The Tecton Concept Description Language

David R. Musser
Rensselaer Polytechnic Institute
Troy, New York 12180*

September 23, 1998

Abstract

This paper presents the syntax and semantics of a small language for describing and using abstract concepts in formal software development and hardware design. The language provides *definition, abbreviation, extension*, and *lemma* constructs, which have general mathematical descriptive power, plus a computation-specific *realization* construct. The semantics, which is denotational, includes specification of the requirements ("legality conditions") that must be met when using each construct. The syntax and semantics are such that a corresponding proof theory requires only first order and inductive proof methods, rather than general higher order techniques as required in some frameworks. The language and some of the main proof issues are illustrated with an extended example of a behavioral and structural description of a carry-lookahead adder circuit, with the circuit realization given in terms of a generic parallel-prefix circuit. (This is a major revision of Sections 2 and 3 of [16].)

*This work was performed while the author was on sabbatical at the Wilhelm-Schickard-Institut für Informatik, Universität Tübingen.

Contents

1 Introduction

[[Temporarily there is no introduction. I plan to add an introduction that updates the one in [16].]]

[[For those who have read [16], the primary differences are:

1. The definition of the semantics of concept instances is significantly different from that given in [16]. The old definition appears to have been logically sound but too restrictive, and some of the examples given in [16] failed to satisfy the implied restrictions. The case in which it was too restrictive is `B [with ... x as y ...]` where both `x` and `y` are already available. The given semantics simply exchanged the meanings of `x` and `y` and left both available, but some of the examples assumed that `y` was no longer available and could therefore be reintroduced with an introduces or uses clause in a derived concept description. The new semantic definition only admits algebras in which `x` and `y` already have the same meaning and makes `y` unavailable, thus supporting the examples. (Some other minor adjustments to the examples do have to be made.)

2. A change in terminology is that "sentence" is now used in place of "statement." This change will make it easier to integrate Tecton with languages (such as SuchThat) that express algorithms in terms of statements.

3. The notion of restrictions of algebras has been extended to restrictions of concepts, and the latter notion is used in defining the semantics of Tecton extension sentences. The new semantics is equivalent to the old.

4. Expression abbreviations have been removed from the abbreviation sentence syntax and semantics. They can be done as extensions.

5. The syntax of the lemma form A `is` B is now A `implies` B, and the syntax of the lemma form A `=` B is now A `is` B. These changes make some lemma examples read more naturally.

6. The range of a function may now be either a single sort or a list of sorts. Actually, in this draft this modification is not yet complete; the definition of sort-correctness has yet to be changed.

7. The major parts of the language definition are rearranged in an order that I hope will make it easier to comprehend. The inductive definitions of sort-correctness and expression valuations have been moved to appendices. They are not difficult but they are detailed, and exact knowledge of the details is not necessary to understanding the rest of the language definition. The hardest part is probably the semantics of concept instances which now appears later than it previously did, and is somewhat better motivated in this position and by some additional discussion. However, this section definitely still needs more work to find better notation and clearer definitions. Even more important, more formal justification of the generality of the definitions should be given via lemmas like the ones about concept restrictions given in Section 2.2.

8. Examples are now presented as "parts" which can be assembled in different ways for testing Tecton language processors or as the basis for development

of further examples.[1] The technique for such presentation is based on *literate programming* [23], but here we are using it for Tecton concept descriptions, not programs. The tool we use is called NUWEB [4], or actually a minor revision of it.[2]

Another improvement still needed is better examples. I may combine this presentation with the STL container and iterator concept examples rather than continuing to use the carry-lookahead adder example.]]

2 Concepts and concept descriptions

Definition of *concept*. In Tecton terminology, a "concept" is a set of many sorted algebras. We assume the existence of two disjoint sets of identifiers (strings), called *sorts* and *function identifiers*. An *indexed family of sets* is a family of pairs (s, S), where s is a sort and S is a set. An *indexed family of functions* is a family $\mathcal{F}$ of pairs (σ, f), where f is a function and σ is a *function description* consisting of a function identifier and an *arity*, which consists of two lists of sorts, the first called *domain sorts* and the second called *range sorts*. Then a *(many-sorted) algebra* is

- an indexed family $\mathcal{S}$ of sets such that $\mathcal{S}$ is a mapping (i.e., if $(s, S_1) \in \mathcal{S}$ and $(s, S_2) \in \mathcal{S}$ then $S_1 = S_2$).
- an indexed family $\mathcal{F}$ of functions, such that $\mathcal{F}$ is a mapping and for each pair (σ, f) in $\mathcal{F}$, where σ has domain sorts $s_1, \ldots, s_n$ and range sorts $r_1, \ldots, r_m$, f is a function with domain $\mathcal{S}(s_1) \times \cdots \times \mathcal{S}(s_n)$ and range $\mathcal{S}(r_1) \times \cdots \times \mathcal{S}(r_m)$.

Finally, we define a *concept* as a set of many-sorted algebras that all satisfy a common set of requirements.

Overview of concept descriptions. The Tecton language permits statement of the requirements on a concept using a series of sentences called *concept descriptions*. These sentences take one of five forms:[3]

Tecton sentence ::= *Tecton definition* | *Tecton abbreviation*
| *Tecton lemma* | *Tecton extension* | *Tecton realization*

[1] There is no connection between this use of the term "part" and the part terminology used within the language definition itself.

[2] A version of Nuweb previously modified by Ramsdell and Mengel was revised mainly to make small changes in terminology in the LaTeX file the tool produces: "part" is used in place of "scrap" and "definition" in place of "macro." This version, called Nuweb 0.91, is available from *http://www.cs.rpi.edu/~musser/gp/*. The new version does not differ from previous versions in the way it produces code files from Nuweb source files.

[3] To give the context-free part of the syntax, we use BNF with the following extensions:

- clauses enclosed in { } are optional;
- *construct*-LIST means a list of *construct* separated by commas;
- *construct*-LIST`[;]` means a list of *construct* separated by semi-colons;
- *construct*-LIST`[ ]` means a list of *construct* separated by blanks.

BNF rules are in some cases augmented by context-sensitive syntax rules, stated in English.

Each of the sentence forms introduces or refers to a concept name, called the *subject* of the sentence. A definition, concept abbreviation, or realization defines its subject: it associates the concept name with a meaning, which is a concept—i.e, a set of algebras. An extension or expression abbreviation adds information about new identifiers to a concept while preserving the meaning it gives to other identifiers, while lemmas assert that certain properties follow from the definition of and extensions to the concept.

A concept description typically builds upon or modifies other concept descriptions, or both. It may name other concepts that it *refines* or *uses*, the distinction being based on whether it changes (refines) the meaning of identifiers in the other concepts, or merely uses the identifiers without change. Although identifier names are part of the semantics of concepts (they appear in the domains of the set and function description mappings of the algebras in concepts), new names can be substituted at any time. Such substitutions are introduced in "concept instances" of the form C [with x_1 as $y_1, \ldots, x_n$ as y_n], where C is the subject of a previously given sentence, $y_1, \ldots, y_n$ are identifiers already "available" in the definition of C, and each x_i is an identifier being substituted for y_i. The definition of availability of identifiers is given in the next section as part of the discussion of sort correctness.

2.1 Tecton definitions

A Tecton definition is a concept description of the form:

```
Tecton definition ::=
   Definition: concept-identifier { refines-clause } { ; }
     { uses-clause } { ; } { introduces-clause } { ; }
     { generates-clause } { ; } { requires-clause } .
```

At least one of the refines, uses, introduces, generates, or requires clauses must be present. A definition introduces a new description of a set of algebras if neither a refines clause nor a uses clause is present, or (the usual case) it derives a description from previously given descriptions.

The introduces clause introduces sorts and function descriptions.

```
introduces-clause ::=
  introduces { sort-or-subsort -LIST } { , }
    { function-declaration -LIST }
sort-or-subsort ::= sort | subsort-declaration
subsort-declaration ::= sort < sort
sort ::= identifier
function-declaration ::=
   function-identifier { ( sort -LIST ) }
      -> sort-or-sort-list { protected-or-private }
|  function-declaration-with-variables = expression
               { protected-or-private }
protected-or-private ::= (protected) | (private)
function-identifier ::= identifier
sort-or-sort-list ::= sort | ( sort -LIST )
function-declaration-with-variables ::=
    function-identifier { ( variable-and-sort -LIST ) }
```

variable-and-sort ::= *variable-identifier* : *sort*
variable-identifier ::= *identifier*

(Function-identifiers include symbols like `+`, `*`, etc. Such lexical rules, rules about operator precedence, and several context-sensitive syntax rules for expressions are not spelled out here.)

⟨Example: Set and Binary-op definitions 4⟩ ≡

```
Definition: Set
  uses Boolean;
  introduces domain.

Definition: Binary-op
  refines Set;
  introduces +(domain, domain) -> domain.
```

◇

Macro referenced in scraps 18ab.

Optional infix notation. Any function identifier declared with two arguments can be used in either prefix or infix notation; e.g., `+(x, y)` or `x + y`.

Function declarations. A function declaration includes a *function description*, as defined earlier. It may also include a *protected* or *private* qualification, which affects the scope of the declared function in other concepts, as described later in the definition of availability.

Subsort declarations. In a subsort declaration s_1 `<` s_2, the set indexed by s_1 is declared to be a subset of the set indexed by s_2.[4]

Syntactic and semantic concepts. The meaning associated with a definition is a concept, a set of algebras. Actually, we associate two sets of algebras with a definition, called the *syntactic concept* and the *semantic concept* (when we use just the word "concept" we mean the semantic concept). If A is a concept name we write syn A for the syntactic concept of A and sem A for the semantic concept of A. We always have $\mathrm{sem}\, A \subseteq \mathrm{syn}\, A$.

Simple concept definitions. For the moment, we only consider definitions that have no refines or uses clause; these are called *simple concept definitions*.

The syntactic concept denoted by a simple concept definition D is defined as follows: an algebra with indexed sets $\mathcal{S}$ and indexed functions $\mathcal{F}$ belongs to the syntactic concept if and only if the sorts introduced in D belong to $\mathrm{dom}\,\mathcal{S}$ (the domain of $\mathcal{S}$), the subsort declarations are satisfied (i.e., $\mathcal{S}(s_1) \subseteq \mathcal{S}(s_2)$ when s_1 `<` s_2), and the function descriptions introduced in D belong to $\mathrm{dom}\,\mathcal{F}$.

Note: There may be other sorts in $\mathrm{dom}\,\mathcal{S}$ besides those introduced in D, and, similarly there may be other function descriptions in $\mathrm{dom}\,\mathcal{F}$ besides those introduced in D. (Either or both of $\mathrm{dom}\,\mathcal{S}$ and $\mathrm{dom}\,\mathcal{F}$ could in fact be infinite).

[4] One use of subsorts is to allow specification of functions as total functions on the subsort which otherwise would be partial functions on the larger sort. (See [12], which also discusses other uses.)

This is a significant difference from the usual approach in which each algebra associated with a syntactic description, called a signature, has exactly the sets and functions named in the signature.

Requires clauses If A is the concept identifier of a simple concept definition with no requires or generates clause, then $\text{sem}\, A = \text{syn}\, A$. A requires clause or generates clause requires the sets and functions of an algebra to have certain properties. We consider requires clauses first.

```
requires-clause ::= requires quantifiers property-LIST
quantifiers ::= ( for { some } variable-declaration-LIST[;] )
variable-declaration ::= variable-identifier-LIST : sort
property ::= expression
expression ::=
    variable-identifier { : sort }
  | function-identifier { ( expression-LIST ) }
  | expression function-identifier expression
  | quantifiers expression
  | if expression then expression else expression
  | expression where variable-identifier = expression
  | ( expression )
```

⟨Example: Associative concept 5⟩ ≡

```
Definition: Associative
  introduces domain;
     +(domain, domain) -> domain;
  requires (for x, y, z: domain)
    x + (y + z) = (x + y) + z.
```

◇

Macro referenced in scrap 27.

Another definition of Associative that uses refinement is given later.

Additional rules on quantifiers. In the quantifiers of a requires clause, each sort must be one that is introduced in the current sentence or in one of the concepts instantiated in the sentence. No variable may appear more than once in the quantifiers. In $v_1, \ldots, v_n : s$, each variable v_i is said to be declared to be of sort s.

Sort correctness. In each expression in the property list, each variable must be declared in the quantifiers and each sort and function identifier must be *available* in the sentence. We define availability inductively as follows:

Definition 1 *A sort or function identifier i is* available *in a sentence S if and only if one of the following conditions holds:*

1. *i is declared in S (i.e., appears as a sort or function identifier in the introduces clause of S);*

2. *i is available in a concept instance C [with x_1 as y_1, ..., x_n as y_n] in the refines clause of S (i.e., either i is available in the sentence U of which C*

is the subject and is not equal to any of $y_1, \ldots, y_n$*; or* $i = x_j$ *for some* $j, 1 \leq j \leq n$*, such that* y_j *is available in* U*) and is not marked private;*

3. *i is available in a concept instance C [with x_1 as $y_1, \ldots, x_n$ as y_n] in the uses clause of S and is not marked private or protected;*
4. *i is available in a concept description C and S is an extension of C; or*
5. *i is available in a concept description C and S is a realization of C.*

In the last two cases, the status of i as private, protected, or neither remains the same in the extension or realization.

A function identifier may appear more than once in the sentence and instantiated concepts, but not with identical arities. If it does appear with different arities, it is said to be *overloaded*. Using these declarations, the expression must be *sort-correct*, as defined in Appendix A. If the expression is sort-correct using more than one set of available declarations, the expression, and the sentence containing it, are considered illegal.

The quantifiers clause that appears at the beginning of the requires clause is treated as though it were included in every property in the requires clause.

Requires clause characterization of subsorts. For every subsort declaration, the requires clause must include a statement of a property characterizing membership in a subsort, which can be done using the : operator.

⟨Example: Characterization of nonzero natural numbers 6⟩ ≡

```
Definition: Natural
  introduces naturals,
     nonzero < naturals,
     0 -> naturals,
     succ(naturals) -> nonzero,
     ...
  requires (for n: naturals)
    n: nonzero = not(n = 0),
    ...
```

◇

Macro referenced in scrap 27.

Assigning truth values. Roughly speaking, the semantics of a simple concept definition D with subject C is that an algebra $(\mathcal{S}, \mathcal{F})$ belongs to the concept $\mathrm{sem}\, C$ if and only if belongs to $\mathrm{syn}\, C$ and under the sort and function description mappings $\mathcal{S}$ and $\mathcal{F}$, each property p in the requires clause of D is a true statement. To make this precise, we must describe exactly how truth values are assigned to properties. Let D be a concept description with subject C and let $a = (\mathcal{S}, \mathcal{F})$ be an algebra in $\mathrm{syn}\, C$. Define an $\mathcal{S}$ *valuation* ν to be a mapping of declared variables to values, in which the value $\nu(v)$ is from the set $\mathcal{S}(s)$ when the declared sort of v is s. An inductive definition of the valuation mapping ν is given in Appendix B, based on the inductive definition of the syntax of properties (expressions). We write $a^{\nu}(p)$ for the value that ν assigns to p. We return to the use of this valuation in defining the semantics of concept descriptions following the definition of *generates clauses*.

Generates clauses.

generates-clause ::=
  generates *sort* { freely } using *function-descriptor*-LIST
function-descriptor ::=
  function-identifier { { (*sort*-LIST) } -> *sort* }

A generates clause restricts algebras in the concept to those in which all elements of the set corresponding to the sort named in the clause are finitely generated by the functions named in the clause. This property serves as a basis for developing induction schemata for proving properties by induction for all values associated with the named sort.

⟨Example: Generates clause in definition natural number concept 7⟩ ≡

```
Definition: Natural
  introduces naturals,
  ...
  generates naturals freely using 0, succ;
  ...
```

◇

Macro referenced in scrap 27.

The complete definition of the Natural concept is given in Section 3.1. From the generates clause we obtain the usual simple induction schema for natural numbers,

$$\frac{P(0) \qquad (\text{for } n : \text{naturals})\ P(n) \text{ implies } P(\text{succ}(n))}{(\text{for } n : \text{naturals})\ P(n)}$$

To state precisely the meaning of the generates clause, let $\psi = \{s_1, \ldots, s_k\}$ be all of the declared subsorts of s, including s itself. In

generates s using σ_1, ..., σ_n

the function descriptors must be from among the available function declarations and must have range sort in ψ. For an algebra $(\mathcal{S}, \mathcal{F})$ to belong to the concept it must have $\mathcal{S}(s) = U$ where every value in U is denotable by a sort-correct expression in which

- every function identifier with range sort in ψ refers to a member of $\{\sigma_1, \ldots, \sigma_n\}$,
- there is no variable whose sort is in ψ.

If the word freely is included in the clause, there is an additional requirement that any two such expressions must denote distinct values of U if they cannot be made identical by replacement of subexpressions whose sort is not in ψ by variables. This results in additional properties: For distinct $\sigma_i = f_i(d_{j1}, \ldots, d_{jn}) \to r_i$ and $\sigma_j = f_j(e_{j1}, \ldots, e_{jm}) \to r_j$,

$$f_i(x_1, \ldots, x_n) \neq f_j(y_1, \ldots, y_m)$$

and for every $\sigma_i = f_i(d_{j1}, \ldots, d_{jn}) \to r_i$,

$$f_i(x_1, \ldots, x_n) = f_i(y_1, \ldots, y_n) \text{ implies } x_1 = y_1 \text{ and } x_2 = y_2 \text{ and } \ldots \text{ and } x_n = y_n.$$

Thus for the above natural numbers example, we obtain the properties

$$\mathrm{succ}(n) \neq 0$$

and

$$\mathrm{succ}(n) = \mathrm{succ}(m) \text{ implies } n = m.$$

Without the word `freely` in the `generates` clause the concept would include not only the natural number algebra but also the residue class rings Z_m, for every $m > 0$.

Semantics of simple concept definitions. We are now prepared to give the precise semantics of a simple concept definition with subject C. An algebra a belongs to $\mathrm{sem}\, C$ if and only if:

1. it belongs to $\mathrm{syn}\, C$;
2. for each property p in the requires clause, $a^{\nu}(p) = \mathbf{true}$, for an arbitrary[5] $\mathcal{S}$ valuation ν; and
3. the `generates` clause, if any, is satisfied.

2.2 Inheritance from previous concept descriptions

The refines and uses clauses of a concept definition allow reuse of previously described concepts, and they also serve to identify components of a definition that can be treated as parameters for concept replacements (which are described in the next subsection).

```
refines-clause ::= refines superior-LIST
superior :: = concept-instance
uses-clause ::= uses part-LIST
part ::= concept-instance
```

Thus superiors and parts are concept instances (concept names modified by replacement lists), which are described in full generality in the next subsection. For the moment we can think of them as just being concept names.

[5]This valuation will not actually be used since there are no free variables allowed—every variable must be declared in the quantifiers.

⟨Example: Tecton definitions with refines and uses clauses 9⟩ ≡

```
Definition: Identity
  refines Binary-op;
  introduces 0 -> domain;
  requires (for x: domain) x + 0 = x.

Definition: Monoid
  refines Associative, Identity.

Definition: Abelian-monoid
  refines Monoid, Commutative.

Definition: Left-distributive
  refines Binary-op, Binary-op [with * as +];
  requires (for x, y, z: domain)
     x * (y + z) = x * y + x * z.
```
◇

Macro referenced in scrap 27.

To give the semantics of a definition D of concept C with refines or uses clauses, let $C_1, \ldots, C_n$ be all of the parts and superiors that appear in D. An algebra is in $\operatorname{syn} C$ if and only if it is in $\operatorname{syn} C_1 \cap \ldots \cap \operatorname{syn} C_n$, and the sorts introduced in D belong to $\operatorname{dom} \mathcal{S}$ and the function descriptions introduced in D belong to $\operatorname{dom} \mathcal{F}$. An algebra a is in $\operatorname{sem} C$ if and only if $a \in \operatorname{syn} C \cap \operatorname{sem} C_1 \cap \ldots \cap \operatorname{sem} C_n$ and each property in the requires list of D is a true statement about the sets and functions of a.

Restriction requirement on parts. There is an important semantic requirement on parts, one that does not apply to superiors. Intuitively, when B has A as a part, we want to require that the meaning of the symbols of A be unchanged by the description of B. This *restriction requirement* allows us to treat A as a parameter for which any concept C contained in $\operatorname{sem} A$ can be substituted, without the possibility of introducing an inconsistency between the properties of C and those imposed within B. In order to state the restriction requirement formally, we make the following definitions.

Definition. For any concept description C, define avs C to be the set of available sorts of C, and avf C to be the set of available function descriptors of C.

Definition. Let C and B be concept descriptions such that $\operatorname{sem} C \subseteq \operatorname{sem} B$, and let $a = (\mathcal{S}, \mathcal{F}) \in \operatorname{sem} C$. The *restriction of a to B* (notation: $a|_B$) is the subalgebra of a whose sorts and function descriptors are restricted exactly to those available in B; i.e., $(\mathcal{S}', \mathcal{F}')$ such that $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{F}' \subseteq \mathcal{F}$, $\operatorname{dom} \mathcal{S}' = \operatorname{avs} B$, and $\operatorname{dom} \mathcal{F}' = \operatorname{avf} B$. Note that $a|_B \in \operatorname{sem} B$.

Definition. Let C and B be concept descriptions such that $\operatorname{sem} C \subseteq \operatorname{sem} B$. The *restriction of C to B* (notation: $C|_B$) is the concept $\{a|_B : a \in \operatorname{sem} C\}$. Note that $C|_B \subseteq B|_B \subseteq \operatorname{sem} B$.

Formal statement of the restriction requirement. Let B and C be concept descriptions such that C uses B. The restriction requirement on C is

$$C|_B \quad = \quad B|_B$$

From the definition of C as an intersection of B with other sets, we have sem $C \subseteq$ sem B, and from this relation and the definition of availability we also have that avs $C \supseteq$ avs B and avf $C \supseteq$ avf B.

⟨Example: Set uses Boolean 10⟩ ≡

```
Definition: Set
    uses Boolean;
    introduces domain,
```

◇

Macro referenced in scrap 27.

Here the restriction requirement is that

$$\text{Set}|_{\text{Boolean}} = \text{Boolean}|_{\text{Boolean}}$$

where the concept on the left is formed by restricting each algebra in sem Set to just those indexed sets and functions required by Boolean, and the concept on the right is formed by restricting each algebra in sem Boolean to just those indexed sets and functions required by Boolean. Equality between these two concepts holds trivially in this example because there is no requires clause in the description of Set.

There is no such restriction requirement on superiors; in fact, when we include a concept instance B in the refines clause of C we usually intend to modify the semantics associated with B, thereby eliminating some algebras from the semantic concept of C.

If A, B, and C are concept descriptions such that B uses A and C uses B, we would like to treat A as a part of C. That is, we should have the same semantics as if we had written explicitly that C uses A, allowing A be treated as a parameter eligible for substitution in creating instances of C. This is in fact the case, as a consequence of the following lemmas.[6]

Lemma 1 *Let A, B, and C be concept descriptions such that sem $C \subseteq$ sem $B \subseteq$ sem A. Then*

$$(C|_B)|_A = C|_A$$

Proof. By the assumption sem $C \subseteq$ sem $B \subseteq$ sem A, each of the restrictions mentioned in the conclusion is well defined, and avs $A \subseteq$ avs B and avf $A \subseteq$ avf B. These relations justify the third equation in the following chain:

$$\begin{aligned}(C|_B)_A &= \{a|_A : a \in C|_B\} \\ &= \{(a|_B)_A : a \in C\} \\ &= \{a|_A : a \in C\} \\ &= C|_A\end{aligned}$$

□

[6]Christoph Schwarzweller formulated and proved these lemmas improving upon an earlier formulation by the author.

Lemma 2 *Let A, B, and C be concept descriptions such that* $sem C \subseteq sem B \subseteq sem A$, $B|_A = A|_A$, *and* $C|_B = B|_B$. *Then*

$$C|_A = A|_A.$$

Proof. By the $\mathrm{sem}\, C \subseteq \mathrm{sem}\, B \subseteq \mathrm{sem}\, A$ assumption and the previous lemma,

$$\begin{aligned} C|_A &= (C|_B)|_A \\ &= (B|_B)|_A \\ &= B|_A \\ &= A|_A \end{aligned}$$

□

2.3 Concept instances

Tecton definitions do not have explicit parameters; instead, one can treat many internal constituents of a definition as parameters subject to replacement. This gives the language user greater flexibility than if parameters had to be explicitly listed at the time of definition, for it is often difficult to anticipate which constituents it might be useful to substitute for in any subsequent use of the defined concept. At the same time, we want to prohibit replacements that could lead to syntactic or sort incorrectness, or to semantic inconsistency (empty concepts). These considerations motivate the following definitions.

A *concept instance* is a concept name optionally followed by a clause specifying replacements:

```
concept-instance ::=  concept-name { [ with  replacement-LIST ] }
concept-name ::=  concept-identifier { by  concept-identifier }
```

The concept name part of a concept instance is called the base of the concept instance. The form A `by` B is introduced in a realization sentence, as discussed in Section 2.7.

There are three kinds of replacements allowed:

```
replacement  ::=  sort as  sort
  |  function-descriptor as  function-descriptor
  |  concept-name as  concept-name
```

A function-descriptor is usually just a function-identifier, but the longer form can also be used, and must be used when it is necessary to distinguish among different instances of an overloaded function-identifier.

⟨Example: Concept instances 11⟩ ≡

```
Binary-op [with * as +]

Binary-op [with * as +(domain, domain) -> domain]

Indexed-circuit [with Boolean as State, bool as states,
                      true as high, false as low].
```

◇

Macro referenced in scrap 27.

Syntactic and sort-correctness requirements. In sort or function replacements y_i `as` x_i, a declaration of x_i must be available in the base concept description. If y_i and x_i are function descriptors and declarations of both are available, they must have the same arity.

Semantics. The semantics of a concept instance is defined in terms of relabellings of algebras. Let c be a string or sequence of strings and $u_1, \ldots, u_n, v_1, \ldots, v_n$ be strings. Then by $c[v_1/u_1, \ldots, v_n/u_n]$, called a *relabelling* of c, we denote the result of simultaneous replacement of u_1 by v_1, ..., u_n by v_n in c.

Case 1: Subsort and function replacements only. To give the semantics of a concept instance C with base B, first consider $B^* = B$ [with y_1 as $x_1, \ldots, y_n$ as x_n] where only the sort and variable replacements of C are applied, not concept replacements, and where the symbols $x_1, \ldots, x_n$ are available in B.

Let $a = (\mathcal{S}, \mathcal{F})$ be any algebra $\in \operatorname{sem} B$ such that for any y_i that is available in B, either $\mathcal{S}(x_i) = \mathcal{S}(y_i)$ if x_i is a sort or $\mathcal{F}(x_i) = \mathcal{F}(y_i)$ if x_i is a function descriptor. (If none of the y_i is available in B then a can be any algebra in $\operatorname{sem} B$.)

Define the algebra $a' = (\mathcal{S}', \mathcal{F}')$ by

$$\begin{aligned}\mathcal{S}' &= \{(s[y_1/x_1, \ldots, y_n/x_n], S) : (s, S) \in \mathcal{S}\} \\ \mathcal{F}' &= \{(\sigma[y_1/x_1, \ldots, y_n/x_n], f) : (\sigma, f) \in \mathcal{F}\}\end{aligned}$$

We call this algebra a' the *relabelling of a using $y_1/x_1, \ldots, y_n/x_n$*. Note the assumption of the previous paragraph is necessary for well-definedness of a'.

Now let a'' be any algebra $(\mathcal{S}'', \mathcal{F}'')$ such that $\mathcal{S}'' \supseteq \mathcal{S}'$ and $\mathcal{F}'' \supseteq \mathcal{F}'$.

Then $a'' \in \operatorname{sem} B^*$, and the only algebras in $\operatorname{sem} B^*$ are those obtainable in this way.

If there are no concept replacements in C, we simply define

$$\operatorname{sem} C = \operatorname{sem} B^*.$$

Case 2: The general case. More generally, let the concept replacements in C be B_1 `as` A_1, ..., B_m `as` A_m. Then we define

$$\operatorname{sem} C = \operatorname{sem} B^* \cap \operatorname{sem} B_1 \cap \cdots \cap \operatorname{sem} B_m.$$

Requirements for legality of concept replacements. In the case of concept name replacements B_i `as` A_i, both names must be defined and there are two further requirements. First, we must have

$$\operatorname{sem} B \subseteq \operatorname{sem} A_i. \tag{1}$$

Second, let $A_i^* = A_i$ [with y_1 as $x_1, \ldots, y_n$ as x_n]. If

$$B|_{A_i} = A_i|_{A_i}$$

then we require

$$\operatorname{sem} B_i \subseteq \operatorname{sem} A_i^*; \tag{2}$$

but if

$$B|_{A_i} \neq A_i|_{A_i}$$

then we require

$$\mathrm{sem}\, B_i = \mathrm{sem}\, A_i^*. \tag{3}$$

A sufficient condition for (1) to hold is that A_i be a part or superior in the concept description named by B. If A_i is a part then the equality between restrictions will hold and only condition (2) is required, but if it is a superior, then we require condition (3).

In effect, the superiors and parts mentioned in the base concept description are formal parameters and can be replaced using other concept names as actual parameters. The concept named by an actual parameter must be contained in the concept of the corresponding formal parameter that is a part, or must be the same concept as that denoted by the formal parameter if the latter is a superior. Even more generally, concepts not explicitly named in the base concept description can also be used as parameters if the legality conditions can be established for them.

⟨Example: Concept replacement 13a⟩ ≡

```
Indexed-circuit [with Boolean as State, bool as states,
                true as high, false as low]
```

◇

Macro referenced in scrap 27.

It suffices for `State` to appear as a part in the concept description of `Indexed-circuit` and the semantic concept of `Boolean` to be contained in the semantic concept of `State [with bool as states, true as high, false as low]`.

2.4 Abbreviations

Tecton abbreviations allow introduction of a new concept name for a concept instance.

abbreviation `::= Abbreviation:` *concept-name* `is` *concept-instance* `.`

The form

`Abbreviation:` C `is` A

is called a concept abbreviation and is just a way of naming a concept instance A with a new concept name C. We define $\mathrm{syn}\, C = \mathrm{syn}\, A$ and $\mathrm{sem}\, C = \mathrm{sem}\, A$.

⟨Example: Concept abbreviations 13b⟩ ≡

```
Abbreviation: Semigroup is Associative.

Abbreviation: Range is Set [with codomain as domain].

Abbreviation: All
   is Generalized-reduction [with bool as codomain, p as f, and as +].

Abbreviation: Exists
   is Generalized-reduction [with bool as codomain, p as f, or as +].
```

◇

Macro referenced in scrap 27.

2.5 Lemmas

A Tecton lemma is a concept description with one of three forms

```
lemma  ::=
  Lemma:  concept-name {  generates-clause } { ; }
    {  requires-clause } .
| Lemma:  concept-instance is  concept-instance .
| Lemma:  concept-instance implies  concept-instance .
```

The first form of a lemma is used to state properties of a previously defined concept that are logical consequences of the properties stated in the definition of or extensions to that concept. At least one of the generates or requires clauses must be present.

⟨Example: Lemmas asserting additional properties 14⟩ ≡

```
Lemma: Group requires (for x: domain) 0 + x = x.

Lemma: Group requires (for x: domain) -(x) + x = 0.

Lemma: Sequence generates sequences freely using null, |-.
◇
```

Macro referenced in scrap 27.

The same syntactic requirements are placed on the quantifiers and properties as in a definition. Each property in the property list of a lemma must be a true statement about the named sets and functions of every algebra in the concept, and the `generates` clause, if present, must be satisfied.

There is, obviously, a proof obligation associated with a lemma, to show that the stated properties follow from the definition and extensions (if any) of the concept.

The subject of a lemma is the concept name it begins with. A lemma does not change the concept associated with its subject. (However, a Tecton implementation should provide queries about concept names which would list all lemmas whose subject was a given concept name.)

The second and third forms of a lemma state a relation between concept instances. The second form

`Lemma:` A `is` B.

asserts that A and B are the same concept; that is, $\mathrm{sem}\, A = \mathrm{sem}\, B$. The third form

`Lemma:` A `implies` B

asserts that $\mathrm{sem}\, A \subseteq \mathrm{sem}\, B$. In either of these two forms, the subject is the base of the first of the two concept instances.

⟨Example: Lemmas relating two concept instances 15a⟩ ≡
```
Lemma: Integer implies Total-order [with integers as domain].

Lemma: Second-order-sequence
   implies Reduction [with Sequence-monoid as Monoid].

Lemma: Natural implies Monoid [with naturals as domain].

Lemma: Natural implies Monoid [with naturals as domain, * as +, 1 as 0].

Lemma: Natural [with domain as naturals, + as *, 0 as 1] implies Monoid.
```
◇

Macro referenced in scrap 27.

2.6 Extensions

A Tecton extension is a concept description of the form

extension ::=
  Extension *concept-name* { *uses-clause* } { ; }
   { *introduces-clause* } { ; } { *requires-clause* } .

At least one of three clauses must be present. An extension is used to introduce new sorts and function identifiers and to state properties about them. An extension associates the new information with a previously defined concept name. In subsequent references to the concept name the sorts and functions introduced in the extension are available, as though they had actually been introduced in the sentence defining the name.

⟨Example: Extension of Sequence concept 15b⟩ ≡
```
Extension: Sequence
  introduces
    reverse(sequences) -> sequences;
  requires (for s: sequences; x: elements)
    reverse(null) = null,
    reverse(x -| s) = reverse(s) |- x.
```
◇

Macro referenced in scrap 27.

A semantic requirement on an extension is that the meaning of any identifiers already available in the concept is not changed (i.e., it is a *conservative extension*). This requirement can be formally stated as follows: given

`Extension:` C *clauses*

let D be a new concept identifier and construct

`Definition:` D `uses` C; *clauses*

(If *clauses* contains a uses clause it is combined with `uses` C.) If this is a legal definition, then after the extension, $\mathrm{syn}\, C = \mathrm{syn}\, D$ and $\mathrm{sem}\, C = \mathrm{sem}\, D$. Among the conditions for the definition to be legal is that $D|_C = C|_C$, which is what guarantees that the meaning of any sorts or function descriptors already available in the concept is not changed. If the constructed definition is not legal, neither is the extension.

Extensions thus have the same meaning as a restricted class of concept definitions, but without the need for a new concept name.

A useful form of extension is introduction of new functions that can be defined in terms of an expression involving old functions.

⟨Example: Extensions that introduce new functions 16a⟩ ≡

```
Extensions: Group introduces -(x: domain, y:domain) = x + -(y).

Extension: Partial-order
   introduces <(x: domain, y: domain) = (x <= y) and not(x = y).

Extension: Array
   introduces swap(a: arrays, i: naturals, j: naturals)
              = assign(assign(a, j, access(a, i)), i, access(a, j)).
```

◇

Macro referenced in scrap 27.

2.7 Realizations

Tecton realizations are concept descriptions of the form:

realization ::=
Realization: *concept-identifier* by *concept-identifier*
{ *refines-clause* } { ; } { *uses-clause* } { ; }
{ *introduces-clause* } { ; } { *requires-clause* } .

In

Realization: A by B ...

the concept-identifiers A and B must already have been the subject of previous sentences. The concept name A by B is the subject of this concept description. The concept C associated with A by B is the same as given by a definition with the same clauses, except that all sorts and function declarations available in A are also available (but the semantics of A is not automatically inherited, see below), and B is treated as a part (as though added to the uses clause).

⟨Example: Realization of the Stack concept by the Sequence concept 16b⟩ ≡

```
Realization: Stack by Sequence
     introduces
         rep(sequences) -> stacks (private),
     requires (for s, u: sequences; t: nonempty-sequences; e: elements)
         empty = rep(null),
         push(e, rep(s)) = rep(s |- x),
         top(rep(t)) = last(t),
         pop(rep(t)) = rep(butlast(t)),
         size(rep(s)) = length(s),
         (rep(s) = rep(u)) = (s = u).
```

◇

Macro referenced in scrap 27.

There is a substantial semantic requirement associated with a realization: the concept relationship that could be stated by a lemma of the form

$$A \texttt{ by } B \quad \texttt{implies} \quad A.$$

The corresponding proof obligation is to prove this lemma, which involves proof of each of the properties in the theory of A from the new information in the realization, including the properties of equality for the = function. For assistance with this proof obligation, one may obtain from the induction principle for A a corresponding induction principle that applies to terms built with a representation function such as `rep` in the above example, as discussed for example in [14, Section 4].

3 Simple examples of concept descriptions

3.1 Boolean and Natural concepts

The Boolean concept is actually predefined by the semantics, but it would look as follows as a Tecton definition.

⟨Example: Boolean concept 17a⟩ ≡

```
Definition: Boolean
  introduces bool,
      true -> bool,
      false -> bool;
  generates bool freely using true, false.
```

◇

Macro referenced in scrap 27.

From the generates clause we conclude that in any algebra in the concept, the elements of the set $\mathcal{S}$(`bool`) must be denotable by at most two expressions, `true` and `false`. By the `freely` keyword, `true` and `false` denote distinct values, so there are exactly two values $\mathcal{S}$(`bool`).

Although `true, false, bool, :` and `=` have a predefined interpretation, all other functions must be explicitly introduced, including other boolean functions like `and`, `or`, etc. We can introduce these boolean functions with an extension.

⟨Example: Boolean extension 17b⟩ ≡

```
Extension: Boolean
  introduces
      not(bool) -> bool,
      and(bool, bool) -> bool,
      or(bool, bool) -> bool,
      xor(bool, bool) -> bool,
      implies(bool, bool) -> bool;
  requires (for x, y: bool)
      not(true) = false,
      not(false) = true,
      (true and x) = x,
      (false and x) = false,
      (x or y) = not(not(x) and not(y)),
      (x xor y) = not(x = y),
      (x implies y) = (not(x) or y).
```

◇

Macro referenced in scrap 27.

The equations in the requires clause can be used to show that any ground expression (an expression with no variable symbols) involving the new functions is equal to one of the expressions `true` or `false`, so the generates clause of the original concept is preserved. So is the property that `true` and `false` denote distinct values. (One way to prove the latter statement is to treat the equations as a set of rewrite rules and show that they have the properties of confluence and finite termination, such that `true` and `false` are both irreducible.)

We have already given definitions of Set and Binary-op concepts, which are a useful base for many other concept definitions, including the Natural number concept.

⟨Example: Natural number concept 18a⟩ ≡

```
⟨Example: Set and Binary-op definitions 4⟩

Definition: Natural
    refines Binary-op [with naturals as domain],
            Binary-op [with naturals as domain, - as +],
            Binary-op [with naturals as domain, * as +];
    introduces
       0 -> naturals,
       1 -> naturals,
       succ(naturals) -> naturals,
       <=(naturals, naturals) -> bool;
    generates naturals freely using 0, succ;
    requires (for n, m: naturals)
       1 = succ(0),
       (0 = 0) = true,
       (succ(n) = 0) = false,
       (0 = succ(n)) = false,
       (succ(m) = succ(n)) = (m = n),
       n + 0 = n,
       n + succ(m) = succ(n + m),
       n - 0 = n,
       0 - n = 0,
       succ(n) - succ(m) = n - m,
       i * 0 = 0,
       i * succ(j) = i * j + i,
       0 <= n,
       not(succ(n) <= 0),
       (succ(m) <= succ(n)) = (m <= n).
◇
```

Macro referenced in scrap 27.

3.2 Some algebraic concepts

Many basic algebraic concepts can easily be described in Tecton.

⟨Example: Algebraic concepts 18b⟩ ≡

```
⟨Example: Set and Binary-op definitions 4⟩
Definition: Associative
  refines Binary-op;
  requires (for x, y, z: domain) x + (y + z) = (x + y) + z.

Abbreviation: Semigroup
```

```
  is Associative.

Definition: Commutative
  refines Binary-op;
  requires (for x, y: domain) x + y = y + x.

Definition: Left-identity
  refines Binary-op;
  introduces 0 -> domain;
  requires (for x: domain) 0 + x = x.

Definition: Right-identity
  refines Binary-op;
  introduces 0 -> domain;
  requires (for x: domain) x + 0 = x.

Definition: Identity
  refines Left-identity, Right-identity.

Definition: Monoid
  refines Semigroup, Identity.

Definition: Abelian-monoid
  refines Monoid, Commutative.

Definition: Left-inverses
  refines Left-identity;
  introduces -(domain) -> domain;
  requires (for x: domain) -(x) + x = 0.

Definition: Right-inverses
  refines Right-identity;
  introduces -(domain) -> domain,
    -(x: domain, y: domain) = x + -(y);
  requires (for x: domain) x - x = 0.

Definition: Inverses
  refines Left-inverses, Right-inverses.
Definition: Group
  refines Monoid, Inverses.

Definition: Abelian-group
  refines Group, Commutative.
◇
```

Macro referenced in scrap 27.

The following defines exponentiation in the most general possible setting.

⟨Exponentiation concept 20a⟩ ≡

```
Definition: Exponentiation
   uses Natural, Monoid [with * as +, 1 as 0];
   introduces ^(domain , naturals) -> domain;
   requires (for x: domain; n: naturals)
     x ^ 0 = 1,
     x ^ (n + 1) = (x ^ n) * x.
◇
```

Macro referenced in scrap 27.

When we use a concept instance as in the Exponentiation definition, we can make it easier to use the instance as a parameter by introducing an abbreviation for it:

⟨Example: Mult-monoid abbreviation 20b⟩ ≡

```
Abbreviation: Mult-monoid is Monoid [with * as +, 1 as 0].
◇
```

Macro referenced in scrap 27.

To make explicit that Mult-monoid can be treated as a parameter of Exponentiation, we can write:

⟨Example: Extension with uses clause 20c⟩ ≡

```
Extension: Exponentiation
   uses Mult-monoid.
◇
```

Macro referenced in scrap 27.

Now we can extend Natural to have ^ as exponentiation.

⟨Example: Extension of Natural number concept to have exponentiation 20d⟩ ≡

```
Extension: Natural
   uses Exponentiation [with Natural as Mult-monoid, naturals as domain].
◇
```

Macro referenced in scrap 27.

3.3 Relations and ordering

It is also easy to describe basic properties of relations in Tecton.

⟨Example: Relations 20e⟩ ≡

```
Definition: Unary-relation
  refines Set;
  introduces p(domain) -> bool.

Definition: Binary-relation
  refines Set;
  introduces r(domain, domain) -> bool.

Definition: Reflexive
  refines Binary-relation;
  requires (for x: domain) x R x.

Definition: Anti-reflexive
```

```
  refines Binary-relation;
  requires (for x: domain) not(x R x).

Definition: Symmetric
  refines Binary-relation;
  requires (for x, y: domain) x R y implies y R x.

Definition: Anti-symmetric
  refines Binary-relation;
  requires (for x, y: domain) (x R y and y R x) implies x = y.

Definition: Transitive
  refines Binary-relation;
  requires (for x, y, z: domain) (x R y and y R z) implies x R z.

Definition: Equivalence-relation
  refines Reflexive, Symmetric, Transitive.

Definition: Partial-order
  refines Reflexive, Anti-symmetric, Transitive.

Definition: Strict-partial-order
  refines Anti-reflexive, Transitive.

Extension: Partial-order
  introduces <(x: domain, y: domain) = ((x R y) and not(x = y)).

Lemma: Partial-order implies Strict-partial-order [with < as R].

Definition: Strict-weak-order
  refines Strict-partial-order [with < as R],
          Transitive [with equiv as R];
  requires (for x, y: domain)
    equiv(x, y) = (not(x < y) and not(y < x)).

Definition: Total-order
   refines Partial-order;
   requires (for x, y: domain) x R y or y R x.

Definition: Strict-total-order
   refines Strict-partial-order;
   requires (for x, y: domain) x R y or y R x or x = y.

Lemma: Strict-total-order implies Strict-weak-order.

Lemma: Strict-weak-order implies Equivalence-Relation [with equiv as R].
```
◇

Macro referenced in scrap 27.

3.4 Binary-trees

[[This a very limited version of binary trees that was used in the carry-lookahead adder example in [16].]]

⟨Example: Binary-tree concept 21⟩ ≡

```
Abbreviation: Index-set
   is Set [with indices as domain].

Definition: Binary-tree-index
  refines Index-set;
  uses Natural;
  introduces
    nonroot < indices,
    root -> indices,
    left(indices) -> indices,
    right(indices) -> indices,
    parent(nonroot) -> indices,
    depth(indices) -> naturals;
  generates indices freely using root, left, right;
  requires (for i: indices)
    i: nonroot = not(i = root),
    depth(root) = 0,
    depth(left(i)) = depth(i) + 1,
    depth(right(i)) = depth(i) + 1,
    parent(left(i)) = i,
    parent(right(i)) = i.

Extension: Binary-tree-index
  introduces level-order-index(indices) -> naturals,
  requires (for i: indices)
    level-order-index(root) = 1
    level-order-index(left(i)) = 2 * level-order-index(i),
    level-order-index(right(i)) = 2 * level-order-index(i) + 1.
◇
```

Macro referenced in scrap 27.

References

[1] R. S. Boyer, D. M. Goldschlag, M. Kaufman, and J S. Moore. "Functional Instantiation in First-Order Logic," in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, edited by Vladimir Lifschitz, Academic Press, 1991.

[2] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[3] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[4] P. Briggs, *Nuweb, a simple literate programming tool*, Version 0.87, 1989.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1991.

[6] W. M. Farmer, J. D. Guttman, F. J. Thayer, "Little Theories," in D. Kapur, ed., *Proc. of 11th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence 607*, June 1992, 567-581.

[7] S. J. Garland, J. V. Guttag, and J. J. Horning, "Debugging Larch Shared Language Specifications," *IEEE Trans. Software Engineering*, Vol 16, No. 9, September 1990, 1044-1057.

[8] M. Gordon, "A proof generating system for Higher-Order Logic," in: G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, Boston, 1988, pp. 73-128.

[9] J. A. Goguen and R. Burstall, "Putting theories together to make specifications," in Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045-1058.

[10] J. A. Goguen, T. Winker, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, "Introducing OBJ," to appear in J.A. Goguen, D. Coleman, and R. Gallimore (editors). *Applications of Algebraic Specification using OBJ*, Cambridge University Press, 1992.

[11] J. A. Goguen "OBJ as a theorem prover, with application to hardware verification." In P. Subrahmanyam and Graham Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 218-267, Springer-Verlag, 1989.

[12] J. A. Goguen and José Meseguer. *Order sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations.* Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989.

[13] J. V. Guttag and J. J. Horning, "Report on the Larch Shared Language," *Sci. Comput. Program.*, vol. 6, no. 2, pp. 103-134, Mar. 1986.

[14] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract Data Types and Software Validation," *Comm. ACM*, 21, December 1978, 1049-1064.

[15] J. Joyce. *Multi-Level Verification of Microprocessor-Based Systems.* PhD Thesis, University of Cambridge, December 1989.

[16] D. Kapur and D. R. Musser, *Tecton: a framework for specifying and verifying generic system components*, RPI Computer Science Department Technical Report 92-20, Troy, NY, July 1992.

[17] D. Kapur and D. R. Musser, *Examples of Tecton concept descriptions*, working paper, May, 1992.

[18] D. Kapur, D. R. Musser, and Alexander A. Stepanov, "Operators and algebraic structures," *Proc. of Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire, October 1981.

[19] D. Kapur, D. R. Musser, and A. A. Stepanov, "Tecton: a language for manipulating generic objects," *Proc. of Program Specification Workshop, University of Aarhus, Denmark*, August 1981, *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 134, 1982.

[20] D. Kapur, D. R. Musser, and X. Nie, "An overview of the Tecton Proof System," *Proc. of a Workshop on Formal Methods in Databases and Software Engineering*, Concordia University, Montreal, May 15-16, 1992.

[21] D. Kapur and P. Narendran, "An equational approach to theorem proving in first-order predicate calculus," *Proc. of 8th IJCAI*, Los Angeles, Calif, 1985.

[22] D. Kapur and H. Zhang, "RRL: Rewrite Rule Laboratory," *Proc. of Ninth International Conference on Automated Deduction (CADE-9)*, Argonne, IL, May 1988.

[23] D.E. Knuth, Literate programming. *Computer Journal* 27 (1984), 97–111.

[24] T. F. Melham, "Abstraction mechanisms for hardware verification," in: G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, Boston, 1988, pp. 267-291.

[25] J. Palsberg and M. Schwartzbach, "Type substitution for object-oriented programming," *OOPSLA ECOOP '90 Proceedings*, Ottawa, Canada, October, 1990, 151-168.

[26] J. Rushby and F. W. von Henke. Formal verification of the interactive convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989.

[27] L. Théry, Y. Bertot, and G. Kahn, "Real theorem provers deserve real user-interfaces," University of Cambridge report, May, 1992.

[28] F. W. von Henke, J. S. Crowe, R. Lee, J. M. Rushby, and R. A. Whitehurst, "EHDM Verification Environment: An Overview," in *11th National Computer Security Conference*, Baltimore, Maryland, 1988.

A Sort-correctness

Let e be an expression in which no undeclared variables and no undeclared function identifiers appear. The sort of e, and the property of sort-correctness, are defined inductively as follows:

1. If e is a variable identifier, the sort of e is the declared sort of the variable.

2. If e is $v : s$, s must be a sort declared as a subsort of s', the sort of v. The sort of e is `bool`.

3. If e is f, where f is a function identifier, let $f \to r_1, \ldots, f \to r_k$ be the function descriptions declared for f. Then in subsequent sort-checking each of the sorts $r_1, \ldots, r_k$ is a candidate for the sort of e.

4. If e is $f(a_1, \ldots, a_n)$ or $a_1 f a_2$ (here $n = 2$), then for each n-argument function description $f(d_1, \ldots, d_n) \to r$ declared for f, the sort of a_i must be d_i or a subsort of d_i, for $i = 1, \ldots, n$, and the sort r is a candidate for the sort of e.

5. If e is `(for` *variable-declaration*-LIST`)` e' or `(for some` *variable-declaration*-LIST`)` e', the sort of e' must be `bool`, which is also the sort of e.

6. If e is `if` e_1 `then` e_2 `else` e_3, the sort of e_1 must be `bool` and e_2 and e_3 must have the same sort, which is also the sort of e.

7. If e is e_1 `where` $x = e_2$, then occurrences of x in e_1 are considered to have the sort of e_2, (regardless of whether x is declared or undeclared), and the sort of e is that of e_1.

8. If e is `(` e' `)`, the sort of e is the sort of e'.

If all of the requirements on subexpressions stated in the above rules are satisfied for a unique set of function descriptions, e is said to be sort-correct.

In the following example from Section 3.2, there are two available function descriptions for 1, namely $1 \rightarrow$ domain and $1 \rightarrow$ naturals, but in `x ^ 0 = 1`, only the former leads to sort-correctness for the whole expression.

⟨Example: Exponentiation concept 25⟩ ≡

```
Definition: Exponentiation
   uses Natural, Monoid [with * as +, 1 as 0];
   introduces ^(domain , naturals) -> domain;
   requires (for x: domain; n: naturals)
     x ^ 0 = 1,
     x ^ (n + 1) = (x ^ n) * x.
```

◇

Macro referenced in scrap 27.

B Rules for Assigning Truth Values to Expressions

Let D be a concept description with subject C and let $a = (\mathcal{S}, \mathcal{F})$ be an algebra in syn C. Define an $\mathcal{S}$ *valuation* ν to be a mapping of declared variables to values, in which the value $\nu(v)$ is from the set $\mathcal{S}(s)$ when the declared sort of v is s.

We assume `true` and `false` are declared as nullary function identifiers with range sort `bool`, which is assumed to be always assigned by $\mathcal{S}$ to a set containing exactly two distinct values, **true** and **false**. We also assume that for every sort s, the $=$ function identifier is declared as $= (s, s) \rightarrow$ bool and $\mathcal{F}(= (s, s) \rightarrow \text{ bool})$ is equality in the set $\mathcal{S}(s)$.

For any expression e containing no undeclared variables or function identifiers, we define inductively the value that $a^\nu = (\mathcal{S}, \mathcal{F}, \nu)$ assigns to e:

1. If e is a variable identifier v, $a^\nu(e) = \nu(v)$.

2. If e is $v : s$, where s is a subsort of the declared sort of v, then $a^\nu(e) =$ **true** if $\nu(v) \in \mathcal{S}(s)$; **false**, otherwise.

3. If e is f, where f is any nullary function identifier other than `true` or `false`, let σ be the function description declared[7] for f. Then $a^\nu(e)$ is the nullary function (a constant) $\mathcal{F}(\sigma)$.

[7] In case of overloading, the function description used is the one selected by the sort-correctness algorithm.

4. If e is $f(a_1, \ldots, a_n)$ or $a_1 f a_2$, let σ be the function description declared for f. Then $a^\nu(e) = \mathcal{F}(\sigma)(v_1, \ldots, v_n)$ where $v_i = a^\nu(a_i), i = 1, \ldots, n$.

5. If e is `(for` ***variable-declaration*-LIST**`)` e', then $a^\nu(e) = \textbf{true}$ if $a^{\nu'}(e') = \textbf{true}$ for every ν' with $\nu'(y) = \nu(y)$ for every variable y not in *variable-declaration*-LIST; **false**, otherwise.

6. If e is `(for some` ***variable-declaration*-LIST**`)` e', then $a^\nu(e) = \textbf{true}$ if $a^{\nu'}(e') = \textbf{true}$ for some ν' with $\nu'(y) = \nu(y)$ for every variable y not in *variable-declaration*-LIST; **false**, otherwise.

7. If e is `if` e_1 `then` e_2 `else` e_3, then $a^\nu(e) = a^\nu(e_2)$ if $a^\nu(e_1) = \textbf{true}$, and $a^\nu(e) = a^\nu(e_3)$ otherwise.

8. If e is e_1 `where` $x = e_2$, then $a^\nu(e) = a^{\nu'}(e_1)$, where $\nu'(x) = a^\nu(e_2)$, and $\nu'(y) = \nu(y)$ for every variable y other than x.

9. If e is `(` e' `)`, then $a^\nu(e) = a^\nu(e')$.

Note that `true, false, bool, :` and `=` have a fixed interpretation, unlike other identifiers.

C Index of examples

D Collected examples

[[For now the examples are collected here and written to a file just to avoid warnings from NUWEB about unreferenced parts. I plan to assemble the parts in various ways as test files for Tecton language processors.]]

`"Tecton-examples.tec"` 27 ≡

⟨Example: Associative concept 5⟩
⟨Example: Characterization of nonzero natural numbers 6⟩
⟨Example: Generates clause in definition natural number concept 7⟩
⟨Example: Tecton definitions with refines and uses clauses 9⟩
⟨Example: Set uses Boolean 10⟩
⟨Example: Concept instances 11⟩
⟨Example: Concept replacement 13a⟩
⟨Example: Concept abbreviations 13b⟩
⟨Example: Lemmas asserting additional properties 14⟩
⟨Example: Lemmas relating two concept instances 15a⟩
⟨Example: Extension of Sequence concept 15b⟩
⟨Example: Realization of the Stack concept by the Sequence concept 16b⟩
⟨Example: Boolean concept 17a⟩
⟨Example: Boolean extension 17b⟩
⟨Example: Natural number concept 18a⟩
⟨Example: Algebraic concepts 18b⟩
⟨Example: Mult-monoid abbreviation 20b⟩
⟨Example: Extension with uses clause 20c⟩
⟨Example: Extension of Natural number concept to have exponentiation 20d⟩
⟨Example: Extensions that introduce new functions 16a⟩
⟨Exponentiation concept 20a⟩
⟨Example: Exponentiation concept 25⟩
⟨Example: Relations 20e⟩
⟨Example: Binary-tree concept 21⟩
◇