do you think this is, C++?)

9.7 Template Specialization

MAJOR PUNT.

Semantic contraint: The forward declaration of a general template must precede any declaration of any specialization with the same family (same name).

Also partial specializations.

See Section 9.10 [Template Definition Bindings], page 30, for the complicated issues regarding template specialization.

RESOLVE: Should we impose any restrictions on whether or not template definitions may be specialized, and if so, where?

For example: forcing all forward declarations of specializations for a particular definition family would solve the problem of specialization coherence, but would make such a family of definitions unextendable for future specializations – not all specializations can be declared up-front! Implementation consideration: For definition families consisting of only specializations (genericless), this can introduce a lot of overhead in having to process specializations irrelevant to a particular compilation unit.

Impose constraints on definitions, perhaps some invariant constraints and relations between generic and specialized definitions? (e.g. contains same named instances) But this would unnecessarily restrict variations in implementation of certain definitions.

Forward declarations of specializations.

RESOLVE: Do ports of specialized definitions have to match that of the generic template definition? Members certainly need not. What are the implications on argument type-checking?

9.8 Partial Ordering of Specializations

Partial and full specializations for a given template definition may be defined on a partial order. Specializations A and B are ordered $A < B$ if and only if all template parameters that satisfy A also satisfy B and $|A| < |B|$. (In English, ...) If $|A| = |B|$, and $A \neq B$, then A and B are not comparable. Examples, please.

9.9 Template Argument Deduction

Not having to specify every (or any) template arguments.

9.10 Template Definition Bindings

Specialization introduces a whole new aspect of complication to the language. When a definition is used to instantiate an object, should it be instantiated with the best-fit definition seen? In C++, the notion of *point-of-instantiation* is used to select the definition. [cite] Roughly, it says that only definitions that are available (complete) before the point in the translation unit are considered for instantiation. This introduces potential headaches when different translation units see different available definitions at different points of instantiation, i.e. C++ has no mechanism for enforcing consistent use of specializations across translation units. The benefit of compile-time binding of definitions is that type-checking of template definitions and uses may be done entirely at compile-time per translation unit.

HAC uses the unroll-phase as the point-of-instantiation for all instances, when all instances are bound to their proper definitions. The consequence of such a choice is that compile-time type checking is *very* limited with respect to template definitions. [Discuss implementation issues.]

RESOLVE: Should the following example be accepted or rejected (as a compilation unit)?

```
template <pbool B>
defproc foo() { }

foo<true> bar;
bool b = bar.x;
```

What if another compilation unit provides a specialization for `foo<true>` with a `bool` member `x`?

Likewise, consider the following similar example:

```
template <pbool B>
defproc foo(bool b) { }

foo<true> bar;
bool b = bar.x;
```

It is conceivable for a specialization to be defined later without *b* as a bool port. Should the last connection statement be accepted at compile-time? If so, does that constrain the specializations that may be introduced later?

PROPOSAL: A separate bind-phase to allow full type-checking of a compilation unit (perhaps with references to available definition families).

IDEA: Allow introduction of new template specializations that *do not interfere* with pre-determined bindings. Rationale: many full-specializations are introduced for one-time use and do not interfere with other instantiations' bindings.

Binding an instantiation to a definition is recursive: i.e. all members and sub-instances of a bound instantiation must already be bound. Implementation issue: bind-if-possible to automatically bind dependent instantiations.

Implementation option: eager or early binding to force definition binding and thus allow type-checking of a compilation unit that uses template definitions.

IMPLEMENTATION: Is there a need to track the instantiation statements that *used* particular specializations? Need to somehow catch inconsistent views of specializations...

BOTTOM LINE: Type-references to template definitions MUST have a consistent view of definitions at instantiation (unroll) time.

9.11 Issues

As useful as templates may seem, they can't be thrown together without expecting some complications in the language. We use this section to slap down issues that may arise. Once these issues are resolved, the text for their resolutions belong to some sort of "rationale" document, possibly in footnotes or appendices in this language specification.

9.11.1 Relaxed Parameters

In a sequential scope, an instance of relaxed type may have its relaxed actuals bound at any time (but once only). This means that at the time of unbound instantiation (during unroll), the relaxed parameters will not be available for use with unrolling. However, at the point of instantiation, the public ports of the instance (which should never depend on relaxed parameters) should be made available for connections. Q: how do we unroll ports in this situation? Q: do we need to worry about internal aliases differing between difference different complete types? Perhaps not because internal aliases *should* be replayed at create-time before finalizing footprints.

9.11.2 Template Parameter References

PUNTED. (This is groundwork for template metaprogramming.)

Given a templated definition, such as

```
template <pint N> defproc foo(...) { ... }
```

should the parameter `foo::N` be accessible to the programmer as an rvalue? If not, then should we allow references to internal member values (that may be copies of actual parameters)?

```
template <pint N> defproc foo(...) { pint _N = N; }
pint M = foo<3>::_N;
```

Should all internal meta-values be publicly accessible as rvalues? Allowing access to such variables is the root of the template metaprogramming paradigm in C++.

Forbidding direct references to the template parameters may inconvenience a programmer by having to explicitly copy-propagate all parameters that she wishes to export. It also avoids any issues that arise with *forward declarations* and *typedefs templates*.

Consider the template signature equivalence examples from Section 9.4 [Template Type Equivalence], page 26. Among a set of equivalent forward declarations, which set would be used for lookup? The first? or last? The best answer might be 'none': parameters may only be referenced if the complete definition is available.

(Looking forward to the chapter on typedefs...) Now consider the following typedef declaration, continuing from our previous example:

```
template <pint N> typedef foo<N+1> goo;
pint P = goo<3>::N;
```

`goo` has its own parameter N that 'shadows' the base definition's parameter of the same name. (Whatif `goo`'s parameter was renamed to not collide? Then `goo<3>::N` would clearly have to refer to `foo`'s N.) Either way this is disambiguated, the meaning would not necessarily be intuitive. We should simply forbid direct references to template parameters.

Proposal: I am in favor of (what I just said above)

9.11.3 Template Specializations

Should we allow specializations (in the C++ sense)?

Introduces a whole set of issues with binding of dependent names vs. non-dependent names.

9.12 Future

Compile-time checking of templates, directives.

10 Connections

Connections are a relation between *instance references*. (Connections are established in the meta-language processing of compilation; they are determined at compile-time only.)

Instance references refer to specific entities at unroll-time. Instance references fall into two categories: implicit and explicit. See Chapter 4 [Arrays], page 11.

What does '`a = b`' mean in the namespace or definition context?

- Parameters: the value of b is assigned to a. This is only valid if '`b`' is instantiated and initialized, and '`a`' is instantiated and uninitialized.
- Datatypes (both built-in and user-defined), channels and processes: '`a`' and '`b`' refer to the same instance, in other words, they are *aliases*.

If the types are user-defined, then aliasing is recursive. For example, if the type of '`a`' and '`b`' has members (either public or private) '`x`' and '`y`' internally aliased, the '`a.x`', '`a.y`', '`b.x`', and '`b,y`' are all valid references to the same instance of '`x`' and '`y`''s type. (Implementation: This can simply be accomplished by mapping '`a`' and '`b`' to the same instance, saving the trouble of recursive aliasing, and generating the combinations of names, not that that is ever a problem.)

Since connections and aliases are unrolled, the actual instance objects are not created until all connections have been processed.

Compiler options (proposed to support):

- '`-Wprocess-alias`': warning for connections between process (since the semantics seem arbitrary at this point and are prone to future change),
- '`-Wchannel-connections`': warning for suspicious wrong connections with channels (multiple senders or multiple receivers)

Support for non-alias connections?

11 Attributes

Instance attributes are a way of communicating to other tools that something is *special* about a particular instance. The language provides some attributes for the standard meta-types.

11.1 Bool Attributes

Thus far all attributes on bools (nodes) are boolean valued. The default un-attributed node assumes default values for all known attributes. Boolean attributes are contagious in that once they are set to non-default values, they cannot be unset. Non-default values also propagate bottom-up through ports in the instance hierarchy. (They cannot propagate top-down, for that would violate modularity.) Furthermore, when connecting nodes, the non-default value always dominates (spreads contagiously). Keep this in mind when deciding at which level of hierarchy to attach attributes.

`iscomb` *b* [Macro]
: Nodes are initially `iscomb=false`. If *b* is true, node is marked as being driven combinationally, *regardless* of the actual fanin rule of the node. This can be used to tell other tools to not expect a staticizer on this node. If attribute value is unspecified, default value is true.

`autokeeper` *b* [Macro]
: Nodes are initially `iscomb=true`. If *b* is false, direct other back-end tools to not automatically staticize this node for simulation or netlist generation purposes. If unspecified, default value is true.

`isrvc1` *b* [Macro]
`isrvc2` *b* [Macro]
`isrvc3` *b* [Macro]
: Nodes are initially `isrvc1=false`, `isrvc2=false`, `isrvc3=false`. If *b* is true, label this node in a way meaningful for redundant keeper circuits. If unspecified, argument is implicitly true.

Diagnostic attributes.

`ignore_interfere` *b* [Macro]
: Diagnostic attribute. If *b* is true, then suppress diagnostics about interference (opposing on-pulls) on this node. In simulation, the behavior should remain, just silence warnings. When *b* is omitted, it is assumed to be true.

`ignore_weak_interfere` *b* [Macro]
: Diagnostic attribute. If *b* is true, then suppress diagnostics about weak-interference (on-pull vs. X-pull or X-pull vs. X-pull) on this node. In simulation, the behavior should remain, just silence warnings. When *b* is omitted, it is assumed to be true.

11.2 Channel Attributes

TODO: add some attributes

11.3 Process Attributes

TODO: add some attributes

12 Typedefs

When and where are typedefs useful? Where may typedefs appear? Can be in namespace, or local to a definition.

Two kinds of typedefs, with three syntaces.

typedef-declaration:

1. `typedef` *existing-definition-id identifier* `;`
2. `typedef` *existing-definition-id* `<` *template argument list* `>` *identifier* `;`
3. *template-signature* `typedef` *existing-definition-id* `<` *template-argument-list* `>` *identifier* `;`

In all three forms, the *identifier* is the name of the new typedef. The only prerequisite for the existing definition is that it has already been declared.

The first form is a pseudo-typedef, a pure *definition* name alias, as if one had written `#define` *identifier existing-definition-id* in C-preprocessing, with the existing defininition being a simple identifier. (*existing-definition-id* may be a relative or absolute hierarchical name.) The potential confusion with the first typedef is that *existing-definition-id* identifier may be mistaken for a templated type with all parameter arguments with defaults (the only condition in which template arguments may be omitted from a templated definition).

The second form of typedef substitutes a fully-specified template type (one with all template parameters supplied) with an identifier. This is convenient for reusing a templated type repeatedly without having to copy the arguments.

The third form of typedef is a template typedef (not yet supported in C++) which wraps a partially-specified template type with a new definition, usually (not always) with fewer arguments. This is particularly useful for binding template arguments to make simple template types. If a definition doesn't already have default arguments, this is one way of supplying additional default values for an existing templated definition.

Typedefs can be defined in terms of other typedefs. There is currently no restriction on the number of indirections of typedef-ing.

By construction, the graph of typedef definitions must form a tree, and thus, cannot have cycles. Ultimately, every typedef must be defined in terms of a unique non-typedef *canonical definition*. The *canonical parameters* are the values of the parameters that are eventually passed to the canonical definition, and are evaluated by the transformations defined by each typedef indirection.

Templates typedefs: should they be specializable? Methinks not: that would make things incredibly confusing.

12.1 Type-equivalence

TODO: This section needs to be revised since the introduction of strict and relaxed template parameters! (Should basically forbid relaxed template parameters in typedef templates.) Old text follows:

We extend the notion of *type-equivalence* to include typedefs as follows:

Two instances of typedefs are type-equivalent if and only if they refer to the same canonical definition, and their canonical parameters are equal.

Some implementation hints. Given two typedef instances or references. Find greatest common ancestor. Common ancestor may simplify equivalence deduction.

12.2 Questions

Forward declarations. Can typedefs have forward declarations? Should we allow this? Rather not. "I'm declaring this typedef's name, but it's definition will be bound later..." Beware of typedef cycles!

Can typedefs have external linkage? Can cycles form? But what class of type is it: channel, process? e.g. `extern typedef foo;`

`extern template <...> typedef foo;`

`extern typedef old new;`

Linkage. See Chapter 13 [Linkage], page 41.

Q: Should typedefs ever be templatable with relaxed arguments? A: Maybe only if we guarantee that strict arguments don't ever depend on (escape to) relaxed parameters.

13 Linkage

This is completely unimplemented...

One of the strengths of the HAC language is modularity. The old implementation of the CAST language was a single-pass interpreter. The current implementation of an HAC compiler allows one to compile modules independently and later link modules together into a coherent object file. Modular compilation leads to efficient recompilation, library development...

13.1 Visibility

Definitions and instantiations within a compilation module can either be publicly accessible to other modules, or private and inaccessible. By default, all entities are public, i.e. their uses are *exported*. To make an entity private, simply prefix the first declaration or prototype with the keyword `static`, like in C. To refer to an entity defined in another module, simply prefix a declaration with the keyword `extern`, like in C.

Implementation: Generate automatic headers from implementation files.

13.2 Ordering

13.3 Questions

How does linkage apply to typedefs? Can one make a definition static, but a typedef thereof exported? (Could be useful for simplifying interfaces to definitions...)

14 Communicating Hardware Processes

This chapter describes the CHP sub-language, which is based on Hoare's CSP *ref:csp*. CHP operates in the non-meta language domain of HAC, meaning that the instances and values referenced may be resolved at compile-time, even after instantiation. In fact, most values and references are only resolved at run-time.

14.1 Expressions

This section describes the kinds of expressions that CHP supports.

14.1.1 Value References

CHP describes the computation and communication of data variables over channels. Since CHP describes the run-time behavior of programs, the values referenced are only resolved at run-time, just like in a traditional C program. The data-types referenced may `bool`, `int`, enums, or user-defined (structs) (Chapter 7 [Datatypes], page 21).

The indices used to address values may themselves be run-time variables. For example, in `x[i]`, `i` may be an `int` received over a channel. Operationally, this means we need run-time array bounds checks on indices, and existence checks in the case of sparse arrays.

In CHP, `pint` s are considered `int<32>` values and `pbool`s are considered `bool` values as far as type-checking is concerned. (Proposal: support for wildcard (automatic) widths when interpreting `pint` as `int`.)

Note: the current implementation does not *yet* support ranged references (`x[i..j]`). We don't expect this to be difficult, but implementation will be deferred until this feature is warranted.

UPDATE: nonmeta languages, including CHP, no longer support aggregate instance or value references. This means meta-valued ranges cannot appear in any nonmeta language. Nor can implicit non-scalar collection references appear in nonmeta language. Simply put, all references in nonmeta languages must be scalar (0-dimensional).

Arbitrarily complex indexed and member references are supported in the nonmeta languages, such as CHP. However, it is up to the downstream toolchain to interpret or impose further restrictions on the references. For example, a reference such as `x[pi][j].y[k]`, where `pi` is a meta-valued index and `j` and `k` are nonmeta valued integers could lead to a very difficult synthesis or analysis.

14.1.2 Operators

Standard binary arithmetic operations. Tentative type restriction: operands must be of equal `int` width. Return type is the same as operands.

Old CAST-style syntax for boolean logic operations.

Proposal: use C-style syntax so we may distinguish bitwise from logical operations.

Proposal: operator overloading to define arithmetic on user-defined types. (Then we could call this a Hierarchical Operator-Overloading Object-Oriented Circuit Description Language, or HOOOORCD.) Low priority.

14.1.3 Bit Slices

TODO: add support for bit slices. Add public bit-array ports to the intrinsic `int` definition. Add a built-in (private) type for bits.

14.2 Channels

Chapter 6 [Channels], page 17 presented the notion of channels in the context of general nonmeta languages.

We (provisionally) stated that the fundamental channel types, such as `chan(bool)`, were abstract in that they describe *what* information was communicated over a channel, but not *how* (encoding and protocol).

As far as the CHP level is concerned, the implementation is irrelevant (?) to the concurrent program semantics and functional behavior (and simulation).

Where does the implementation come into play? In automatic production-rule generation and mixed-level simuation involving production rule details.

14.3 Statements

This section describes the various statements that CHP supports.

14.3.1 Communications

To receive data over a channel, one simply writes:

- *CHP-receive* : *channel-reference* `?` `(` *data-reference-list* `)`

For example, `X?(x, w)` means: receive values x and w over the two fields of channel X.

To send data over a channel, one writes:

- *CHP-send* : *channel-reference* `!` `(` *nonmeta-expr-list* `)`

For example, `Y!(y, z)` means: send values y and z over the two fields of channel Y.

The channels referenced in sends and receives may be either fundamental channel types or user-defined channel types. However, the channel reference must be scalar (0-dimensional). The variables in the reference list or expression list must type-check against the fields of the underlying fundamental channel type. See Section 6.2 [Fundamental Channel Types], page 19, regarding fundamental channel types. If any types are template-parameter dependent, then type-checking is deferred until the template types have been instantiated.

Operational semantics: Sends and receives in CHP have blocking semantics, i.e., a communication does not complete until its complement (the other side) is also reached. After both sides of the communication have 'synchronized,' can the communication proceed.

Execution clarification: Suppose we have the statement `X[i]!(y[j])`, where `i` and `j` are nonmeta (run-time) variables. If we reach this program point, and find that `X[i]` is blocked (not ready to send), then we must suspend further execution until one of the following conditions changes:

- `X[k]` receive executes for *some* value `k`
- `i` changes the reference to a different channel

If we were to be precise, and track dependencies dynamically (rather than conservatively and statically), we could narrow the first conditional to only `X[i]` receiving. Only at the time of execution, do we evaluate the value of `y[j]` for sending. The value and reference of `y[j]` is *permitted to change* between the time it is blocked and the time the communication is executed! (Can we trap or alert when this is not intended?)

TODO: Probes (implemented, but not documented yet)

14.3.2 Assignments

The syntax for a variable assignment is simple:

- *CHP-assignment* : *lvalue* `:=` *rvalue*

For example, `x := y` assigns the value of y to x. The *lvalue* must refer to a scalar instance of a data type, while the *rvalue* may be any (nonmeta) expression. The types for *lvalue* and *rvalue* must match.

Execution: just assign the current value of the rvalue to the lvalue. Assignments are atomic, so we need not consider changing references.

14.3.3 Wait

When a wait statement is reached, the program simply waits for a condition to become true before proceeding.

- *CHP-wait* : `[` *CHP-expr* `]`

The expression must, of course, be boolean in value.

Execution: When arriving at a wait event, evaluate the guard expresions. If true, then proceed immediately to the event that follows. Otherwise, block this event pending any change on variables or channels that could possibly change the value of the expression. In implementation, the set may be precise or conservative, but the resulting evaluation must remain equivalent.

14.3.4 Composition

CHP statements may be composed either sequentially or concurrently.

- *CHP-sequence* : *CHP-stmt* `;` *CHP-stmt* ...
- *CHP-concurrence* : *CHP-stmt* `,` *CHP-stmt* ...

Concurrent composition has higher precedence than sequential composition, so $X, Y; Z$ is interpreted as $(X, Y); Z$. However, one may explictly parenthesize $X, (Y; Z)$.

Execution of concurrent branches behaves like a fork and join (barrier). Upon initial execution, each branch is begun concurently, but the execution is not completely until all branches have reached the join-barrier. (This can be easily implemented as a decrementing barrier counter.)

14.3.5 Skip

Skipping this section... haha.

Note on syntax: A skip statement may only appear by itself in a CHP body, i.e. never in a sequential or concurrent composition. It may appear in-place of any *CHP-stmt-list*.

14.4 Flow Control

This section describes the various flow control statements available in CHP.

14.4.1 Loops

Loops never end. Most hardware one will describe with CHP will contain a loop. (What good is a program that only works once?)

- *CHP-loop* : `*[` *CHP-stmt-list* `]`

Execution: after the last action in the loop executes, schedule the first action for execution.

14.4.2 Guarded Commands

- *CHP-guarded-command* : *CHP-expr* `->` *CHP-stmt-list*

A special case of a guarded commands is an else-clause, which replaces the guard expression with the keyword `else`. An else clause may only appear at the end of deterministic selections, but not any other selection statements.

Execution: Interpretation depends on the context in which the guarded command appears, e.g. deterministic vs. nondeterministic selection, or do-while loops.

14.4.3 Deterministic Selection

- *CHP-det-selection* : `[` *CHP-det-guarded-command-list* `]`
- *CHP-det-guarded-command-list* : *CHP-guarded-command* `[]` ...

The guarded command list must contain at least two guarded commands (else it's not a selection). The last guarded statement may be an else clause.

Operational Semantics: (Basically exclusive switch-case.) Only one of the guards is allowed to be true at a time (mutual exclusion). If more than one guard is ever true, then there is an error in the program. A deterministic selection blocks until one of its guards has become true and its guarded commands executed.

Execution: Since branches are executed mutually exclusively, as soon as any branch finishes executing its last event, the events that immediately follow the selection may be processes as if in the same sequence. If at any time more than one guard evaluates true, a diagnostic is required, though signaling and error is recommended.

If initially none of the guards evaluate true, then the selection is blocked until one of them becomes true. (This can be accomplished by registering all dependent variables on a global watch-list. When any variable on the watch-list changes status, then subscribed expressions are re-evaluated, to see if new events may be scheduled.)

If the guards include an else-clause, then this selection never blocks, because the else-clause will guarantee that one clause will execute.

14.4.4 Nondeterministic Selection

- *CHP-nondet-selection* : `[` *CHP-nondet-guarded-command-list* `]`
- *CHP-nondet-guarded-command-list* : *CHP-guarded-command* `:` ...

Note: for the sake of a cleaner grammar, we use `:` instead of `|` to denote a nondeterministic selection. Can nondeterministic selections contain else-clauses?

Operational Semantics: nondeterministic selection blocks until at least one guard becomes true. While any number of guards may be true, one of the true guards is chosen arbitrarily[1] as the path of execution.

Execution: More than one guard is allowed to be true, but only branch is chosen to be executed. Q: Do we use the notion of a time-window before evaluating guards?

14.4.5 Do-While

- *CHP-det-selection* : `*[` *CHP-det-guarded-command-list* `]`

No else clauses allowed.

Operational Semantics: Loop until all guards are false.

Execution: Never blocks because there is an implicit else-clause that skips/exits the loop.

14.5 Metaparameter loop constructs

New syntaces for compile-time expanded repetitive constructs:

Sequential composition

```
{ ; i:N: ... }
```

Concurrent composition

```
{ , i:N: ... }
```

Determinstic selection

```
[ [] i:N: ... ]
```

Nondeterminstic selection

```
[: i:N: ... ]
```

14.6 Extensions

14.6.1 Function Calls

Calling C/C++ functions.

This is better documented in the CHPSIM Manual.

[1] Weakly fair.

15 Production Rule Set (PRS)

This chapter describes the Production Rule Set (PRS) sub-language. PRS operates strictly in the meta-language domain of HAC, meaning that all involved instance references and connections are resolved at compile-time, upon instantiation of each complete definition. Production rules, like connections, may be programmed to depend on meta-language parameters.

15.1 Basics

Basic production rules are written as follows:

- *rule* : *PRS-expr* `->` *node dir*

(Reference to lines of grammar...) (Denotational semantics, type-inference later...)

The *dir* is either `+` (pull-up) or `-` (pull-down). A *literal* is an occurence of a (`bool`) *node* on the left-hand-side of a production rule. A PRS *literal* and the right-hand-side node must be a refer to a single (scalar) `bool` instance. A *PRS-expr* may be any boolean expression using the operators `~`, `&`, `|`, and literals. (The unary `~` operator has the highest precedence, and the `&` operator has higher precedence than the `|` operator.)

The rule arrow `->` can be substituted with one of its shorthand forms. The `=>` arrow automatically generates the complementary rule (pulling on opposite direction) using the DeMorgan inverse of the guard. The `#>` arrow mirrors the rule pulling in the opposite direction with the same topology but *inverted* literals, mostly useful for writing C-elements. For example:

```
x & y & z    => w-
x & y & z    #> c-
```

expands to

```
x & y & z    -> w-
~x | ~y | ~z -> w+
x & y & z    -> c-
~x & ~y & ~z -> c+
```

Rules involving internal nodes may only use the plain `->` notation Section 15.1.2 [PRS Internal Nodes], page 50.

Since production rules are an abstract description of logic, the rules themselves need not be CMOS-implementable. Enforcement of CMOS-implementability can be introduced by later tools or compiler phases where desired. (**TODO**: write a CMOS checking pass.)

15.1.1 Sizing

We provide a way of specifying transistor widths for every literal. Each literal on the LHS may be followed by an optional size argument:

- *literal* : *node-reference* `{ < float > }`$_{opt}$

Ideas: Specify a width/strength on the RHS and automatically infer the sizes of the literals on the LHS (only for this rule).

Actually, literal parameters may be any generalized list of expressions.

15.1.2 Internal Nodes

A literal may be optionally prefixed with '`@`' to indicate that it is only an internal node, and is not declared as a normal bool. Internal nodes are useful for specifying more general circuit topologies that share common foot transistors. Internal nodes may appear on the left-hand-side of production rules arbitrarily many times.

```
en -> @_en_int-          // defines an internal node
~@_en_int & Ld -> _rd-   // uses internal node as foot transistor
~@_en_int & Hd -> _md-
```

Above, `_en_int` is not declared as a bool, but the first rule that drives it effectively declares it – an implicit declaration. An internal node may only be referenced as the leftmost literal of any conjunctive (and) term. (**TODO**: position is not yet checked, currently performs straightforward expression substitution.)

The effective production rules are obtained by substituting the internal nodes' associated expressions wherever they are used. With the above example, the effective production rules are:

```
en & Ld -> _rd-
en & Hd -> _md-
```

Internal nodes are implicitly declared in the scope of the enclosing definition (or top-level) so their names cannot conflict with existing declarations. Likewise, subsequent declarations cannot re-use names of existing internal nodes.

Internal nodes can also be used in arrays, where the dimensions are implied by the indexing. (**Warning**: node arrays are not supported in ACT.) One can declare arrays of internal nodes in loops, for example:

```
(:i:N:
  en & x[i] -> @_en_i[i]-
  ~@_en_i[i] & y[i] -> _z[i]-
)
```

Each unique internal node may only be defined once in one direction, pull-up or pull-down. Using an internal node in the wrong sense constitutes an error, e.g.:

```
x -> @y-
@y & z -> w-
```

is an error because `@y` is defined as a pull-down only expression, but is being evaluated active-high in the rule for `w-`. Negations of internal nodes are *bound to the referenced node* and dictate the sense in which the node is pulled, unlike regular boolean expressions.

The following is an erroroneous attempt to define the same internal node in two directions:

```
x -> @y-
~x -> @y+
```

Rules involving internal nodes may only use the plain `->` notation.

Since internal nodes just define re-usable subexpressions, production rule attributes are not applicable to them; they are simply ignored.

Status: implemented and tested.

15.2 Attributes

We need a clean way to tag nodes and rules with attributes for various tools.

15.2.1 Node attributes

What happens when we connect nodes with conflicting attributes?

Attributes from super-cells or sub-cells?

15.2.2 Rule attributes

We propose the following syntax for per-rule attributes:

- *rule-attrib-list* : `[` *rule-attrib* `;` ... `]`
- *rule-attrib* : *identifier* `=` *expr-list*

Rule-attribs are generalized as key-value(s) pairs, which permits the programmer to add arbitrary attributes to the language without adding more keywords to the language. *Rule-attrib-list*s are just semicolon-delimited lists of one or more rule-attributes. In the case of repeated attributes, the latter pair will override the former. Rule-attribute-lists are optional prefixes to PRS-rules.

For now, the purpose of these attributes is to emit attribute lines suitable for consumption by another text-based tool, such as old versions of `prsim`.

Q: What happens when we OR-combine rules with different attributes?

Some existing attributes:

`after` d [Macro]
: Applies a fixed delay d to a single rule. Affects `hflat` output and `hacprsim` operation.

`after_min` d [Macro]
`after_max` d [Macro]
: Specifies upper and lower bounds on delays for a rule. The upper bound should be greater than or equal to the lower bound, however, this is not checked here.

`weak` b [Macro]
: If b is true (1), rule is considered weak, e.g. feedback, and may be overpowered by non-weak rules. If unspecified, default value is true.

`unstab` b [Macro]
: If b is true (1), rule is allowed to be unstable, as an exception. If unspecified, default value is true.

`comb` b [Macro]
: If b is true (1), use combinational feedback.

`iskeeper` [b] [Macro]
: If b is true (1), flag that this rule is part of a standard keeper. If unspeficied, default value is true.

`isckeeper` [b] [Macro]
: If b is true (1), flag that this rule is part of a combinational feedback keeper. If unspeficied, default value is true.

`keeper` *b* [Macro]
: For LVS, If *b* is true (1), staticize (explicitly). This attribute will soon be deprecated in favor of a node attribute `autokeeper`.

`output` *b* [Macro]
: If *b* is true (1), staticize (explicitly). Q: should this really be a rule-attribute? better off as node-attribute?

`loadcap` *C* [Macro]
: Use *C* as load capacitance instead of inferring from configuration.

`always_random` *b* [Macro]
: If *b* is true (1), rule delay is based on random exponential distribution. If unspecified, default value is true.

`N_reff` *R* [Macro]
`P_reff` *R* [Macro]
: Use *R* as effective resistance to override the automatically computed value in other back-end tools. NOTE: This is a hack that should be replaced with a proper implementation of the "fold" expression macro. Consider this attribute deprecated from the start.

15.2.3 Literal attributes

The literals of the rule expressions may have attributes. Literal attribute are mostly for back-end tool-specific use.

`lvt` [Macro]
: Specifies that a transistor has low-Vt type, mainly for netlist generation and LVS checking.

`svt` [Macro]
: Specifies that a transistor has standard-Vt type, mainly for netlist generation and LVS checking. This is also the default type, when Vt is left unspecified.

`hvt` [Macro]
: Specifies that a transistor has high-Vt type, mainly for netlist generation and LVS checking.

Ideas:

Instance-specific attributes?

Procedural layout?

Automatic sizing

15.2.4 Operator attributes

The `&` operator may take an optional suffix attribute to indicate that an internal node is precharged. For example:

```
en &{+~en} Ld & Ci -> _rd-
```

states that the internal node between gates `en` and `Ld` is precharged with a PFET gated with `~en`.

Status: parsed, but ignored. Should this be done with `|` as well?

15.3 Loops

Loop syntax, unrolling, etc...

Loops can appear in expressions and in rules in the PRS language. A *rule-loop* can be written as:

- `(:` *loop-var* `:` *range* `:` *rules* `)`

The *loop-var* is declared with a identifier, and may be referenced in the body rules. The rule-loop is repeatedly expanded using the values spanned by the *range*. (The range may be written implicitly or explicitly.) If the range evaluates empty, then the body is skipped during unrolling. Rule-loops may be nested, i.e., they may contain other loops. The current limit for the size of an expression is 65535 sub-expressions.

An expression-loop is written as:

- `(` *op* `:` *loop-var* `:` *range* `:` *PRS-expr* `)`

The *loop-var* and *range* have the same meanings as when used in rule-loops. *op* may be `&` or `|`. The body expression is repeatedly expanded with the *op* operator. Expression-loops may be nested, i.e., they may contain other expression-loops. If the range evaluates empty, ... **We need to specify these semantics**!

Interpret the following:

- `(&:i:0: x[i]) -> y-`
- `(|:i:0: x[i]) -> y-`
- `z & (&:i:0: x[i]) -> y-`
- `z & (|:i:0: x[i]) -> y-`
- `z | (&:i:0: x[i]) -> y-`
- `z | (|:i:0: x[i]) -> y-`

15.4 Extensions

This section describes some of the recent extensions to the PRS language.

15.4.1 Macros

As an alternative to a PRS-rule, one may write a macro to represent some custom topology of a netlist or as shorthand for an expansion.

We propose the following syntax for macros:

- *PRS-macro* : *identifier* `<` *expr-list* `>` `(` *PRS-literal-list* `)`
- *PRS-macro* : *identifier* `(` *PRS-literal-list* `)`

We do not hard-code any built-in macros into the language with keywords, rather we allow the programmer to define the meaning of each macro. Macros can also take parameters inside angle-brackets, where the *expr-list* is a list of comma-separated expressions. The number of arguments for a macro may also be variable, and is defined by the macro's implementation. The macro mechanism can potentially be used to attach attributes to nodes and other subnets. As the list of macros grows, they should be documented here.

Examples of macros one may wish to define:

- `passn` — `<W,L>(g,s,d)` W is an optional transistor width, L is an optional length. If only one parameter is passed, it is interpreted as the width. g, s, d are the gate, source, and drain, respectively. The pseudo production rule generated (`cflat prsim`) is *uni-directional*, i.e. the drain is driven as the output.
- `passp` — analogous to `passn`
- `pass` — full symmetric pass-gate
- `assert`
- `stat`
- `comb-fb`

The remaining sections discuss other extensions that have been proposed at other times. See which ones could be folded into a general macro!

Below are a list of macros documented in the source file '`Object/lang/PRS_macro_registry.cc`'.

`passn` *W L g s d* [Macro]

Usage: '`passn<W,L>(g, s, d)`' or '`passn(g, s, d)`'

Declares an NFET pass-transistor with gate g, source s, and drain d. Sizing parameters W and L are optional. In `hflat prsim` mode, this prints a uni-directional (sized) production rule

```
after 0  g & ~s -> d-
```

In `hflat lvs` mode, this just prints '`passn(g, s, d)`' back out. Sizes are printed only if '`-fsizes`' is passed to hflat.

`passp` *W L g s d* [Macro]

Usage: '`passp<W,L>(g, s, d)`' or '`passp(g, s, d)`'

Declares a PFET pass-transistor with gate g, source s, and drain d. Sizing parameters W and L are optional. In `hflat prsim` mode, this prints a uni-directional (sized) production rule

```
after 0 ~g & s -> d+
```

In `hflat lvs` mode, this just prints '`passp(g, s, d)`' back out. Sizes are printed only if '`-fsizes`' is passed to hflat.

`echo` *nodes...* [Macro]

Diagnostic. This macro just prints '`echo(...)`' back out where the original arguments are substituted with canonical hierarchical instance (node) names. This demonstrates how one can add custom PRS macros.

15.4.2 Pass-gates

True pass-gate logic was missing from the original CAST-PRS implementation. A pass-gate could be emulated as a latch if the 'direction' of operation was known at compile time. Not having to support pass-gates greatly simplified other pieces of the tool-chain, such as LVS and PRSIM.

We propose the following syntax for pass-gates:

1. $node_1$ `<-` $node_2$ `->` $node_3$ denotes an NFET connecting $node_1$ and $node_3$ gated by $node_2$
2. $node_1$ `<+` $node_2$ `+>` $node_3$ denotes a PFET connecting $node_1$ and $node_3$ gated by $node_2$

$node_{1...3}$ are production rule literals. In case 1, when $node_2$ is logic-1, the nodes on either side are connected. In case 2, when $node_2$ is logic-0, the other terminals are connected. $node_2$ may be given an optional size to specify the width of the transmission-gate. In all cases, if nodes on opposite sides are both driving in opposite directions, then it is considered a short-circuit (error). If neither side is driving, and the nodes' states are in opposition, then both nodes will become unknown (X). Otherwise, the one side that is being driven will flip the other side.

$node_2$ may be given an optional `<size>` argument for specifying gate width.

Technically, one may use pass-gate to construct arbitrary transistor topologies.

15.5 Options

Compiler warnings:

CMOS-implementability (exceptions allowed for attribute).

Staticizers?

16 SPEC Directives

This chapter describes the various directives available in the `spec` sub-language. The following documentation is extracted from source file '`Object/lang/SPEC_registry.cc`'.

`unaliased` *nodes...* [Directive]

Usage: '`unaliased(...)`'

Error out if any of *nodes* are aliased to each other. Tool-independent. Useful for verifying that certain nodes are not accidentally connected.

`assert` *P* [Directive]

Usage: '`assert<P>()`'

Error out if predicate expression *P* is false. Note that this is a *compile-time* check, which is enforced during unroll/create compilation. Useful for enforcing parametric constraints. Tool-independent. For run-time invariants, see `$(expr)`-syntax below.

`exclhi` *nodes...* [Directive]

Usage: '`exclhi(...)`'

Emits directives to check that *nodes* are mutually exclusive high at run-time. (This corresponds to the old `CHECK_CHANNELS` method of checking for exclusivity.) In `hacprsim`, these form checking rings. In `cflat lvs`, these directives affect charge-sharing and sneak-path analysis.

`excllo` *nodes...* [Directive]

Usage: '`excllo(...)`'

Emits directives to check that *nodes* are mutually exclusive low at run-time. (This corresponds to the old `CHECK_CHANNELS` method of checking for exclusivity.) In `hacprsim`, these form checking rings. In `cflat lvs`, these directives affect charge-sharing and sneak-path analysis.

`order` *nodes...* [Directive]

For `cflat lvs`, specify the node checking order for BDD algorithms.

`unstaticized` *node* [Directive]

For `cflat lvs`, specify that node should remain unstaticized.

`cross_coupled_inverters` *x y* [Directive]

For `cflat lvs`, just emit the directive back out with substituted canonical node names.

`mk_exclhi` *nodes...* [Directive]

For `cflat prsim` and `hacprsim`, enforce logic-high mutual exclusion among *nodes*. This is often used in describing arbiters.

`mk_excllo` *nodes...* [Directive]

For `cflat prsim` and `hacprsim`, enforce logic-low mutual exclusion among *nodes*. This is often used in describing arbiters.

`min_sep` *dist nodes...* [Directive]

Usage: '`min_sep<dist>(nodes...)`'

Specify that *nodes* should have a miminum physical separation of distance *dist*. *nodes* can be organized into aggregate groups: for '`min_sep({a,b},{c,d})`', `a` and `b` must be separated from `c` and `d`. Affects `cflat` for layout and `prsim`.

`runmodestatic` *node* [Directive]

For `cflat lvs`, mark this node as one that seldom or never changes value, for the sake of netlist analysis. NOTE: eventually this directive will be deprecated in favor of applying a proper node attribute. The decision to implement this as a spec-directive is an unfortunate consequence of compatibility with another tool.

Another class of specification directives is *invariants*. Invariants are conditions which should always hold true. Invariants are useful for telling other tools what assumptions can be made about circuits. `exclhi` and `excllo` are examples of invariants that use the normal directive syntax.

`$` *PRS-expr* [Directive]

This declares a run-time invariant expression that is emits invariant directives to back-end tools and also tells simulators to check and report violations of violations, similar to `exclhi` and `excllo`. *PRS-expr* is a production rule guard that should #*always* be true.

```
spec {
  $(~(x & y))
}
```

is equivalent to `exclhi(x, y)`.

Appendix A Keywords

This appendix describes some of the keywords and terminal tokens in the HAC language. The following are all special words recognized by the lexer.

Terminals (tokens):

TODO: extract from `yacc` output file.

Keywords:

`__FILE__` A string that repesents the current file (absolute path).

`__LINE__` Compile-time integer (`pint`) that holds the current line number in the input stream or file.

`import` The directive for including another source file. Automatically ignores the file if it was already read. (implicit pragma-include-once)

`#FILE` Reserved, not intended for general use. Embedded file directive, emitted by flattening a source file's `import` directives recursively. Allows a single file to behave as if sections were included hierarchically.

Appendix B Grammar

> "Yacc" owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right.
>
> – S. C. Johnson, "Yacc guide acknowledgements

This appendix describes the HAC language's grammar. The grammar is context-free and LR(1), so traditional `yacc` and `bison` (LALR(1)) parser generators will work with it.

```
 0 $accept: module $end
 1 module: embedded_module
 2 embedded_module: imports_optional top_root
 3 imports_optional: imports
 4                 | /* empty */
 5 imports: imports import_item
 6        | import_item
 7 import_item: IMPORT
 8            | EMBEDFILE '{' embedded_module '}'
 9 top_root: body
10         | /* empty */
11 body: body body_item
12     | body_item
13 body_item: namespace_item
14          | definition
15          | prototype_declaration
16 namespace_item: namespace_management
17               | instance_item_extended
18               | type_alias
19 namespace_management: NAMESPACE ID '{' top_root '}'
20                     | OPEN namespace_id RARROW ID ';'
21                     | OPEN namespace_id ';'
22 namespace_id: relative_id
23 definition: defproc
24           | defdatatype
25           | defchan
26           | defenum
27 prototype_declaration: declare_proc_proto ';'
28                      | declare_datatype_proto ';'
29                      | declare_chan_proto ';'
30                      | declare_enum ';'
31 type_alias: optional_export optional_template_specification TYPEDEF physical_type_ref ID ';'
32 template_specification: TEMPLATE template_formal_decl_list_in_angles
33                     | TEMPLATE template_formal_decl_list_optional_in_angles template_formal_decl_nodefault_list_in_angles
34 optional_template_specification: template_specification
35                               | /* empty */
36 optional_export: EXPORT
37                | /* empty */
38 def_or_proc: DEFINE
39           | DEFPROC
40 declare_proc_proto: optional_export optional_template_specification def_or_proc ID optional_port_formal_decl_list_in_parens
41 defproc: declare_proc_proto '{' optional_definition_body '}'
42 optional_port_formal_decl_list_in_parens: '(' port_formal_decl_list ')'
43                                         | '(' ')'
44 template_formal_decl_list_in_angles: '<' template_formal_decl_list '>'
```

```
  45 template_formal_decl_nodefault_list_in_angles: '<' template_formal_decl_nodefault_list '>'
  46 template_formal_decl_list_optional_in_angles: template_formal_decl_list_in_angles
  47                                             | '<' '>'
  48 template_formal_decl_list: template_formal_decl_list ';' template_formal_decl
  49                          | template_formal_decl
  50 template_formal_decl_nodefault_list: template_formal_decl_nodefault_list ';' template_formal_decl_nodefault
  51                                    | template_formal_decl_nodefault
  52 template_formal_decl: base_param_type template_formal_id_list
  53 template_formal_decl_nodefault: base_param_type template_formal_id_nodefault_list
  54 template_formal_id_list: template_formal_id_list ',' template_formal_id
  55                        | template_formal_id
  56 template_formal_id_nodefault_list: template_formal_id_nodefault_list ',' template_formal_id_nodefault
  57                                  | template_formal_id_nodefault
  58 template_formal_id_default: ID optional_dense_range_list '=' expr
  59 template_formal_id_nodefault: ID optional_dense_range_list
  60 template_formal_id: template_formal_id_default
  61                   | template_formal_id_nodefault
  62 port_formal_decl_list: port_formal_decl_list ';' port_formal_decl
  63                      | port_formal_decl
  64 port_formal_decl: physical_type_ref port_formal_id_list
  65 port_formal_id_list: port_formal_id_list ',' port_formal_id
  66                    | port_formal_id
  67 port_formal_id: ID optional_dense_range_list
  68 generic_type_ref: generic_id strict_relaxed_template_arguments optional_chan_dir
  69 optional_chan_dir: '?'
  70                  | '!'
  71                  | '?' '?'
  72                  | '!' '!'
  73                  | /* empty */
  74 physical_type_ref: generic_type_ref
  75                  | base_chan_type
  76                  | base_data_type_ref
  77 base_data_type_ref: base_data_type strict_relaxed_template_arguments
  78 data_type_ref: base_data_type_ref
  79              | generic_type_ref
  80 type_id: physical_type_ref
  81        | base_param_type
  82 base_param_type: PINT_TYPE
  83                | PBOOL_TYPE
  84                | PREAL_TYPE
  85 base_chan_type: chan_or_port data_type_ref_list_optional_in_parens
  86 chan_or_port: CHANNEL optional_chan_dir
  87 data_type_ref_list_optional_in_parens: '(' data_type_ref_list_optional ')'
  88 data_type_ref_list_optional: data_type_ref_list
  89                            | /* empty */
  90 data_type_ref_list: data_type_ref_list ',' data_type_ref
  91                   | data_type_ref
  92 base_data_type: INT_TYPE
  93               | BOOL_TYPE
  94 declare_datatype_proto: optional_export optional_template_specification DEFTYPE ID DEFINEOP data_type_ref op-
tional_port_formal_decl_list_in_parens
  95 defdatatype: declare_datatype_proto '{' optional_datatype_body set_body get_body '}'
  96 set_body: SET '{' chp_body_optional '}'
  97 get_body: GET '{' chp_body_optional '}'
  98 declare_enum: ENUM ID
  99 defenum: ENUM ID '{' enum_member_list '}'
 100 enum_member_list: enum_member_list ',' ID
 101                 | ID
```

```
102 declare_chan_proto: optional_export optional_template_specification DEFCHAN ID DEFINEOP base_chan_type optional_port_formal_decl_list_in_parens
103 defchan: declare_chan_proto '{' optional_datatype_body send_body recv_body '}'
104 send_body: SEND '{' chp_body_optional '}'
105 recv_body: RECV '{' chp_body_optional '}'
106 definition_body: definition_body definition_body_item
107                | definition_body_item
108 definition_body_item: instance_item_extended
109                     | type_alias
110 optional_definition_body: definition_body
111                         | /* empty */
112 optional_datatype_body: datatype_body
113                       | /* empty */
114 datatype_body: datatype_body datatype_body_item
115              | datatype_body_item
116 datatype_body_item: connection_body_item
117                   | lang_spec
118 connection_body: connection_body connection_body_item
119                | connection_body_item
120 connection_body_item_base: connection_statement
121                          | nonempty_alias_list ';'
122                          | instance_type_completion_statement
123                          | instance_type_completion_connection_statement
124 connection_body_item: connection_body_item_base
125                     | loop_connections
126                     | conditional_connections
127 instance_management_list: instance_management_list instance_item_extended
128                         | instance_item_extended
129 instance_item_extended: instance_item
130                       | language_body
131 instance_item: type_instance_declaration
132              | connection_body_item_base
133              | loop_instantiation
134              | conditional_instantiation
135 loop_instantiation: '(' ';' ID ':' range ':' instance_management_list ')'
136 conditional_instantiation: '[' guarded_instance_management_list ']'
137 loop_connections: '(' ';' ID ':' range ':' connection_body ')'
138 conditional_connections: '[' guarded_connection_body ']'
139 type_instance_declaration: type_id instance_id_list ';'
140 instance_id_list: instance_id_list ',' instance_id_item
141                | instance_id_item
142 instance_id_item: ID optional_template_arguments_in_angles sparse_range_list
143                | ID optional_template_arguments_in_angles
144                | ID optional_template_arguments_in_angles connection_actuals_list
145                | ID optional_template_arguments_in_angles '=' alias_list
146 connection_statement: member_index_expr connection_actuals_list ';'
147 instance_type_completion_statement: index_expr complex_expr_optional_list_in_angles ';'
148                                   | generic_id complex_expr_optional_list_in_angles ';'
149 instance_type_completion_connection_statement: index_expr complex_expr_optional_list_in_angles connection_actuals_list ';'
150                                              | generic_id complex_expr_optional_list_in_angles connection_actuals_list ';'
151 nonempty_alias_list: nonempty_alias_list '=' complex_aggregate_reference
152                    | complex_aggregate_reference '=' complex_aggregate_reference
153 alias_list: alias_list '=' complex_aggregate_reference
154           | complex_aggregate_reference
155 connection_actuals_list: '(' complex_aggregate_reference_list ')'
156 guarded_instance_management_list: guarded_instance_management_list_unmatched THICKBAR instance_management_else_clause
```

```
157                                 | guarded_instance_management_list_unmatched
158 guarded_instance_management_list_unmatched: guarded_instance_management_list_unmatched THICK-
BAR guarded_instance_management
159                                       | guarded_instance_management
160 guarded_instance_management: expr RARROW instance_management_list
161 instance_management_else_clause: ELSE RARROW instance_management_list
162 guarded_connection_body: guarded_connection_body_unmatched THICKBAR connection_body_else_clause
163                         | guarded_connection_body_unmatched
164 guarded_connection_body_unmatched: guarded_connection_body_unmatched THICKBAR guarded_connection_body_clause
165                                   | guarded_connection_body_clause
166 guarded_connection_body_clause: expr RARROW connection_body
167 connection_body_else_clause: ELSE RARROW connection_body
168 language_body: CHP_LANG '{' chp_body_optional '}'
169               | HSE_LANG '{' hse_body_optional '}'
170               | PRS_LANG optional_template_arguments_in_angles '{' prs_body_optional '}'
171               | lang_spec
172 lang_spec: SPEC_LANG '{' spec_body_optional '}'
173 chp_body: full_chp_body_item_list
174 chp_body_optional: chp_body
175                  | /* empty */
176 chp_body_or_skip: chp_body
177                 | SKIP
178 chp_sequence_group: '{' full_chp_body_item_list '}'
179 full_chp_body_item_list: full_chp_body_item_list ';' full_chp_body_item
180                         | full_chp_body_item
181 full_chp_body_item: chp_concurrent_group
182 chp_body_item: chp_statement_attributes chp_statement
183              | chp_statement
184 chp_statement_attributes: '$' '(' chp_statement_attr_list ')'
185 chp_statement_attr_list: chp_statement_attr_list ';' chp_statement_attr
186                         | chp_statement_attr
187 chp_statement_attr: ID '=' expr
188 chp_statement: chp_loop
189              | chp_do_until
190              | chp_selection
191              | chp_wait
192              | chp_binary_assignment
193              | chp_bool_assignment
194              | chp_send
195              | chp_recv
196              | chp_peek
197              | LOG expr_list_in_parens
198              | chp_metaloop_selection
199              | chp_metaloop_statement
200              | function_call_expr
201 chp_loop: BEGINLOOP chp_body ']'
202 chp_do_until: BEGINLOOP chp_unmatched_det_guarded_command_list ']'
203 chp_wait: '[' chp_guard_expr ']'
204 chp_selection: '[' chp_matched_det_guarded_command_list ']'
205              | '[' chp_nondet_guarded_command_list ']'
206 chp_metaloop_selection: '[' ':' ID ':' range ':' chp_guarded_command ']'
207                       | '[' THICKBAR ID ':' range ':' chp_guarded_command ']'
208 chp_metaloop_statement: '{' ',' ID ':' range ':' chp_body '}'
209                       | '{' ';' ID ':' range ':' chp_body '}'
210 chp_nondet_guarded_command_list: chp_nondet_guarded_command_list ':' chp_guarded_command
211                                | chp_guarded_command ':' chp_guarded_command
212 chp_matched_det_guarded_command_list: chp_unmatched_det_guarded_command_list THICKBAR chp_else_clause
213                                     | chp_unmatched_det_guarded_command_list
```

```
214 chp_unmatched_det_guarded_command_list: chp_unmatched_det_guarded_command_list THICKBAR chp_guarded_command
215                                       | chp_guarded_command
216 chp_guarded_command: chp_guard_expr RARROW chp_body_or_skip
217 chp_guard_expr: chp_logical_or_expr
218 chp_unary_bool_expr: chp_simple_bool_expr
219                     | chp_not_expr
220                     | '(' chp_logical_or_expr ')'
221                     | chp_probe_expr
222                     | function_call_expr
223 chp_probe_expr: '#' member_index_expr
224 chp_simple_bool_expr: member_index_expr
225                      | BOOL_TRUE
226                      | BOOL_FALSE
227 chp_unary_expr: '-' chp_unary_expr
228               | chp_unary_bool_expr
229               | loop_expr
230               | INT
231               | FLOAT
232 chp_mult_expr: chp_unary_expr
233              | chp_mult_expr muldiv_op chp_unary_expr
234 chp_add_expr: chp_mult_expr
235             | chp_add_expr_only
236 chp_add_expr_only: chp_add_expr '+' chp_mult_expr
237                  | chp_add_expr '-' chp_mult_expr
238 chp_paren_add_expr: '(' chp_add_expr_only ')'
239                   | chp_mult_expr
240 chp_shift_expr: chp_paren_add_expr
241               | chp_shift_expr EXTRACT chp_add_expr
242               | chp_shift_expr INSERT chp_add_expr
243 chp_relational_expr: chp_shift_expr relational_op chp_shift_expr
244                    | chp_unary_bool_expr
245 chp_bitwise_and_expr: chp_relational_expr
246                     | chp_bitwise_and_expr '&' chp_relational_expr
247 chp_bitwise_xor_expr: chp_bitwise_and_expr
248                     | chp_bitwise_xor_expr '^' chp_bitwise_and_expr
249 chp_bitwise_or_expr: chp_bitwise_xor_expr
250                    | chp_bitwise_or_expr '|' chp_bitwise_xor_expr
251 chp_logical_and_expr: chp_bitwise_or_expr
252                     | chp_logical_and_expr LOGICAL_AND chp_bitwise_or_expr
253 chp_logical_or_expr: chp_logical_and_expr
254                    | chp_logical_or_expr LOGICAL_OR chp_logical_and_expr
255 chp_not_expr: '~' chp_unary_bool_expr
256 chp_else_clause: ELSE RARROW chp_body_or_skip
257 chp_binary_assignment: member_index_expr ASSIGN expr
258 chp_bool_assignment: member_index_expr '+'
259                    | member_index_expr '-'
260 chp_concurrent_item: chp_body_item
261                    | chp_sequence_group
262 chp_concurrent_group: chp_concurrent_group ',' chp_concurrent_item
263                     | chp_concurrent_item
264 chp_send: member_index_expr '!' connection_actuals_list
265         | member_index_expr '!'
266 chp_recv: member_index_expr '?' member_index_expr_list_in_parens_optional
267 chp_peek: member_index_expr '#' member_index_expr_list_in_parens
268 hse_body_optional: hse_body
269                  | /* empty */
270 hse_body: full_hse_body_item_list
271 full_hse_body_item_list: full_hse_body_item_list ';' full_hse_body_item
```

```
272                       | full_hse_body_item
273 full_hse_body_item: hse_body_item
274 hse_body_item: hse_loop
275              | hse_do_until
276              | hse_wait
277              | hse_selection
278              | hse_assignment
279              | SKIP
280 hse_loop: BEGINLOOP hse_body ']'
281 hse_do_until: BEGINLOOP hse_matched_det_guarded_command_list ']'
282 hse_wait: '[' expr ']'
283 hse_selection: '[' hse_matched_det_guarded_command_list ']'
284              | '[' hse_nondet_guarded_command_list ']'
285 hse_guarded_command: expr RARROW hse_body
286 hse_else_clause: ELSE RARROW hse_body
287 hse_nondet_guarded_command_list: hse_nondet_guarded_command_list ':' hse_guarded_command
288                               | hse_guarded_command ':' hse_guarded_command
289 hse_matched_det_guarded_command_list: hse_unmatched_det_guarded_command_list THICKBAR hse_else_clause
290                                     | hse_unmatched_det_guarded_command_list
291 hse_unmatched_det_guarded_command_list: hse_unmatched_det_guarded_command_list THICKBAR hse_guarded_command
292                                       | hse_guarded_command
293 hse_assignment: unary_assignment
294 prs_body_optional: prs_body
295                  | /* empty */
296 prs_body: prs_body prs_body_item
297         | prs_body_item
298 prs_body_item: single_prs
299              | prs_loop
300              | prs_conditional
301              | prs_macro
302              | TREE_LANG optional_template_arguments_in_angles '{' prs_body_optional '}'
303              | SUBCKT_LANG optional_template_arguments_in_angles '{' prs_body_optional '}'
304 prs_macro: prs_literal mandatory_member_index_expr_list_in_parens
305 prs_loop: '(' ':' ID ':' range ':' prs_body ')'
306 prs_conditional: '[' prs_guarded_list ']'
307 prs_guarded_list: prs_guarded_list_unmatched THICKBAR prs_else_clause
308                 | prs_guarded_list_unmatched
309 prs_guarded_list_unmatched: prs_guarded_list_unmatched THICKBAR prs_guarded_body
310                           | prs_guarded_body
311 prs_guarded_body: expr RARROW prs_body_optional
312 prs_else_clause: ELSE RARROW prs_body
313 single_prs: prs_rule_attribute_list_in_brackets prs_expr prs_arrow prs_literal_base dir
314           | prs_expr prs_arrow prs_literal_base dir
315 prs_rule_attribute_list_in_brackets: '[' prs_rule_attribute_list ']'
316 prs_rule_attribute_list: prs_rule_attribute_list ';' prs_rule_attribute
317                        | prs_rule_attribute
318 prs_rule_attribute: ID '=' expr_list
319 prs_arrow: RARROW
320          | IMPLIES
321          | HASH_ARROW
322 dir: '+'
323    | '-'
324 prs_expr: prs_or
325 prs_paren_expr: '(' prs_expr ')'
326 prs_literal_base: relative_member_index_expr
327                 | '@' ID optional_dense_range_list
328 prs_literal: prs_literal_base expr_list_in_angles_optional
329 prs_unary_expr: prs_literal
```

```
330               | prs_paren_expr
331               | prs_and_loop
332               | prs_or_loop
333 prs_not: '~' prs_unary_expr
334        | prs_unary_expr
335 prs_and: prs_and '&' prs_operator_attribute_optional prs_not
336        | prs_not
337 prs_or: prs_or '|' prs_and
338       | prs_and
339 prs_operator_attribute: '{' dir prs_expr '}'
340 prs_operator_attribute_optional: prs_operator_attribute
341                                | /* empty */
342 prs_and_loop: '(' '&' ':' ID ':' range ':' prs_expr ')'
343 prs_or_loop: '(' '|' ':' ID ':' range ':' prs_expr ')'
344 spec_body_optional: spec_body
345                   | /* empty */
346 spec_body: spec_body spec_item
347          | spec_item
348 spec_item: spec_directive
349 spec_directive: ID expr_list_in_angles_optional grouped_reference_list_in_parens
350 grouped_reference_list_in_parens: '(' grouped_reference_list ')'
351 grouped_reference_list: grouped_reference_list ',' grouped_reference
352                       | grouped_reference
353 grouped_reference: '{' mandatory_member_index_expr_list '}'
354                  | member_index_expr
355 paren_expr: '(' expr ')'
356 literal: INT
357        | FLOAT
358        | string
359        | BOOL_TRUE
360        | BOOL_FALSE
361 string: string STRING
362       | STRING
363 id_expr: generic_id
364 generic_id: relative_id
365           | absolute_id
366 absolute_id: SCOPE relative_id
367 relative_id: qualified_id
368            | ID
369 qualified_id: qualified_id SCOPE ID
370             | ID SCOPE ID
371 mandatory_member_index_expr_list_in_parens: '(' mandatory_member_index_expr_list ')'
372 mandatory_member_index_expr_list: mandatory_member_index_expr_list ',' member_index_expr
373                                 | member_index_expr
374 member_index_expr_list: member_index_expr_list ',' optional_member_index_expr
375                       | optional_member_index_expr
376 optional_member_index_expr: member_index_expr
377                           | /* empty */
378 member_index_expr: id_expr
379                  | index_expr
380                  | member_expr
381 relative_member_index_expr: ID
382                           | local_index_expr
383                           | local_member_expr
384 local_index_expr: local_member_expr sparse_range_list
385                 | ID sparse_range_list
389 index_expr: member_expr sparse_range_list
390           | id_expr sparse_range_list
```

```
394 simple_expr: member_index_expr
395            | literal
396 unary_expr: simple_expr
397           | function_call_expr
398           | paren_expr
399           | loop_expr
400           | '-' unary_expr
401           | '!' unary_expr
402           | '~' unary_expr
403 function_call_expr: member_index_expr connection_actuals_list
404 multiplicative_expr: unary_expr
405                    | multiplicative_expr muldiv_op unary_expr
406 muldiv_op: '*'
407          | '/'
408          | '%'
409 additive_expr: multiplicative_expr
410              | additive_expr '+' multiplicative_expr
411              | additive_expr '-' multiplicative_expr
412 shift_expr: additive_expr
413           | shift_expr EXTRACT additive_expr
414           | shift_expr INSERT additive_expr
415 relational_equality_expr: shift_expr
416                         | '(' relational_equality_expr '<' shift_expr ')'
417                         | '(' relational_equality_expr '>' shift_expr ')'
418                         | relational_equality_expr LE shift_expr
419                         | relational_equality_expr GE shift_expr
420                         | relational_equality_expr EQUAL shift_expr
421                         | relational_equality_expr NOTEQUAL shift_expr
422                         | '(' relational_equality_expr '=' shift_expr ')'
423 relational_op: '<'
424              | '>'
425              | LE
426              | GE
427              | EQUAL
428              | NOTEQUAL
429 and_expr: relational_equality_expr
430         | and_expr '&' relational_equality_expr
431 exclusive_or_expr: and_expr
432                  | exclusive_or_expr '^' and_expr
433 inclusive_or_expr: exclusive_or_expr
434                  | inclusive_or_expr '|' exclusive_or_expr
435 logical_and_expr: inclusive_or_expr
436                 | logical_and_expr LOGICAL_AND inclusive_or_expr
437 logical_or_expr: logical_and_expr
438                | logical_or_expr LOGICAL_OR logical_and_expr
439 unary_assignment: member_index_expr PLUSPLUS
440                 | member_index_expr MINUSMINUS
441 loop_expr: '(' loop_assoc_op ':' ID ':' range ':' expr ')'
442 loop_assoc_op: '+'
443              | '*'
444              | '&'
445              | '|'
446              | '^'
447              | LOGICAL_AND
448              | LOGICAL_OR
449 expr: logical_or_expr
450 strict_relaxed_template_arguments: complex_expr_optional_list_in_angles optional_template_arguments_in_angles
451                                  | /* empty */
```

```
optional_template_arguments_in_angles: complex_expr_optional_list_in_angles
                                     | /* empty */
expr_list_in_angles_optional: expr_list_in_angles
                            | /* empty */
expr_list_in_angles: '<' expr_list '>'
complex_expr_optional_list_in_angles: '<' complex_expr_optional_list '>'
complex_expr_optional_list: complex_expr_optional_list ',' optional_complex_expr
                          | optional_complex_expr
optional_complex_expr: array_concatenation
                     | /* empty */
member_index_expr_list_in_parens_optional: member_index_expr_list_in_parens
                                         | /* empty */
member_index_expr_list_in_parens: '(' member_index_expr_list ')'
expr_list_in_parens: '(' expr_list ')'
expr_list: expr_list ',' expr
         | expr
range: expr RANGE expr
     | expr
optional_dense_range_list: dense_range_list
                         | /* empty */
dense_range_list: dense_range_list bracketed_dense_range
                | bracketed_dense_range
sparse_range_list: sparse_range_list bracketed_sparse_range
                 | bracketed_sparse_range
bracketed_dense_range: '[' expr ']'
bracketed_sparse_range: '[' range ']'
complex_aggregate_reference: array_concatenation
array_concatenation: array_concatenation '#' complex_expr_term
                   | complex_expr_term
complex_expr_term: array_construction
                 | expr
array_construction: '{' mandatory_complex_aggregate_reference_list '}'
optional_complex_aggregate_reference: complex_aggregate_reference
                                    | /* empty */
mandatory_complex_aggregate_reference_list: mandatory_complex_aggregate_reference_list ',' complex_aggregate_reference
                                          | complex_aggregate_reference
complex_aggregate_reference_list: complex_aggregate_reference_list ',' optional_complex_aggregate_reference
                                | optional_complex_aggregate_reference
```

Function Index

Concept Index

L

M

N

O

P

R

S

T

U

V

W