国 际 电 信 联 盟

CCITT

国际电报电话咨询委员会

蓝 皮 书

卷 X.4

建议Z.100的附件F.2
SDL形式定义

静态语义学

第 九 次 全 体 会 议

1988年11月14—25日 墨尔本

1989年 日内瓦

国 际 电 信 联 盟

CCITT

国 际 电 报 电 话 咨 询 委 员 会

蓝 皮 书

卷 X.4

建议Z.100的附件F.2
SDL形式定义

静态语义学

第 九 次 全 体 会 议

1988年11月14—25日 墨尔本

1989年 日内瓦

ISBN 92-61-03785-2

中国印刷

CCITT 图书目录
第九次全体会议（1988 年）

蓝 皮 书

卷 I

卷 I.1 — 全会会议记录和报告
研究组及研究课题一览表

卷 I.2 — 意见和决议
关于 CCITT 的组织和工作程序的建议（A 系列）

卷 I.3 — 术语和定义　缩略语和首字母缩写词　关于措词含义的建议（B 系列）和综合电信统计的建议（C 系列）

卷 I.4 — 蓝皮书索引

卷 II

卷 II.1 — 一般资费原则 — 国际电信业务的资费和帐务　D 系列建议（第 III 研究组）

卷 II.2 — 电话网和 ISDN — 运营、编号、选路和移动业务　建议 E.100-E.333（第 II 研究组）

卷 II.3 — 电话网和 ISDN — 服务质量、网络管理和话务工程　建议 E.401-E.880（第 II 研究组）

卷 II.4 — 电报业务和移动业务 — 运营和服务质量　建议 F.1-F.140（第 I 研究组）

卷 II.5 — 远程信息处理业务、数据传输业务和会议电信业务 — 运营和服务质量　建议F.160-F.353、F.600、F.601、F.710-F.730（第 I 研究组）

卷 II.6 — 报文处理和查号业务 — 运营和服务的限定　建议 F.400-F.422、F.500（第 I 研究组）

卷 III

卷 III.1 — 国际电话连接和电路的一般特性　建议 G.100-G.181（第 XII 和 XV 研究组）

卷Ⅲ.2 — 国际模拟载波系统　建议 G.211-G.544（第 XV 研究组）

卷Ⅲ.3 — 传输媒质 — 特性　建议 G.601-G.654（第 XV 研究组）

卷Ⅲ.4 — 数字传输系统的概况；终端设备　建议 G.700-G.795（第 XV 和第 XⅧ研究组）

卷Ⅲ.5 — 数字网、数字段和数字线路系统　建议 G.801-G.956（第 XV 和第 XⅧ研究组）

卷Ⅲ.6 — 非话信号的线路传输　声音节目和电视信号的传输　H 和 J 系列建议（第 XV 研究组）

卷Ⅲ.7 — 综合业务数字网（ISDN）— 一般结构和服务能力　建议 I.110-I.257（第 XⅧ研究组）

卷Ⅲ.8 — 综合业务数字网（ISDN）— 全网概貌和功能、ISDN 用户—网络接口　建议 I.310-I.470（第 XⅧ研究组）

卷Ⅲ.9 — 综合业务数字网（ISDN）— 网间接口和维护原则　建议 I.500-I.605（第 XⅧ研究组）

卷Ⅳ

卷Ⅳ.1 — 一般维护原则：国际传输系统和电话电路的维护　建议 M.10-M.782（第Ⅳ研究组）

卷Ⅳ.2 — 国际电报、相片传真和租用电路的维护　国际公用电话网的维护　海事卫星和数据传输系统的维护　建议 M.800-M.1375（第Ⅳ研究组）

卷Ⅳ.3 — 国际声音节目和电视传输电路的维护　N 系列建议（第Ⅳ研究组）

卷Ⅳ.4 — 测量设备技术规程　O 系列建议（第Ⅳ研究组）

卷 V — 电话传输质量　P 系列建议（第 XⅡ 研究组）

卷Ⅵ

卷Ⅵ.1 — 电话交换和信令的一般建议　ISDN 中服务的功能和信息流　增补　建议 Q.1-Q.118（乙）（第Ⅺ研究组）

卷Ⅵ.2 — 四号和五号信令系统技术规程　建议 Q.120-Q.180（第Ⅺ研究组）

卷Ⅵ.3 — 六号信令系统技术规程　建议 Q.251-Q.300（第Ⅺ研究组）

卷Ⅵ.4 — R1 和 R2 信令系统技术规程　建议 Q.310-Q.490（第Ⅺ研究组）

卷Ⅵ.5 — 综合数字网和模拟—数字混合网中的数字本地、转接、组合交换机和国际交换机　增补　建议 Q.500-Q.554（第Ⅺ研究组）

卷Ⅵ.6 — 各信令系统之间的配合　建议 Q.601-Q.699（第Ⅺ研究组）

卷Ⅵ.7 — 七号信令系统技术规程　建议 Q.700-Q.716（第Ⅺ研究组）

卷Ⅵ.8 — 七号信令系统技术规程　建议 Q.721-Q.766（第Ⅺ研究组）

卷Ⅵ.9 — 七号信令系统技术规程　建议 Q.771-Q.795（第Ⅺ研究组）

卷Ⅵ.10 — 一号数字用户信令系统（DSS 1）　数据链路层　建议 Q.920-Q.921（第Ⅺ研究组）

卷Ⅵ.11 — 一号数字用户信令系统（DSS 1）　网络层、用户—网路管理　建议 Q.930-Q.940（第Ⅺ研究组）

卷Ⅵ.12 — 公用陆地移动网 与ISDN和PSTN的互通 建议Q.1000-Q.1032（第Ⅺ研究组）

卷Ⅵ.13 — 公用陆地移动网 移动应用部分和接口 建议Q.1051-Q.1063（第Ⅺ研究组）

卷Ⅵ.14 — 与卫星移动通信系统的互通 建议Q.1100-Q.1152（第Ⅺ研究组）

卷Ⅶ

卷Ⅶ.1 — 电报传输 R系列建议 电报业务终端设备 S系列建议 （第Ⅸ研究组）

卷Ⅶ.2 — 电报交换 U系列建议（第Ⅸ研究组）

卷Ⅶ.3 — 远程信息处理业务的终端设备和协议 建议T.0-T.63（第Ⅷ研究组）

卷Ⅶ.4 — 智能用户电报各建议中的一致性测试规程 建议T.64（第Ⅷ研究组）

卷Ⅶ.5 — 远程信息处理业务的终端设备和协议 建议T.65-T.101，T.150-T.390（第Ⅷ研究组）

卷Ⅶ.6 — 远程信息处理业务的终端设备和协议 建议T.400-T.418（第Ⅷ研究组）

卷Ⅶ.7 — 远程信息处理业务的终端设备和协议 建议T.431-T.564（第Ⅷ研究组）

卷Ⅷ

卷Ⅷ.1 — 电话网上的数据通信 V系列建议（第ⅩⅦ研究组）

卷Ⅷ.2 — 数据通信网：业务和设施，接口 建议X.1-X.32（第Ⅶ研究组）

卷Ⅷ.3 — 数据通信网：传输，信令和交换，网络概貌，维护和管理安排 建议X.40-X.181（第Ⅶ研究组）

卷Ⅷ.4 — 数据通信网：开放系统互连（OSI）— 模型和记法表示，服务限定 建议X.200-X.219（第Ⅶ研究组）

卷Ⅷ.5 — 数据通信网：开放系统互连（OSI）— 协议技术规程，一致性测试 建议X.220-X.290（第Ⅶ研究组）

卷Ⅷ.6 — 数据通信网：网间互通，移动数据传输系统，网间管理 建议X.300-X.370（第Ⅶ研究组）

卷Ⅷ.7 — 数据通信网：报文处理系统 建议X.400-X.420（第Ⅶ研究组）

卷Ⅷ.8 — 数据通信网：号码簿 建议X.500-X.521（第Ⅶ研究组）

卷Ⅸ — 干扰的防护 K系列建议（第Ⅴ研究组） 电缆及外线设备的其他部件的结构、安装和防护 L系列建议（第Ⅵ研究组）

卷Ⅹ

卷Ⅹ.1 — 功能规格和描述语言（SDL） 使用形式描述方法（FDT）的标准 建议Z.100和附件A、B、C和E 建议Z.110（第Ⅹ研究组）

卷Ⅹ.2 — 建议Z.100的附件D：SDL用户指南（第Ⅹ研究组）

卷X.3 — 建议Z.100的附件F.1：SDL形式定义 介绍（第X研究组）

卷X.4 — 建议Z.100的附件F.2：SDL形式定义 静态语义学（第X研究组）

卷X.5 — 建议Z.100的附件F.3：SDL形式定义 动态语义学（第X研究组）

卷X.6 — CCITT高级语言（CHILL） 建议Z.200（第X研究组）

卷X.7 — 人机语言（MML） 建议Z.301-Z.341（第X研究组）

蓝皮书卷 X. 4 目录

建议 Z. 100 的附件 F. 2

卷　首　说　明

1　在 1989—1992 研究期内委托给各研究组的研究课题可查阅给该研究组的第 1 号文献。

2　本卷中的“主管部门”一词是电信主管部门和经认可的私营机构两者的简称。

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

目　　录

卷 X.4

建议 Z.100 的附件 F.2

SDL 形式定义
静态语义学

引 言

这部分形式定义规定SDL的静态性质。对于形式定义的整体结构的描述和所用符号的说明参见附件F.1：形式定义介绍。

附件F.2定义以下内容：

- 所用的良形式条件。
  但是，为了方便起见，涉及到等式细节的良形式条件的介绍将推迟到在附件F.3：动态语义中构造实体字典 *Entity-dict* 的时候（见附件F.3的第5章的引言）。即使这些条件被放入动态语义中，它们也是静态语义性质的，因此在对SDL进程进行任何解释之前要先应用这些良形式条件。

  这些良形式条件如下：
  1. 检查在一个判定中的各个回答不应该重叠。
  2. 检查一个被包围的作用域单位不应该使得在包围的作用域单位中定义的某个类别的性质无效。
  3. 检查不应该有字面值等于错误项。
  4. 检查布尔字面值TRUE和FALSE应代表不同的值。
- 具体语法和抽象语法之间的关系在Z.100中定义（叫做AS_1）。在本形式定义中不再重复AS_1。有一个摘要在Z.100的附件B中。为了把AS_1的实物和Meta-IV函数中的其他实物区别开来，每个AS_1的名字都带有一个后缀“1”。
  例如，在Z.100的§2.2.2中规定的标识符 *identifier* 的定义是：

  Identifier :: *Qualifier Name*

  而在形式定义中的相应定义为：

  $Identifer_1$ :: $Qualifier_1$ $Name_1$

1 表示具体语法的抽象语法

以下定义了反映具体语法的域 AS_0。尽管 AS_0 在一种抽象形式上表示了具体语法，但在某些方面它确实是在比 Z.100 中的语法规则“更低的层次”上定义了此语言的语法。这是由于下述事实，出现在语法规则中的上下文有关的信息并没有反映到 AS_0 中（解释见附件 F.1 的第三章）。

与 AS_1 中相反，AS_0 中使用了域名的缩写形式以减少实际行宽有限所引起的问题。附在域定义后面的注释定义了域的全名，对于出现在缩写中的字符采用了斜体字形。

为了把 AS_0 的实物与 *Meta-IV* 函数中的实物区别开，每个 AS_0 域都带有一个后缀“0”。

由于 AS_0 是从正文语法中导出的抽象语法，以下几项不包含在形式定义之内：

- 词法规则（包括宏）
  在 Z.100、§2.2.1 中形式地定义了这些规则的大部分
- 语法规则（AS_0 没有规定放置关键字和分隔符的位置，它只规定了组成一个系统定义的实物）。
  在 Z.100 的正文语法各小节中形式地定义了语法规则
- 正文语法与图形语法之间的关系
  在多数情况下，这种关系是直接对应的，也就是〈图〉对应于〈定义〉。在图形语法中的符号“**is followed by**”在正文语法中就用文字“is followed by”来表示。

域的定义的次序与在 Z.100 中定义的对应的语法规则的次序相同。例如，在 AS_0 中的第一个域定义是 Id_0，而在 Z.100 中的第一个 BNF 产生式是〈identifier〉。

1.1 基本 SDL

1	Id_0	::	$Qualifier_0$ $Name_0$
2	$Qualifier_0$	=	$Qualifierelem_0^*$
3	$Qualifierelem_0$	=	$Scopeunit_0$ $Name_0$
4	$Scopeunit_0$	=	**SYSTEM | BLOCK | PROCESS | SERVICE | PROCEDURE | SUBSTRUCTURE | TYPE | SIGNAL**
5	$Name_0$	::	$Char^+$ **[EXCLAMATIONMARK]**

- 一个标识符 *Id* 由一个限定词 *Qualifier* 和一个名字 *Name* 组成。
- 一个限定词 *Qualifier* 是一个可以为空表的限定词元素 *Qualifierelem* 表。
- 一个限定词元素 *Qualifierelem* 是一个作用域单位类型 *Scopeunit* 和一个名字 *Name*。
- 一个作用域单位类型 *Scopeunit* 可以是系统、功能块、进程、服务、过程、或者是一个功能块（信道）子结构、一个部分数据类型定义或者是一个信号的具体化。
- 一个名字 *Name* 由一个非空字符表组成，可以后随一个任选感叹号。
  为了方便起见，在 AS_0 中的名字 *Name* 包含了一类名字，与 Z.100 中 BNF 语法规则所定义的一类名字略有不同。
  下面是有关 AS_0 的假定：
  — 跟在拼字 $Char^+$ 后面的任选感叹号仅能出现在按照 BNF 规则在语法上允许出现的地方。

— 拼字是一种与 Z.100 中定义的词法规则相一致的大写字符序列。

6 $String_0$ :: $Char^*$

- 字符串 *String* 由一个可空的字符表组成。

7 $Text_0$ :: $String_0$

- 非形式的正文 *Text* 包含一个字符串 *String*。在表达式中，非形式正文 *Text* 不能与串 *String* 区别开来。因此，在 AS_0 中，正文 *Text* 只能在任务 *Task* 中出现，用来替换赋值语句。

8 Sys_0 :: $Sysdef_0$ $Remotedef_0$

- 系统 *Sys* 由一个系统定义 *Sysdef* 和一个间接定义 *Remotedef* 组成。

9 $Remotedef_0$ = $Refdecl_0^*$
10 $Refdecl_0$ = $Blockdef_0$ | $Prdef_0$ | $Servicedef_0$ | $Procdef_0$ | $Chansubdef_0$

- 间接定义 *Remotedef* 是一连串被引用的声明 *Refdecl* 组成的表。
- 被引用的声明 *Refdecl* 可以是一个功能块定义 *Blockdef*、一个进程定义 *Prdef*、一个服务定义 *Servicedef*、一个过程定义 *Procdef* 或一个信道子结构定义 *Chansubdef*。

功能块子结构也可以是间接的。但是，因为不可能从语法上来区别间接功能块子结构和间接信道子结构，如果上下文说明此间接定义实际上是个功能块子结构，就必须将信道子结构转换成功能块子结构（参见消去引用函数 *remove-references*）。

11 $Sysdef_0$ :: $Sysname_0$ $Sysdecl_0^*$ [$Tailname_0$]
12 $Sysname_0$ = $Name_0$
13 $Sysdecl_0$ = $Blockdef_0$ | $Blockref_0$ | $Chandef_0$ | $Sigdef_0$ | $Signallistdef_0$ | $Datadef_0$ | $Select_0$

- 系统定义 *Sysdef* 由一个系统名 *Sysname*、一个系统声明 *Sysdecl* 表和一个结束系统定义的任选的尾名 *Tailname* 组成。
- 系统名 *Sysname* 是一个名字 *Name*。
- 系统声明 *Sysdecl* 可以是一个功能块定义 *Blockdef*、一个功能块引用 *Blockref*、一个通道定义 *Chandef*、一个信号定义 *Sigdef*、一个信号表定义 *Signallistdef*、一个数据定义 *Datadef* 或者是一个选择定义 *Select*。

14 $Blockref_0$ :: $Blockname_0$
15 $Blockdef_0$ :: $Blockid_0$ $Blockdecl_0^*$ [$Blocksub_0$] [$Tailid_0$]
16 $Blockid_0$ = Id_0
17 $Tailid_0$ = Id_0
18 $Blockdecl_0$ = $Sigdef_0$ | $Prdef_0$ | $Prref_0$ | $Datadef_0$ | $Connect_0$ | $Sigroutedef_0$ | $Signallistdef_0$ | $Select_0$
19 $Blockname_0$ = $Name_0$

- 一个功能块引用 *Blockref* 包含一个功能块名 *Blockname*。

- 功能块定义 *Blockdef* 包含一个功能块标识符 *Blockid*、一连串的功能块声明 *Blockdecl*、一个任选功能块子结构定义 *Blocksub* 和一个结束功能块定义的任选尾标识符 *Tailid*。
  应该注意，在具体语法中不可能从语法上区别名字与标识符。因此，在名字和标识符都允许的地方 AS_0 具有标识符 *Id*。
- 功能块标识符 *Blockid* 是一个标识符 *Id*。
- 尾标识符 *Tailid* 是一个标识符 *Id*。
- 功能块声明 *Blockdecl* 可以是一个信号定义 *Sigdef*、一个进程定义 *Prdef*、一个进程引用 *Prref*、一个数据定义 *Datadef*、一个属于信号路由连接 *Connect* 的信道、一个信号路由定义 *Sigroutedef*、一个信号表定义 *Signallistdef* 或是一个选择 *Select*。
- 功能块名 *Blockname* 是一个名字 *Name*。

20	$Prref_0$	::	$Prname_0$ [$Instances_0$]
21	$Prname_0$	=	$Name_0$
22	$Prdef_0$	::	$Prid_0$ [$Instances_0$] $Parm_0$* [$Inputset_0$] $Prdecl_0$* $Processbody_0$ [$Tailid_0$]
23	$Prid_0$	=	Id_0
24	$Prdecl_0$	=	$Vardef_0$ | $Viewdef_0$ | $Datadef_0$ | $Sigdef_0$ | $Signallistdef_0$ | $Importdef_0$ | $Procdef_0$ | $Procref_0$ | $Timerdef_0$ | $Select_0$
25	$Procref_0$	::	$Procname_0$
26	$Inputset_0$	::	[$Signallist_0$]
27	$Processbody_0$	=	$Body_0$ | $Decomposition_0$
28	$Parm_0$	::	$Varname_0$+ $Sortid_0$
29	$Instances_0$	::	[$Initial_0$] [$Maximum_0$]
30	$Initial_0$	=	$Expr_0$
31	$Maximum_0$	=	$Expr_0$

- 进程引用 *Prref* 包含一个进程名 *Prname* 和一个任选的说明实例 *Instance* 规格的数。
- 进程名 *Prname* 是一个名字 *Name*。
- 进程定义 *Prdef* 由一个进程标识符 *Prid*、一个任选的说明实例规格的数 *Instances*、一个进程形式参数 *Parm* 表、一个任选的有效输入信号集合 *Inputset*、一连串的进程声明 *Prdecl*、进程体 *Processbody* 和一个结束进程定义的任选尾标识符 *Tailid* 所组成。
- 进程标识符 *Prid* 是一个标识符 *Id*。
- 进程声明 *Prdecl* 可以是一个变量定义 *Vardef*、一个视见定义 *Viewdef*、一个数据定义 *Datadef*、一个信号定义 *sigdef*、一个信号表定义 *Signallistdef*、一个进口定义 *Importdef*、一个过程定义 *Procdef*、一个过程引用 *Procref*、一个定时器定义 *Timerdef* 或是一个选择 *Select*。
- 过程引用 *Procref* 包括一个过程名 *Procname*。
- 有效输入信号集 *Inputset* 含有一个可空（如果是 **nil** 的话）的信号表 *signallist*。
- 进程体 *Processbody* 可以是一个图形体 *Body* 或服务分解 *Decomposition*。
- 进程形式参数 *Parm* 由一个变量名 *Varname* 表和一个类别标识符 *Sortid* 组成。
- 说明实例规格的数 *Instances* 由一个任选的初始数字 *Initial* 和一个任选的最大数字 *Maximum* 组成。
- 初始数字 *Initial* 是一个简单的表达式 *Expr*。
- 最大数字 *Maximum* 是一个简单的表达式 *Expr*。

32 *Procdef*$_0$ :: *Procid*$_0$ *Procparm*$_0$* *Procdecl*$_0$* *Body*$_0$ [*Tailid*$_0$]
33 *Procid*$_0$ = *Id*$_0$
34 *Procdecl*$_0$ = *Vardef*$_0$ | *Datadef*$_0$ | *Procdef*$_0$ | *Procref*$_0$ | *Select*$_0$
35 *Procname*$_0$ = *Name*$_0$
36 *Procparm*$_0$ = *Inoutparm*$_0$ | *Inparm*$_0$
37 *Inoutparm*$_0$ :: *Varname*$_0$+ *Sortid*$_0$
38 *Inparm*$_0$ :: *Varname*$_0$+ *Sortid*$_0$

- 过程定义 *Procdef* 由一个过程标识符 *Procid*、一个过程形式参数表 *Procparm*、一个过程声明表 *Procdecl**、一个过程体 *Body* 和一个结束过程定义的任选尾标识符 *Tailid* 组成。
- 一个过程标识符 *Procid* 是一个标识符 *Id*。
- 过程声明 *Procdecl* 可以是一个变量定义 *Vardef*、数据定义 *Datadef*、一个过程定义 *Procdef*、一个过程引用 *Procref* 或一个选择 *Select*。
- 过程名 *Procname* 是一个名字 *Name*。
- 过程形式参数 *Procparm* 可以是一个变量 *In/Out* 参数 *Inoutparm* 或者是一个变量 *In* 参数 *Inparm*。
- *In/Out* 参数 *Inoutparm* 由一个变量名 *Varname* 表和一个类别标识符 *Sortid* 组成。
- *In* 参数 *Inparm* 由一个变量名 *Varname* 表和一个类别标识符 *Sortid* 组成。

39 *Chandef*$_0$ :: *Channame*$_0$ *Chanpath*$_0$ [*Chanpath*$_0$] [*Chansub*$_0$] [*Tailname*$_0$]
40 *Chanpath*$_0$ :: *Orig*$_0$ *Dest*$_0$ *Signallist*$_0$
41 *Channame*$_0$ = *Name*$_0$
42 *Orig*$_0$ = *Blockid*$_0$ | ENV
43 *Dest*$_0$ = *Blockid*$_0$ | ENV
44 *Tailname*$_0$ = *Name*$_0$

- 信道定义 *Chandef* 由一个信道名 *Channame*、一个或两个信道通信路径 *Chanpath*、一个任选的信道子结构 *Chansub* 定义和一个结束信道定义的任选尾名 *Tailname* 组成。信道通信路径 *Chanpath* 由一个起始端点 *Orig*、一个目的端点 *Dest* 和一个信号表 *Signallist* 组成。
- 起始端点 *Orig* 或者是一个功能块标识符 *Blockid*，或者是环境 **ENV**。
- 目的端点 *Dest* 或者是一个功能块标识符 *Blockid*，或者是环境 **ENV**。
- 结束一个定义的名字 *Name* 是尾名 *Tailname*。

45 *Sigroutedef*$_0$ :: *Sigroutename*$_0$ *Sigroutepath*$_0$ [*Sigroutepath*$_0$]
46 *Sigroutepath*$_0$ :: *Origin*$_0$ *Destination*$_0$ *Signallist*$_0$
47 *Origin*$_0$ = *Id*$_0$ | ENV
48 *Destination*$_0$ = *Id*$_0$ | ENV
49 *Sigroutename*$_0$ = *Name*$_0$

- 信号路由定义 *Sigroutedef* 包含一个信号路由名 *Sigroutename* 和一个或两个信号路由通信路径 *Sigroutepath*。
- 信号路由通信路径 *Sigroutepath* 包含一个起始进程或服务 *Origin*、一个目的进程或服务 *Destination* 和一个信号表 *Signallist*。
- 起始进程或服务 *Origin* 可以是一个标识符 *Id* 或是环境 **ENV**。
- 目的进程 *Destination* 可以是一个进程标识符 *Id* 或是环境 **ENV**。
- 信号路由名 *Sigroutename* 是一个名字 *Name*。

50 *Connect*$_0$ :: *Connectpoint*$_0$ *Id*$_0^+$
51 *Connectpoint*$_0$ = (*Id*$_0$ | ENV)

- 连接 *Connect* 包含一个连接点 *Connect-point* 和一些被连接的标识符 *Id*。注意，*Connect*$_0$ 表示信道端点的连接、信道与路由的连接、信号路由的连接和信道的连接这四种连接，这是因为如果这四种连接在选择定义中出现，就无法从语法上区别它们。
- 连接点 *Connectpoint* 是一个标识符 *Id* 或环境 **ENV**（在信道端点连接的情况下）。

52 *Sigdef*$_0$ :: *Sigelem*$_0^+$
53 *Sigelem*$_0$ :: *Signame*$_0$ *Sortid*$_0^*$ [*Refinement*$_0$]
54 *Signame*$_0$ = *Name*$_0$
55 *Sortid*$_0$ = *Id*$_0$

- 信号定义 *Sigdef* 由一个信号元素 *Sigelem* 表组成。
- 信号元素 *Sigelem* 由一个信号名 *Signame*、一个类别标识符 Sortid* 表和一个任选的具体化 *Refinement* 部分组成。
- 信号名 *Signame* 是一个名字 *Name*。
- 类别标识符 *Sortid* 是一个标识符 *Id*。

56 *Signallistdef*$_0$ :: *Signallistname*$_0$ *Signallist*$_0$
57 *Signallistname*$_0$ = *Name*$_0$
58 *Signallist*$_0$ = (*Signallistid*$_0$ | *Sigid*$_0$)$^+$
59 *Signallistid*$_0$ :: *Id*$_0$
60 *Sigid*$_0$ = *Id*$_0$

- 信号表定义 *Signallistdef* 由一个信号表名字 *Signallistname* 和一个信号表 *Signallist* 组成。
- 信号表名字 *Signallistname* 是一个名字 *Name*。
- 信号表 *Signallist* 是一个由信号表标识符 *Signallistid* 和信号标识符 *Sigid* 组成的非空表。
- 信号表标识符 *Signallistid* 含有一个标识符 *Id*。
- 信号标识符 *Sigid* 是一个标识符 *Id*。

注意，由于信号表标识符 *Signallistid* 是一棵树，所以总是可以区别信号表标识符和信号标识符（与在具体语法中的情形一样）。

61 *Vardef*$_0$ :: [REVEALED] [EXPORTED] *Vardefelem*$_0^+$
62 *Vardefelem*$_0$ :: *Varname*$_0^+$ *Sortid*$_0$ [*Value*$_0$]
63 *Varname*$_0$ = *Name*$_0$

- 变量定义 *Vardef* 由一个任选的透露属性 **REVEALED**、一个任选的出口属性 **EXPORTED** 和一个变量定义元素 *Vardefelem* 表组成。
- 变量定义元素 *Vardefelem* 由一个变量名 *Varname*$^+$ 表、一个类别标识符 *Sortid* 和一个任选的初始值 *Value* 组成。
- 变量名 *Varname* 是一个名字 *Name*。

64 *Viewdef*$_0$ :: *Viewdefelem*$_0^+$
65 *Viewdefelem*$_0$ :: *Varid*$_0^+$ *Sortid*$_0$
66 *Varid*$_0$ = *Id*$_0$

- 视见定义 *Viewdef* 由一个视见定义元素 *Viewdefelem* 表组成。
- 视见定义元素 *Viewdefelem* 由一个视见变量标识符 *Varid* 表和一个类别标识符 *Sortid* 组成。
- 变量标识符 *Varid* 是一个标识符 *Id*。

67 ***Body*$_0$** :: ***Transition*$_0$ *Statebody*$_0$***
68 ***Statebody*$_0$** :: **(*Statenamelist*$_0$ | *Starredlist*$_0$) *Statespec*$_0$* [*Tailname*$_0$]**
69 ***Statenamelist*$_0$** :: ***Statename*$_0$$^+$**
70 ***Statename*$_0$** = ***Name*$_0$**
71 ***Starredlist*$_0$** :: **[*Statenamelist*$_0$]**

- 体 *Body* 由起始跃迁 *Transition* 和一个状态体 *Statebody* 表组成。
- 一个状态体 *Statebody* 含有一个状态名表 *Statenamelist* 或一个带星号的名字表 *Starredlist*，并且后随一连串的状态激励规格 *Statespec* 和一个结束这个状态体的任选尾名 *Tailname*。
- 状态名表 *Statenamelist* 由一连串的状态名字 *Statename* 组成。
- 状态名 *Statename* 是一个名字 *Name*。
- 一个带星号的名字表 *Starredlist* 含有一个可空的（如果是 **nil** 的话）状态名表 *Statenamelist*。

72 ***Statespec*$_0$** = ***Savespec*$_0$ | *Inputspec*$_0$ | *Contspec*$_0$ | *Priinput*$_0$**

- 状态激励规格 *Statespec* 可以是一个保存规格 *Savespec*、一个输入规格 *Inputspec*、一个连续信号规格 *Contspec* 或是一个优先级输入规格 *Priinput*。

73 ***Savespec*$_0$** :: ***Signallist*$_0$ | *Starred*$_0$**
74 ***Inputspec*$_0$** :: **(*Starred*$_0$ | *Inputvars*$_0$$^+$) *Enabling*$_0$ *Transition*$_0$**
75 ***Starred*$_0$** :: **()**
76 ***Inputvars*$_0$** :: ***Sigid*$_0$ [*Varid*$_0$]***

- 保存规格 *Savespec* 包含一个信号表 *Signallist* 或一个星号。
- 输入规格 *Inputspec* 包含一个星号（即一棵带星 *Starred* 的树代表一个星号）或一个非空的输入信号变量表 *Inputvars*，并后随一个允许条件 *Enabling* 和一个跃迁 *Transition*。
- 输入信号变量 *Inputvars* 包含一个信号标识符 *Sigid* 和一个任选的变量标识符 *Varid* 表。

77 ***Transition*$_0$** :: ***Actstmt*$_0$* [*Termstmt*$_0$]**
78 ***Actstmt*$_0$** :: **[*Label*$_0$] *Act*$_0$**
79 ***Label*$_0$** = ***Name*$_0$**

- 跃迁 *Transition* 包含一连串的动作语句 *Actstmt* 和一个任选的终端符语句 *Termstmt*。
  假如省略了终端符语句 *Termstmt*，则动作语句 *Actstmt* 表就是非空的；假如动作语句 *Actstmt* 表是空的，则终端符语句 *Termstmt* 就一定要出现（它不需要被检验，因为它是一个上下文无关的具体语法规则）。
- 动作语句 *Actstmt* 含有一个任选标号 *Label* 和一个动作 *Act*。
- 标号 *Label* 是一个名字 *Name*。

80 ***Act*$_0$** = ***Task*$_0$** | ***Output*$_0$** | ***Create*$_0$** | ***Decision*$_0$** | ***Export*$_0$** | ***Option*$_0$** | ***Call*$_0$** | ***Prioutput*$_0$** | ***Set*$_0$** | ***Reset*$_0$**

- 动作 *Act* 可以是一个任务 *Task*、一个输出 *Output*、一个创建请求 *Create*、一个判定 *Decision*、一个出口 *Export*、一个跃迁任选 *Option*、一个过程调用 *Call*、一个优先级输出 *Prioutput*、一个定时器设置 *Set* 或是一个定时器复位 *Reset*。

81 ***Termstmt*$_0$** :: [***Label*$_0$**] ***Terminator*$_0$**
82 ***Terminator*$_0$** = ***Nextstate*$_0$** | ***Join*$_0$** | ***Stop*$_0$** | ***Return*$_0$**
83 ***Nextstate*$_0$** :: [***Statename*$_0$**]
84 ***Join*$_0$** :: ***Label*$_0$**
85 ***Stop*$_0$** :: ()
86 ***Return*$_0$** :: ()

- 终端符语句 *Termstmt* 含有一个任选标号 *Label* 和一个终端符 *Terminator*.
- 终端符 *Terminator* 可以是一个下一状态 *Nextstate*、一个汇接 *Join*、一个停止 *Stop* 或是一个返回 *Return*。
- 下一状态 *Nextstate* 含有一个任选状态名 *Statename*（如果下一状态是个划线 *dash*，则状态名是 **nil**）。
- 汇接 *Join* 含有一个标号 *Label*。
- 停止 *Stop* 不含附加信息。
- 返回 *Return* 不含附加信息。

87 ***Task*$_0$** :: ***Assignstmt*$_0$**$^+$ | ***Text*$_0$**$^+$

- 任务 *Task* 包含一个赋值语句 *Assignstmt* 表或是一个非形式正文 *Text* 表。

88 ***Create*$_0$** :: ***Prid*$_0$** ***Actparmlist*$_0$**
89 ***Actparmlist*$_0$** = [***Expr*$_0$**]*

- 创建请求 *Create* 包含一个进程标识符 *Prid* 和一个实际参数表 *Actparmlist*。
- 实际参数表 *Actparmlist* 是一个任选表达式 *Expr* 表。

90 ***Call*$_0$** :: ***Procid*$_0$** ***Actparmlist*$_0$**

- 过程调用 *Call* 包含一个过程标识符 *Procid* 和一个实际参数表 *Actparmlist*。

91 ***Output*$_0$** :: ***Outputsig*$_0$**$^+$ [***Piexpr*$_0$**] ***Via*$_0$**
92 ***Outputsig*$_0$** :: ***Sigid*$_0$** ***Actparmlist*$_0$**
93 ***Piexpr*$_0$** = ***Expr*$_0$** ¦ ***Scopeexpr*$_0$**
94 ***Via*$_0$** = ***Id*$_0$***
95 ***Scopeexpr*$_0$** :: ***Scope*$_0$** ***Expr*$_0$**
96 ***Scope*$_0$** = ***Qualifier*$_0$**

- 输出 *Output* 包含一个输出信号 *Outputsig* 表、一个任选的 *PId* 表达式 *Piexpr* 和一个经由 *Via*。
- 输出信号 *Outputsig* 由一个信号标识符 *Sigid* 和一个实际参数表 *Actparmlist* 组成。
- *PId* 表达式 *Piexpr* 可以是一个表达式 *Expr* 或是一个作用域表达式 *Scopeexpr*。
- 经由 *Via* 是一个可空的标识符 *Id* 表。

- 作用域表达式 *Scopeexpr* 在具体语法中没有相应的构件。它是在 AS_0 中引入的一个实用构件，目的是为了便于进行服务的变换。在进行服务变换时，它们被并入一个 AS_0 进程图中。除了含有输出和判定的跃迁是允许条件和连续信号所隐含的，在结果的 AS_0 进程图中的每一个跃迁必须在某一服务的范围中加以变换。在作用域表达式 *Scopeexpr* 中的作用域 *Scope* 是一个指示服务的限定符 *Qualifier*，该服务“拥有”出现在允许条件中判定内的表达式 *Expr*。
  各个作用域表达式 $Scopeexpr_0$ 都在扩展允许条件和连续信号的期间产生。

```
97  Decision0      :: Question0 Answer0+ [Elsepart0]
98  Question0      = Expr0 | Scopeexpr0
99  Elsepart0      :: [Transition0]
100 Answer0        :: Conditionlist0 [Transition0]
```

- 判定 *Decision* 含有一个问题 *Question*、一个由一些回答 *Answer* 组成的非空表和一个任选的 *Else* 部分 *Elsepart*。
- 问题 *Question* 可以是一个表达式 *Expr* 或是一个作用域表达式 *Scopeexpr*（如上所释）。
- *Else* 部分 *Elsepart* 含有一个任选的跃迁 *Transition*。
- 回答 *Answer* 包含一个条件表 *Conditionlist* 和一个任选的跃迁 *Transition*。

```
101 Timerdef0      :: Timerelem0+
102 Timerelem0     :: Timername0 Sortid0*
103 Timername0     = Name0
```

- 定时器定义 *Timerdef* 由一个定时器元素 *Timerelem* 表组成。
- 一个定时器元素 *Timerelem* 包含一个定时器名 *Timername* 和一个类别标识符 *Sortid* 表。
- 定时器名 *Timername* 是一个名字 *Name*。

```
104 Reset0         :: Resetelem0+
105 Resetelem0     :: Timerid0 Expr0*
106 Timerid0       = Id0
```

- 一个定时器复位 *Reset* 包含一个定时器复位元素 *Resetelem* 表。
- 一个定时器复位元素 *Resetelem* 包含一个定时器标识符 *Timerid* 和一个可空的表达式 *Expr* 表。
- 一个定时器标识符 *Timerid* 是一个标识符 *Id*。

```
107 Set0           :: Setelem0+
108 Setelem0       :: Expr0 Timerid0 Expr0*
```

- 定时器设置 *Set* 包含一个定时器设置元素 *Setelem* 表。
- 一个定时器设置元素 *Setelem* 包含一个定时到时表达式 *Expr*、一个定时器标识符 *Timerid* 和一个可空的表达式 *Expr* 表。

1.2 结构概念

1 ***Blocksub*$_0$** = ***Blocksubdef*$_0$** | ***Blocksubref*$_0$**
2 ***Blocksubdef*$_0$** :: [***Blocksubid*$_0$**] ***Blocksubdecl*$_0$**$^+$ [***Tailid*$_0$**]
3 ***Blocksubid*$_0$** = ***Id*$_0$**
4 ***Blocksubdecl*$_0$** = ***Signallistdef*$_0$** | ***Connect*$_0$** | ***Blockdef*$_0$** | ***Blockref*$_0$** | ***Chandef*$_0$** | ***Sigdef*$_0$** | ***Datadef*$_0$** | ***Select*$_0$**
5 ***Blocksubref*$_0$** :: ***Blocksubname*$_0$**
6 ***Blocksubname*$_0$** = ***Name*$_0$**

- 功能块子结构 *Blocksub* 可以是一个功能块子结构定义 *Blocksubdef* 或是一个功能块子结构引用 *Blocksubref*。
- 功能块子结构定义 *Blocksubdef* 含有一个功能块子结构标识符 *Blocksubid*、一个功能块子结构声明 *Blocksubdecl* 表和一个结束功能块子结构定义 *Blocksubdef* 的任选尾标识符 *Tailid*。
- 功能块子结构标识符 *Blocksubid* 是一个标识符 *Id*。
- 功能块子结构声明 *Blocksubdecl* 可以是一个信号表定义 *Signallistdef*、一个功能块子结构连接 *Connect*、一个功能块定义 *Blockdef*、一个功能块引用 *Blockref*、一个信道定义 *Chandef*、一个信号定义 *Sigdef*、一个数据定义 *Datadef* 或是一个选择 *Select*。
- 功能块子结构引用 *Blocksubref* 包含一个功能块子结构名 *Blocksubname*。
- 功能块子结构名 *Blocksubname* 是一个名字 *Name*。

7 ***Chansub*$_0$** = ***Chansubdef*$_0$** | ***Chansubref*$_0$**
8 ***Chansubdef*$_0$** :: ***Chansubid*$_0$** ***Chansubdecl*$_0$**$^+$ [***Tailid*$_0$**]
9 ***Chansubid*$_0$** = [***Id*$_0$**]
10 ***Chansubdecl*$_0$** = ***Chandef*$_0$** | ***Blockdef*$_0$** | ***Sigdef*$_0$** | ***Blockref*$_0$** ***Signallistdef*$_0$** | ***Datadef*$_0$** | ***Connect*$_0$** | ***Select*$_0$**
11 ***Chansubref*$_0$** :: ***Chansubname*$_0$**
12 ***Chansubname*$_0$** = ***Name*$_0$**

- 信道子结构 *Chansub* 可以是一个信道子结构定义 *Chansubdef* 或是一个信道子结构引用 *Chansubref*。
- 信道子结构定义 *Chansubdef* 由一个信道子结构标识符 *Chansubid*、一个信道子结构声明 *Chansubdecl* 表和一个结束信道子结构定义 *Chansubdef* 的任选尾标识符 *Tailid* 组成。
- 信道子结构标识符 *Chansubid* 是一个任选的标识符 *Id*。
- 信道子结构声明 *Chansubdecl* 可以是一个信道定义 *Chandef*、一个功能块定义 *Blockdef*、一个信号定义 *Sigdef*、一个功能块引用 *Blockref*、一个信号表定义 *Signallistdef*、一个数据定义 *Datadef*、一个信道端点连接 *Connect* 或是一个选择 *Select*。
- 一个信道子结构引用 *Chansubref* 含有一个信道子结构名 *Chansubname*。
- 信道子结构名 *Chansubname* 是一个名字 *Name*。

13 ***Refinement*$_0$** :: ***Subsignal*$_0$**$^+$
14 ***Subsignal*$_0$** :: [**REVERSE**] ***Sigdef*$_0$**

- 一个具体化部分 *Refinement* 含有一个子信号定义 *Subsignal* 表。
- 子信号定义 *Subsignal* 包含一个标志 **REVERSE** 用来指明该子信号是否指向相反方向，还包含一个信号定义 *Sigdef*。

1.3 补充概念

1 ***Select*$_0$** :: ***Expr*$_0$ *Decl*$_0$$^+$**

2 ***Decl*$_0$** = ***Blockdef*$_0$** | ***Blockref*$_0$** | ***Chandef*$_0$** | ***Sigdef*$_0$** | ***Prdef*$_0$** | ***Prref*$_0$** | ***Procdef*$_0$** | ***Procref*$_0$** | ***Servicedef*$_0$** | ***Serviceref*$_0$** | ***Datadef*$_0$** | ***Sigroutedef*$_0$** | ***Signallistdef*$_0$** | ***Vardef*$_0$** | ***Importdef*$_0$** | ***Viewdef*$_0$** | ***Timerdef*$_0$** | ***Connect*$_0$** | ***Select*$_0$**

- 选择定义 *Select* 由一个表达式 *Expr* 和一个声明 *Decl* 表组成。
- 一个声明 *Decl* 可以是一个功能块定义 *Blockdef*、一个功能块引用 *Blockref*、一个信道定义 *Chandef*、一个信号定义 *sigdef*、一个进程定义 *Prdef*、一个进程引用 *Prref*、一个过程定义 *Procdef*、一个过程引用 *Procref*、一个服务定义 *Servicedef*、一个服务引用 *Serviceref*、一个数据定义 *Datadef*、一个信号路由定义 *Sigroutedef*、一个信号表定义 *Signallistdef*、一个变量定义 *Vardef*、一个进口定义 *Importdef*、一个视见定义 *Viewdef*、一个定时器定义 *Timerdef*、一个连接 *Connect* 或是一个选择定义 *Select*。

3 ***Option*$_0$** :: ***Question*$_0$ *Answer*$_0$$^+$ [*Elsepart*$_0$]**

- 一个跃迁任选 *Option* 包含一个问题 *Question*、一个回答 *Answer* 表和一个任选的其它部分 *Elsepart*。

4 ***Decomposition*$_0$** :: ***Decompositiondecl*$_0$$^+$**

5 ***Decompositiondecl*$_0$** = ***Sigroutedef*$_0$** | ***Connect*$_0$** | ***Servicedef*$_0$** | ***Serviceref*$_0$** | ***Select*$_0$**

- 一个分解 *Decomposition* 包含一个分解声明 *Decompositiondecl* 表。
- 分解声明 *Decompositiondecl* 可以是一个信号路由定义 *Sigroutedef*、一个信号路由连接 *Connect*、一个服务定义 *Servicedef*、一个服务引用 *Serviceref* 或是一个选择 *Select* 定义。

6 ***Servicedef*$_0$** :: ***Serviceid*$_0$ [*Inputset*$_0$] *Servicedecl*$_0$* *Body*$_0$ [*Tailid*$_0$]**

7 ***Servicedecl*$_0$** = ***Vardef*$_0$** | ***Procdef*$_0$** | ***Procref*$_0$** | ***Viewdef*$_0$** | ***Importdef*$_0$** | ***Timerdef*$_0$** | ***Datadef*$_0$** | ***Select*$_0$**

8 ***Serviceid*$_0$** ⊆ ***Id*$_0$**

9 ***Serviceref*$_0$** :: ***Servicename*$_0$**

10 ***Servicename*$_0$** = ***Name*$_0$**

- 服务定义 *Servicedef* 由一个服务标识符 *Serviceid*、一个任选的有效服务输入信号集 *Inputset*、一个服务声明 *Servicedecl* 表、一个服务体 *Body* 和一个结束这个服务定义 *Servicedef* 的任选尾标识符 *Tailid* 组成。
- 服务声明 *Servicedecl* 可以是一个变量定义 *Vardef*、一个过程定义 *Procdef*、一个过程引用 *Procref*、一个视见定义 *Viewdef*、一个进口定义 *Importdef*、一个定时器定义 *Timerdef*、一个数据定义 *Datadef* 或是一个选择定义 *Select*。
- 服务标识符 *Serviceid* 是一个标识符 *Id*。
- 服务引用 *Serviceref* 包含一个服务名 *Servicename*。
- 服务名 *Servicename* 是一个名字 *Name*。

11 ***Priinput*$_0$** :: ***Inputvars*$_0$$^+$ *Transition*$_0$**

12 ***Prioutput*$_0$** :: ***Outputsig*$_0$$^+$**

- 优先级输入 *Priinput* 由一个信号输入变量 *Inputvars* 表和一个跃迁 *Transition* 组成。
- 优先级输出 *Prioutput* 包含一个输出信号 *Outputsig* 表。

13 *Contspec*$_0$ :: *Expr*$_0$ *Priority*$_0$ *Transition*$_0$
14 *Priority*$_0$ = [*Name*$_0$]
15 *Enabling*$_0$ = [*Expr*$_0$]

- 连续信号规格 *Contspec* 包含一个表达式 *Expr*、一个优先级 *Priority* 和一个跃迁 *Transition*。
- 优先级 *Priority* 是一个任选的整数名 *Name*。
- 允许 *Enabling* 是一个任选的表达式 *Expr*。

16 *Export*$_0$ :: *Varid*$_0$$^+$

- 出口 *Export* 包含一个出口变量标识符 *Varid* 表。

17 *Importdef*$_0$ :: *Importelem*$_0$$^+$
18 *Importelem*$_0$ :: *Varname*$_0$$^+$ *Sortid*$_0$
19 *Importexpr*$_0$ :: *Varid*$_0$ [*Expr*$_0$]

- 进口定义 *Importdef* 包含一个进口元素 *Importelem* 表。
- 进口元素 *Importelem* 是一个进口变量名 *Varname* 表和一个类别标识符 *Sortid*。
- 进口表达式 *Importexpr* 包含一个进口变量标识符 *Varid* 和一个任选的 PID 表达式 *Expr*。

1.4 数据

1 *Partialtypedef*$_0$ :: *Sortname*$_0$ [*Extproperties*$_0$] *Properties*$_0$ [*Conditionlist*$_0$] [*Tailname*$_0$]
2 *Sortname*$_0$ = *Name*$_0$

- 部分类型定义 *Partialtypedef* 由一个类别名 *Sortname*、一些任选的扩展性质 *Extproperties*、一些性质 *Properties*、一个任选的范围条件值表 *conditionlist* 和一个任选的结束这个定义的类别名 *Tailname* 组成。请注意，与 Z.100 中的语法规则相反，这个定义也包括了隐含了部分类型定义的同义类型（即在 *Conditionlist*$_0$ 不是 **nil** 的情况下）。
- 类别名 *Sortname* 是一个名字 *Name*。

3 *Properties*$_0$ :: *Literal*$_0$* *Op*$_0$* *Axiom*$_0$* *Mappingaxiom*$_0$* [*Initialvalue*$_0$]

- 性质 *Properties* 由一个字面值标记 *literal* 表、一个运算符标记 *Op* 表、一个公理 *Axiom* 表、一个字面值映射公理 *Mappingaxiom* 表和一个任选的初始（缺省）值 *value* 组成。

4 *Literal*$_0$ = *Name*$_0$ | *String*$_0$ | *Nmclass*$_0$

- 字面值标记 *Literal* 可以是一个名字 *Name*、一个字符串 *String* 或是一个名字类 *Nmclass*。

5 *Op*$_0$ = *Ordering*$_0$ | *Opspec*$_0$
6 *Opspec*$_0$ :: *Operatorname*$_0$ *Sortid*$_0$$^+$ *Sortid*
7 *Operatorname*$_0$ = *Name*$_0$ | *Quotedop*$_0$

- 运算符标记 *Op* 可以是序符 *Ordering* 或是一个运算符规格 *Opspec*。
- 运算符规格 *Opspec* 由一个运算符名 *Operatorname*、一个变量类别标识符 *Sortid* 表和一个结果类别标识符 *Sortid* 组成。
- 运算符名 *Operatorname* 是一个名字 *Name* 或是一个引用文运算符 *Quotedop*。

8 $Axiom_0$ = $Unquantequation_0$ | $Condequation_0$ | $Quantequation_0$
9 $Unquantequation_0$ = $Equation_0$ | $Term_0$

- 公理 *Axiom* 可以是一个非量化的等式 *Unquantequation*、一个条件等式 *Condequation*，一个量化等式 *Quant equation* 或是一个项 *Term*。
- 非量化等式 *Unquantequation* 可以是一个等式 *Equation* 或是一个项 *Term*。

10 $Equation_0$ :: $Term_0$ $Term_0$

- 等式 *Equation* 由一个左边项 *Term* 和一个右边项 *Term* 组成。

11 $Quantequation_0$ :: $Valuename_0^+$ $Sortid_0$ $Axiom_0^+$
12 $Valuename_0$ = $Name_0$

- 量化等式 *Quantequation* 由一个值名 *Valuename* 表、一个类别标识符 *Sortid* 和一个公理 *Axiom* 表组成。
- 值名 *Valuename* 是一个名字 *Name*。

13 $Term_0$ = Id_0 | $Operatorterm_0$ | $Condterm_0$ | $Stringterm_0$
$Monadterm_0$ | $Infixterm_0$ | $Errorterm_0$ | $Spellingterm_0$

- 项 *Term* 可以是一个标识符 *Id*、一个运算符项 *Operatorterm*、一个条件项 *Condterm*、一个字符串项 *Stringterm*、一个一元项 *Monadterm*、一个中缀项 *Infixterm*、一个错误项 *Errorterm* 或是一个拼字项 *Spellingterm*。

  注意，在 AS_0 中，项内和表达式内的各种运算符在语法上没有区别，它们仅在语义上不同。例如，在 Z.100 的 § 5.4 中所定义的〈扩展合成项〉和〈扩展基项〉在语法上没有区别（即没有把名字与定义连系起来），这意味着在 AS_0 中它们被相同的域定义所包括。

14 $Operatorterm_0$ :: (Id_0 | $Qualop_0$) $Term_0^+$

- 运算符项 *Operatorterm* 由一个标识符 *Id* 或一个被限定的运算符 *Qualop* 并后随一个变元项 *Term* 表组成。

15 $Condterm_0$ :: $Term_0$ $Term_0$ $Term_0$

- 条件项 *Condterm* 由一个条件项 *Term*、一个后继项 *Term* 和一个替换项 *Term* 组成。

16 $Stringterm_0$ :: $Qualifier_0$ $String_0$

- 字符串项 *Stringterm* 由一个限定词 *Qualifier* 和一个字符串 *String* 组成。

17 *Monadterm*$_0$ :: (**NOT** | **MINUS**) *Term*$_0$

- 一元项 *Monadterm* 由一个运算符 "**NOT**" 或 "-" 后随一个变元项 *Term* 组成。

18 *Infixterm*$_0$ :: *Term*$_0$ *Infixop*$_0$ *Term*$_0$
19 *Infixop*$_0$ = **IMPLY** | **OR** | **XOR** | **AND** | **IN** | **MOD** | **REM** | **PLUS** | **MINUS** | **CONC** | **MULT** | **DIV** | *Relop*$_0$
20 *Relop*$_0$ = **NE** | **EQ** | **GT** | **LT** | **LE** | **GE**

- 中缀项 *Infixterm* 由两个变元项 *Term* 和一个中缀运算符 *Infixop* 组成。
- 中缀运算符 *Infixop* 可以是 "⇒"、"**OR**"、"**XOR**"、"**AND**"、"**IN**"、"**MOD**"、"**REM**"、"+"、"−"、"//"、"*"、"/" 或是一个关系运算符 *Relop*。
- 关系运算符 *Relop* 可以是 "/="、"="、">"、"<"、"<=" 或是 ">="。

注意，与 Z.100 的 § 5.4.2.2 中定义的一样，任何子树的分组都反映优先规则。

21 *Errorterm*$_0$ :: ()

- 错误项 *Errorterm* 不包含附加信息。

22 *Condequation*$_0$ :: *Unquantequation*$_0$$^+$ *Unquantequation*$_0$

- 条件等式 *Condequation* 由一个限制的非量化等式 *Unquantequation* 表和一个被限制的非量化等式 *Unquantequation* 组成。

23 *Extproperties*$_0$ = *Struc*$_0$ | *Inherited*$_0$ | *Geninstlist*$_0$

- 扩展性质 *Extproperties* 可以是结构性质 *Struc*、继承性质 *Inherited* 或是生成程序实例表 *Geninstlist* 性质。

24 *Syntypedef*$_0$ :: *Syntypename*$_0$ *Parentid*$_0$ [*Initialvalue*$_0$] [*Conditionlist*$_0$] [*Tailname*$_0$]
25 *Syntypename*$_0$ = *Name*$_0$

- 同义类型定义 *Syntypedef* 由一个同义类型名 *Syntypename*、一个父辈标识符 *Parentid*、一个任选的初始值 *Initialvalue*、一个任选的范围条件表 *Conditionlist* 和一个结束这个定义的同义类型名 *Tailname* 组成。
- 同义类型名 *Syntypename* 是一个名字 *Name*。

26 *Conditionlist*$_0$ = *Condition*$_0$$^+$
27 *Condition*$_0$ :: [*Value*$_0$ | *Relop*$_0$] *Value*$_0$
28 *Value*$_0$ = *Expr*$_0$

- 范围条件表 *Conditionlist* 是一连串的范围条件 *Condition*。
- 范围条件 *Condition* 由一个任选值 *Value* 或关系运算符 *Relop* 后随一个值 *Value* 组成。
- 值 *Value* 是一个表达式 *Expr*。

29 *Struc*$_0$ :: *Fieldspec*$_0$$^+$

30 *Fieldspec*$_0$ :: *Fieldname*$_0$$^+$ *Sortid*$_0$

31 *Fieldname*$_0$ = *Name*$_0$

- 结构性质 *Struc* 由一连串的字段规格 *Fieldspec* 组成。
- 字段规格 *Fieldspec* 包含一连串字段名字 *Fieldname* 和一个类别标识符 *Sortid*。
- 字段名 *Fieldname* 是一个名字 *Name*。

32 *Inherited*$_0$ :: *Parentid*$_0$ *Literalrenaming*$_0$ (ALL | *Operatorrenaming*$_0$)

33 *Parentid*$_0$ = *Id*$_0$

34 *Literalrenaming*$_0$ = *Literalpair*$_0$*

35 *Literalpair*$_0$ :: *Newliteral*$_0$ *Oldliteral*$_0$

36 *Newliteral*$_0$ = *Name*$_0$ | *String*$_0$

37 *Oldliteral*$_0$ = *Name*$_0$ | *String*$_0$

38 *Operatorrenaming*$_0$ = *Operatorpair*$_0$*

39 *Operatorpair*$_0$ :: *Newoperator*$_0$ *Oldoperator*$_0$

40 *Newoperator*$_0$ = [*Operatorname*$_0$]

41 *Oldoperator*$_0$ = *Operatorname*$_0$

- 继承性质 *Inherited* 由一个父辈标识符 *Parentid*、字面值重新命名 *Literalrenaming* 和继承运算符组成，继承运算符可以是 ALL 运算符，也可以是运算符重新命名 *Operatorrenaming*。
- 父辈标识符 *Parentid* 是一个标识符 *Id*。
- 字面值重新命名 *Literalrenaming* 是一个字面值偶对 *Literalpair* 表。
- 一个字面值偶对 *Literalpair* 由新字面值名 *Newliteral* 和旧（父辈）字面值名 *Oldliteral* 组成。
- 新字面值名 *Newliteral* 可以是一个名字 *Name* 或是一个字符串 *String*。
- 旧字面值名 *Oldliteral* 可以是一个名字 *Name* 或是一个字符串 *String*。
- 运算符重新命名 *Operatorrenaming* 是一个运算符偶对 *Operatorpair* 表。
- 运算符偶对 *Operatorpair* 由新运算符名 *Newoperator* 和旧运算符名 *Oldoperator* 组成。
- 如果规定了新运算符名 *Newoperator*，则它是一个运算符名 *Operatorname*。
- 旧的（父辈）运算符名 *Oldoperator* 是一个运算符名 *Operatorname*。

42 *Sortgenerator*$_0$ :: *Generatorname*$_0$ *Genparm*$_0$$^+$
[*Geninstlist*$_0$] *Properties*$_0$ [*Tailname*$_0$]

43 *Generatorname*$_0$ = *Name*$_0$

44 *Genparm*$_0$ = *Sortparm*$_0$ | *Termparm*$_0$ | *Litparm*$_0$ | *Opparm*$_0$

45 *Sortparm*$_0$ :: *Name*$_0$$^+$

46 *Termparm*$_0$ :: *Name*$_0$$^+$

47 *Litparm*$_0$ :: *Name*$_0$$^+$

48 *Opparm*$_0$ :: *Name*$_0$$^+$

- 类别生成程序 *Sortgenerator* 由一个生成程序名 *Generatorname*、一个生成程序形式参数 *Genparm* 表、一个可空的生成程序实例表 *Geninstlist*、一些性质 *Properties* 和一个结束这个定义的类别生成程序名 *Tailname* 组成。
- 生成程序名 *Generatorname* 是一个名字 *Name*。
- 生成程序形式参数 *Genparm* 可以是一个类别参数 *Sortparm*、一个项参数 *Termparm*、一个字面值参数 *Litparm* 或一个运算符参数 *Opparm*。
- 类别参数 *Sortparm* 由一个名字 *Name* 表组成。

- 项参数 *Termparm* 由一个名字 *Name* 表构成。
- 字面值参数 *Litparm* 由一个名字 *Name* 表构成。
- 运算符参数 *Opparm* 由一个名字 *Name* 表构成。

```
49  Geninstlist0        :: Geninst0+
50  Geninst0            :: Generatorid0 Genactparm0+
51  Generatorid0        =  Id0
52  Genactparm0         =  Term0 | Quotedop0 | Nmclass0
```

- 生成程序实例表 *Geninstlist* 由一连串的生成程序实例 *Geninst* 组成。
- 生成程序实例 *Geninst* 由一个生成程序标识符 *Generatorid* 和一个生成程序实际参数 *Genactparm* 表组成。
- 生成程序标识符 *Generatorid* 是一个标识符 *Id*。
- 生成程序实际参数 *Genactparm* 是一个项 *Term* 或是一个引用文运算符 *Quotedop*，或是一个名字类 *Nmclass*。即象其它种类的实际参数一样，类别标识符与运算符名在语法上是项 *Term*。另一方面，引用文运算符和名字类自己不能形成一个项，因此被明确地规定为生成程序实际参数 $Genactparm_0$ 中的替换成分。

```
53  Synonymdef0         :: Synonymname0 [Sortid0] [Initialvalue0]
54  Synonymname0        =  Name0
```

- 同义词定义 *Synonymdef* 由一个同义词名 *Synonymname*、一个任选的类别标识符 *Sortid* 和一个任选的初始（缺省）值 *Initialvalue* 组成。
- 同义词名 *Synonymname* 是一个名字 *Name*。

```
55  Nmclass0            :: Regularexp0
56  Regularexp0         =  Partregexp0 | Orregexp0 | Andregexp0
57  Orregexp0           :: Regularexp0 Partregexp0
58  Andregexp0          :: Regularexp0 Partregexp0
59  Partregexp0         =  Rngregexp0 | Singregexp0 | Parenregexp0
60  Rngregexp0          :: String0 String0 [Regexpexp0]
61  Singregexp0         :: String0 [Regexpexp0]
62  Parenregexp0        :: Regularexp0 [Regexpexp0]
63  Regexpexp0          =  MULT | PLUS | Name0
```

- 名字类 *Nmclass* 由一个正规式 *Regularexp* 构成。
- 正规式 *Regularexp* 可以是一个部分正规式 *Partregexp*、一个中缀 *Or* 正规式 *Orregexp* 或是一个中缀 *And* 正规式 *Andregexp*。
- 中缀 *Or* 正规式 *Orregexp* 由一个正规式 *Regularexp* 和一个部分正规式 *Partregexp* 组成。
- 中缀 *And* 正规式 *Andregexp* 由一个正规式 *Regularexp* 和一个部分正规式 *Partregexp* 组成。
- 部分正规式 *Partregexp* 可以是一个范围正规式 *Rngregexp*，一个简单正规式 *Singregexp* 或是一个括号正规式 *Parenregexp*。
- 范围正规式 *Rngregexp* 由两个字符串 *String* 和一个任选的重复的正规式表达式 *Regexpexp* 组成。
- 简单正规式 *Singregexp* 由一个字符串 *String* 和一个任选的重复的正规式表达式 *Regexpexp* 组成。

- 括号正规式 *Parenregexp* 由一个正规式 *Regularexpr* 和一个任选的重复的正规式表达式 *Regexpexp* 组成。
- 重复的正规式表达式 *Regexpexp* 可以是一个“*”、一个“+”或是名字 *Name*。

64 $Ordering_0$ :: ()

- 序符 *Ordering* 不含附加信息。

65 $Mappingaxiom_0$ :: $Valuename_0^+$ $Sortid_0$ $Literalaxiom_0^+$
66 $Literalaxiom_0$ = $Axiom_0$ | $Mappingaxiom_0$
67 $Spellingterm_0$ :: Id_0

- 映射公理 *Mappingaxiom* 由一个值名 *Valuename* 表、一个类别标识符 *Sortid* 和一个字面值公理 *Literalaxiom* 表组成。
- 字面值公理 *Literalaxiom* 可以是一个公理 *Axiom* 或是一个映射公理 *Mappingaxiom*。
- 拼字项 *Spellingterm* 含有一个标识符 *Id*。

68 $Expr_0$ = Id_0 | $Stringterm_0$ | $Condexpr_0$ |
$Operatorapp_0$ | $Monadexpr_0$ | $Infixexpr_0$ |
$Impoperator_0$ | $Selectexpr_0$ | $Tupleexpr_0$

- 表达式 *Expr* 可以是一个标识符 *Id*、一个字符串项 *Stringterm*、一个条件表达式 *Condexpr*、一个运算符应用 *Operatorapp*、一个一元表达式 *Monadexpr*、一个中缀表达式 *Infixexpr*、一个命令运算符 *Impoperator*、一个字段选择表达式 *Select-expr* 或是一个结构元组表达式 *Tupleexpr*。

69 $Monadexpr_0$ :: (NOT | MINUS) $Expr_0$

- 一元表达式 *Monadexpr* 由一个运算符“NOT”或“−”后随一个变量表达式 *Expr* 组成。

70 $Infixexpr_0$ :: $Expr_0$ $Infixop_0$ $Expr_0$

- 中缀表达式 *Infixexpr* 由两个变元表达式 *Expr* 和一个中缀运算符 *Infixop* 组成。

71 $Selectexpr_0$ :: $Expr_0$ $Name_0$

- 字段选择表达式 *Selectexpr* 由一个表达式 *Expr* 和一个字段名 *Name* 组成。

72 $Tupleexpr_0$ :: $Qualifier_0$ $Expr_0^+$

- 元组表达式 *Tupleexpr* 包含一个限定符 *Qualifier* 和一个表达式 *Expr* 表。

73 $Condexpr_0$ :: $Expr_0$ $Expr_0$ $Expr_0$

- 条件表达式 *Condexpr* 由一个条件表达式 *Expr*、一个后继表达式 *Expr* 和一个替换表达式 *Expr* 组成。

74 $Datadef_0$ = $Partialtypedef_0$ | $Syntypedef_0$ | $Sortgenerator_0$ | $Synonymdef_0$

- 数据定义 *Datadef* 可以是一个部分类型定义 *Partialtypedef*、一个同义类型定义 *Syntypedef*、一个类别生成程序 *Sortgenerator* 或是一个同义词定义 *Synonymdef*。

75 ***Exprlist*$_0$** = ***Expr*$_0$***

- 表达式表 *Exprlist* 是一连串的表达式 *Expr*。

76 ***Operatorapp*$_0$** :: (***Expr*$_0$** | ***Qualop*$_0$**) ***Exprlist*$_0$**
77 ***Qualop*$_0$** :: ***Qualifier*$_0$** ***Quotedop*$_0$**
78 ***Quotedop*$_0$** :: ***Infixop*$_0$** | **NOT**

- 一个运算符应用 *Operatorapp* 包含一个表达式 *Expr* 或一个限定运算符 *Qualop* 后随一个表达式表 *Exprlist*。在运算符应用 *Operatorapp* 里的表达式可以是一个标识符 *Id*（表示一个运算符的标识符），或者象 Z.100 的 § 5.4.2.4 和 § 5.4.2.5 中所定义的那样，表示一个初等项 〈primary〉。
- 限定运算符 *Qualop* 包含一个限定符 *Qualifier* 和一个引用文运算符 *Quotedop*。
- 引用文运算符 *Quotedop* 是一个中缀运算符 *Infixop* 或 **"NOT"**。

79 ***Assignstmt*$_0$** :: ***Variable*$_0$** ***Expr*$_0$**
80 ***Variable*$_0$** = ***Varid*$_0$** | ***Indexedvar*$_0$** | ***Fieldvar*$_0$**
81 ***Indexedvar*$_0$** :: ***Variable*$_0$** ***Exprlist*$_0$**
82 ***Fieldvar*$_0$** :: ***Variable*$_0$** ***Name*$_0$**

- 赋值语句 *Assignstmt* 包含一个变量 *Variable* 和一个表达式 *Expr*。
- 变量 *Variable* 可以是一个变量标识符 *Varid*、一个被标引的变量 *Indexedvar* 或是一个字段变量 *Fieldvar*。
- 被标引的变量 *Indexedvar* 由一个变量 *Variable* 和一个表达式表 *Exprlist* 组成。请注意，被标引的变量 *Indexedvar* 可以表示一个字段变量 *Fieldvar*，例如

  `v(a) := ...`

  可以（依赖于上下文）是

  另一种书写方法

  `v! a := ...`
- 字段变量 *Fieldvar* 包含一个变量 *Variable* 和一个字段名 *Name*。

83 ***Viewexpr*$_0$** :: ***Varid*$_0$** ***Expr*$_0$**

- 视见表达式 *Viewexpr* 由一个视见变量标识符 *Varid* 和一个表达式 *Expr* 组成。

84 ***Initialvalue*$_0$** = ***Expr*$_0$**

- 初始（缺省）值 *Initialvalue* 是一个表达式 *Expr*。

85 ***Impoperator*$_0$** = ***Importexpr*$_0$** | ***Viewexpr*$_0$** | ***Nowexpr*$_0$** | ***Activeexpr*$_0$** | ***Parentexpr*$_0$** | ***Offspringexpr*$_0$** | ***Senderexpr*$_0$** | ***Selfexpr*$_0$**
86 ***Nowexpr*$_0$** :: ()
87 ***Selfexpr*$_0$** :: ()
88 ***Parentexpr*$_0$** :: ()
89 ***Offspringexpr*$_0$** :: ()
90 ***Senderexpr*$_0$** :: ()
91 ***Activeexpr*$_0$** :: ***Timerid*$_0$** ***Exprlist*$_0$**

- 命令运算符 *Impoperator* 可以是一个进口表达式 *Importexpr*、一个视见表达式 *Viewexpr*、一个现在表达式 *Nowexpr*、一个定时器活跃表达式 *Activeexpr*、一个父辈表达式 *Parentexpr*、一个后代表达式 *Offspringexpr*、一个发送者表达式 *Senderexpr* 或是一个自身表达式 *Selfexpr*。
- 现在表达式 *Nowexpr* 不含附加信息。
- 自身表达式 *Selfexpr* 不含附加信息。
- 父辈表达式 *Parentexpr* 不含附加信息。
- 后代表达式 *Offspringexpr* 不含附加信息。
- 发送者表达式 *Senderexpr* 不含附加信息。
- 定时器活跃表达式 *Activeexpr* 包含一个定时器标识符 *Timerid* 和一个表达式表 *Exprlist*。

2 内部域

语义域对组合实物的域作出定义。组合实物域拥有变换期间所需的一些导出（上下文）信息（属性）。有两种不同的信息。一种是附属于指定实体的信息，例如有关信道端点、变量类别、信号类别表等信息。另一种是“公用”数据，例如哪一个作用域单位包围一个定义表、隐式变量名的集合等信息。这两种信息（描述字典 *Descriptordict* 和引用文字典 *Quotdict*）汇集为一个实物字典 *Dict*，使得当函数使用它或生成它时分别仅需要一个额外的形式参数或一个额外的结果。

1 ***Dict*** = ***Descriptordict*** ∪ ***Quotdict***

作为形式参数给出的字典 *Dict* 实物习惯上命名为 *dict* 并且在函数首部第二个变量表中出现。

下面详细地描述这两个域。

2.1 描述字典 *Descriptordict* 的说明

描述字典 *Descriptordict* 是一个映射表，它把标识符映射到它们的描述符。

1 *Descriptordict* = *Qual* $\overrightarrow{m}$ *Descr*
2 *Qual* = (*Qualelem* | *Operatorqualelem* | *Importqualelem* | *Viewqualelem*)$^+$

限定表 *Qual* 指 AS_0 标识符的内部表示法。在 SDL 中，只有在同一实体类中的 AS_0 标识符才是唯一的；为了把所有标识符编入同一个映射表中，该实体类必须是这个映射表中的一部分项目。因此，限定表 *Qual* 是由一些限定元素组成的一个表，表中的最后那个限定元素就是该实体类和该实体名。运算符限定元素 *Operatorqualelem*、进口限定元素 *Importqualelem* 和视见限定元素 *Viewqualelem* 要用专门的方法来处理，因为需要附加信息以保证那些实体的唯一性。

例：

某系统 SYS 中的功能块 BL 在描述字典 *Descriptordict* 中有一个限定表 *Qual*，它是

⟨(**SYSTEM**, **mk-***Name*$_0$(**"SYS"**, **nil**)), (**BLOCK**, **mk-***Name*$_0$(**"BL"**, **nil**))⟩

功能块 BL 中的信号 SIG 有一个限定表 *Qual*，它是

⟨(**SYSTEM**, **mk-***Name*$_0$(**"SYS"**, **nil**)), (**BLOCK**, **mk-***Name*$_0$(**"BL"**, **nil**)), (**SIGNAL**, **mk-***Name*$_0$(**"SIG"**, **nil**))⟩

3 *Qualelem* = *Kind* (*Name*$_0$ | *String*$_0$)
4 *Kind* = *Scopeunit*$_0$ | *Entity*
5 *Entity* = **SIGNALROUTE** | **CHANNEL** | **SIGNALLIST** | **GENERATOR** | **VALUE** | **LITERAL**
6 *Operatorqualelem* = **OPERATOR** ((*Name*$_0$ | *Quotedop*$_0$) *Sortqual*$^+$ *Sortqual*)
7 *Importqualelem* = **IMPORT** (*Name*$_0$ *Sortqual*)
8 *Viewqualelem* = **VIEW** *Qual*

一个限定元素 *Qualelem* 是由作用域单位的类型 *Scopeunit* 和名字组成的偶对。当它在表中（即 *Qual* 中）作为最后的限定元素 *Qualelem* 出现时，它表示本实体类和本实体名。是一个实体 *Entity* 的种类 *Kind* 总是表示一个实体类，也就是说，它总是在 *Qual* 中作为最后的限定元素 *Qualelem* 出现。对于特殊的限定元素 *Operatorqualelem*、*Importqualelem* 和 *Viewqualelem* 就总是这种情况。实体类 **VALUE** 表示变量、同义词和值标识符。

在“部分数据类型”作用域单位内使运算符唯一的信息（即运算符限定元素 *Operatorqualelem*）是名字 *Name*$_0$ 或引用文运算符 *Quotedop*$_0$，一些变元类别 *Sortqual*$^+$ 和结果类别 *Sortqual*。

在一作用域单位内使被进口实体唯一的信息（即进口限定元素 *Importqualelem*）是被进口实体的名字 *Name*$_0$ 和它的类别 *Sortqual*。

在一个进程定义中使一被视见变量唯一的信息（即视见限定元素 *Viewqualelem*）是这个被视见变量的标识符 *Qual*。

9 *Descr* = *SystemD* | *BlockD* | *ChannelD* | *SignalD* | *TimerD* | *SignalrouteD* | *SignallistD* | *ProcedureD* | *ProcessD* | *SortD* | *SyntypeD* | *GeneratorD* | *SynD* | *VarD* | *ImportD* | *ViewD* | *LiteralD* | *OperatorD* | *ValueidD* |

ServiceD | *BlocksubD* | *ChannelsubD* | *ErrorD*

描述字典 *Descriptordict* 中的描述符 *Descr* 可以是一个系统(或最外层次)的描述符、或一个功能块描述符、一个信道描述符、一个信号描述符、一个定时器描述符、一个信号路由描述符、一个信号表描述符、一个过程描述符、一个进程描述符、一个类别描述符、一个同义类型描述符、一个生成程序描述符、一个同义词描述符、一个变量描述符、一个进口变量描述符、一个视见变量描述符、一个字面值描述符、一个运算符描述符、一个公理变量（值标识符）描述符、一个服务描述符、一个功能块子结构描述符、一个信道子结构描述符或一个递归描述符。

关于限定表 *Qual* 和在字典 *Dict* 中与它们相关的描述符 *Descr* 之间的关系，总是存在着下面的 Meta-IV 的断言：

is-consistent-Dict(*dict*) $\triangleq$

```
1  (∀qual ∈ dom dict)
2   ((let (kind, ) = qual[len qual] in
3     cases kind:
4      (SYSTEM
5        → is-SystemD(dict(qual)),
6       BLOCK
7        → is-BlockD(dict(qual)),
8       CHANNEL
9        → is-ChannelD(dict(qual)),
10      SIGNAL
11       → is-SignalD(dict(qual)) ∨ is-TimerD(dict(qual)),
12      SIGNALROUTE
13       → is-SignalrouteD(dict(qual)),
14      SIGNALLIST
15       → is-SignallistD(dict(qual)) ∨ is-ErrorD(dict(qual)),
16      PROCEDURE
17       → is-ProcedureD(dict(qual)),
18      PROCESS
19       → is-ProcessD(dict(qual)),
20      TYPE
21       → is-SortD(dict(qual)) ∨ is-SyntypeD(dict(qual)),
22      VALUE
23       → is-VarD(dict(qual)) ∨ is-SynD(dict(qual)) ∨
24         is-ValueidD(dict(qual)) ∨ is-ErrorD(dict(qual)),
25      GENERATOR
26       → is-GeneratorD(dict(qual)) ∨ is-ErrorD(dict(qual)),
27      SUBSTRUCTURE
28       → is-BlocksubD(dict(qual)) ∨ is-ChannelsubD(dict(qual)),
29      LITERAL
30       → is-LiteralD(dict(qual)),
31      SERVICE
32       → is-ServiceD(dict(qual)),
33      OPERATOR
34       → is-OperatorD(dict(qual)),
35      IMPORT
36       → is-ImportD(dict(qual)),
37      VIEW
38       → is-ViewD(dict(qual)))))
```

type: *Dict* → *Bool*

我们规定上面的函数仅仅是为了用于解释。

10 ***SystemD*** :: ()
11 ***BlocksubD*** :: ()

系统描述 *SystemD* 是一个系统层的描述符。

功能块子结构描述 *BlocksubD* 是一个功能块子结构的描述符。提供这些描述符是由于系统层和功能块子结构是定义作用域单位的实体。因此，这些名字被用于标识符中的限定词内。(在字典 *Dict* 的域中可以找到作为作用域单位的所有被完全规定的限定词)

12 ***ChannelsubD*** :: ***Blockqual***

信道子结构描述 *ChannelsubD* 是一个信道子结构的描述符。

Blockqual 在 AS_1 中，信道子结构被表示为一个综合功能块定义的功能块子结构。*Blockqual* 表示这个功能块子结构的标识符。每当使用一个 AS_0 标识符在它的限定部分中引用信道子结构时，该限定词就按照 *Blockqual* 来修改。

13 ***ServiceD*** :: ***Transition$_0$ Statedict Labeldict Validinputset Priinputset***
14 ***Priinputset*** = ***Signalqual-set***

服务描述符 *ServiceD* 是一个 SDL 服务的描述符。

Transition$_0$ 是取自于服务定义体 *Body* 的初始跃迁。

Statedict，*Labeldict* 是用于此服务的 *Statedict* 和 *Labeldict*（见 *Quotdict* 的描述）。

Validinputset 此服务的全部有效的输入信号集合（包括定时器）。

Priinputset 此服务的优先输入信号集合。

15 ***BlockD*** :: ***Exportchannels Importchannels Explicitroutes BlockconnectionD***
16 ***Exportchannels*** = ***ExpimpchanD***
17 ***Importchannels*** = ***ExpimpchanD***
18 ***ExpimpchanD*** = ***NameclosureD*** $\overrightarrow{m}$(***Otherend Channame$_0$***)*
19 ***Otherend*** = ***Blockqual*** | **ENV**
20 ***Explicitroutes*** = ***Bool***
21 ***BlockconnectionD*** = ***Channelqual*** $\overrightarrow{m}$***Qual-set***
22 ***Channelqual*** = ***Qual***

功能块描述 *BlockD* 是一个功能块描述符。

Exportchannels 包含关于进入和走出功能块的隐式信道的信息，这是由于包含在功能块里进程中的出口变量引起的。

Importchannels 包含进入和走出功能块的隐式信道信息，这是由于包含在功能块里进程中的进口变量引起的。对于进口信道 *importchannel* 中所含有的隐式信道，其信息可从在其它功能块描述符中出现的出口信道 *Exportchannels* 中推导出来。

ExpimpchanD 是一个映射表，它把进—出口闭包（参见名字闭包描述 *Name-closureD* 的域定义）映射到信道另一端点（*Otherend*）和附加于本闭包的双向信道。

信道里的第一个信号表包含信号 xtQUERY，信道里的第二个信号表包含信号 xtREPLY。

Explicitroutes 将为真，如果为功能块规定了显式信号路由。*Explicitroutes* 用于为所包含的进程导出全部有效的输入信号集。

BlockconnectionD 是信道与连接到这个信道上的信号路由之间的关系。用所包含的进程的 VIA 构件中的信号路由标识符来替代信道标识符时要使用 *BlockconnectionD*。另外，用连接中合适的新信道标识符来替代拥有一个子结构的信道标识符时，也要使用 *BlockconnectionD*。

Channelqual 是一个信道的限定表 *Qual*。

23 ***ChannelD*** :: ***Endpoint Endpoint Signalqual-set Signalqual-set*** [***Newchannels***]
24 ***Endpoint*** = ***Blockqual*** | **ENV**
25 ***Signalqual*** = ***Qual***
26 ***Blockqual*** = ***Qual***
27 ***Newchannels*** = ***Endpoint Channame$_0$ Endpoint Channame$_0$***

信道描述 *ChannelD* 是一个信道的描述符。

Endpoint 是一个端点，这个端点可以是一个功能块标识符或是环境。

Signalqual-**set** 是从起点 *Origin* 传送到目的地 *Destination* 的所有信号标识符的集合。对于双向信道，为了表示从第二个 *Endpoint*（端点）功能块传送到第一个（*Endpoint*）端点功能块的信号，还要提供另一个信号集。

Newchannels 如果信道包含一个信道子结构，这个信道就由 AS_1 中的两个信道来表示。新信道 *Newchannels* 包含这两个信道的名字和起始端点，当一个旧的信道标识符要被一个 VIA 集合中的新的信道标识符替换时，将要使用这些信息。

28 ***SignalD*** :: ***Sortqual* Signalqual-set Signalqual-set***
29 ***Sortqual*** = ***Qual***

信号描述 *SignalD* 是一个信号的描述符。

*Sortqual** 是由信号传送的值的类别。

Signalqual-**set** 子信号的两个集合，第一个集合是与父辈信号方向相同的信号集合，第二个集合是指向相反方向的信号集合。

30 ***TimerD*** :: ***Sortqual* Newqual***

定时器描述 *TimerD* 是一个定时器的描述符。

*Sortqual** 是由定时器传送的值的类别。

Newqual 表示 AS_1 所使用的标识符。如果定时器是在一个服务中定义的，则它在新限定表 *Newqual* 中的名字已经改变了。

31 *SignalrouteD* :: *Originprocess Destinationprocess Signalqual-set Signalqual-set*
32 *Originprocess* = *Processqual* | **ENV**
33 *Destinationprocess* = *Processqual* | **ENV**

信号路由描述 *SignalrouteD* 是一个信号路由的描述符。

Originprocess 是起始端点，它可以是一个进程标识符或是环境。

Destinationprocess 是终止端点，它可以是一个进程标识符或是环境。

Signalqual-**set** 是从起点 *Origin* 传送到目的地 *Destination* 的信号标识符集合。对于双向信号路由，为了表示从目的地进程传送到起始点进程的信号，要提供另一个信号集合。

34 *SignallistD* :: $Signallist_0$

信号表描述 *SignallistD* 是一个信号表描述符。它包含附属于信号表的一连串 AS_0 信号标识符（和信号表标识符）。

35 *ProcedureD* :: *FormparmD* Newqual*
36 *FormparmD* = *InDescr* | *InoutDescr*
37 *InDescr* :: *Sortqual*
38 *InoutDescr* :: *Sortqual*
39 *Newqual* = *Qual*

过程描述 *ProcedureD* 是一个过程描述符。

*FormparmD** 在引用过程时所用的一个专用的描述符表，用于检查实际参数的类别是否与形式参数一致；即，这些描述符包含相对应的形式参数的性质。

InDescr 一个 **IN** 变量参数

InoutDescr 一个 **IN/OUT** 变量参数

Newqual 表示在 AS_1 中要用到的标识符。如果在服务中定义了这个过程，在 *Newqual* 中，它的名字已改变。

40 *ProcessD* :: *ParameterD* Validinputset Outputset ProcessconnectionD*
41 *Validinputset* = *Signalqual-set*
42 *Outputset* = *Signalqual-set*
43 *ParameterD* = *Sortqual*
44 *ProcessconnectionD* = *Qual* $\overset{m}{\rightarrow}$ *Qual-set*

进程描述 *ProcessD* 是一个进程描述符。

*ParameterD** 创建进程时所使用的一个描述符表，用于检查实际参数的类别是否合适；即，这些描述符包含相对应的形式参数的类别。

Signalqual-**set** 分别是有效输入信号的集合和有效输出信号的集合（从进程图中所推导出的有效输出信号）。

ProcessconnectionD 是进程的信号路由与服务信号路由之间的关系。如果没有规定服务信号路由，则这个映射表是空的。它用于检查服务中的 VIA。

45 ***SortD*** :: ***Equations*$_1$** [***Parentqual***] [***Expression*$_1$**] ***Newqual***
46 ***Parentqual*** = ***Sortqual***

类别描述 *SortD* 是一个部分类型定义（新类型）的描述符。

Equations$_1$ 为该类别定义的 AS$_1$ 等式 *Equations*$_1$。当另一个类别继承这个类别时要用这些等式。

Parentqual 此类别的父辈类别。如果父辈限定表 *Parentqual* 为 **nil**，则这个类别就没有父辈。*Parentqual* 用于检验类别定义的递归性。

Expression$_1$ 是一个任选的 AS$_1$ 表达式，它对应于在部分类别定义中能被规定的任选的变量初始值。

Newqual 表示在 AS$_1$ 中使用的标识符。如果服务中定义了部分数据类型，那么在 *Newqual* 中它的名字已经改变。

47 ***SyntypeD*** :: ***Parentqual Newqual*** [***Expression*$_1$**] ***Range-condition*$_1$**

同义类型描述 *SyntypeD* 是一个同义类型的描述符。

Parentqual 是父辈类别；即，如果在 AS$_0$ 中规定的父辈是一个同义类型，它就是那个同义类型的父辈。

Newqual 表示 AS$_1$ 中使用的标识符。如果在一服务中定义了该同义类型，则在 Newqual 中它的名字已经改变。

Expression$_1$ 是任选的 AS$_1$ 表达式，它对应于能够在同义类型定义中规定的任选的变量的初始值。

Range-conclition$_1$ 是 AS$_1$ 的范围条件，它用于为同义类型变量产生缺省赋值。

48 ***GeneratorD*** :: ***Genparm*$_0$$^+$ *Properties*$_0$**

生成程序描述 *GeneratorD* 是一个数据类别生成程序的描述符。

Genparm$_0$$^+$ 取自于生成程序首部的 AS$_0$ 形式参数表

Properties$_0$ AS$_0$ 的生成程序体

49 ***SynD*** :: ***Sortqual Expr*$_0$**

同义描述 *SynD* 是一个同义词的描述符。

Sortqual 此同义词的类别。如果在 AS$_0$ 中没有这个类别，就从同义词定义中所含有的表达式类别中导出类别限定表 *Sortqual*。

Expr$_0$ *AS*$_0$ 的同义词表达式。

50 ***VarD*** :: ***Sortqual*** [REVEALED] [EXPORTED] [***Expression*$_1$**] ***Newqual***

变量描述 *VarD* 是一个变量的描述符。

Sortqual	变量的类别
REVEALED	任选的透露属性 REVEALED，与 AS_0 中的一样
EXPORTED	任选的出口属性 EXPORTED，与 AS_0 中的一样
[*Expression*$_1$]	在定义变量时规定的任选初始表达式的 AS_1 形式
Newqual	在 AS_1 中所用的限定表 *Qual*。如果在服务中定义了该变量，则在 *Newqual* 中它的名字就与原来的名字不同了。

51 ***ImportD*** :: ***Sortqual***

进口描述 *ImportD* 是一个进口变量的描述符，包含进口变量的类别。

52 ***ViewD*** :: ***Qual***

视见描述 *ViewD* 是一个视见变量描述符，包含相应的透露变量的限定表 *Qual*。

53 ***LiteralD*** :: ***Result***
54 ***Result*** = ***Sortqual***

字面描述 *LiteralD* 是一个字面值的描述符，其中含有该字面值的类别。

55 ***OperatorD*** :: ***Sortqual***$^+$ ***Result Newqual Explicit***
56 ***Explicit*** = ***Bool***

运算描述 *OperatorD* 是一个运算符的描述符。

Sortqual$^+$	对应于运算符变元的类别标识符表
Result	对应于结果的类别标识符
Newqual	在 AS_1 中使用的唯一的运算符标识符
Explicit	是一个标志。如果此运算符是隐含地从另一类别继承来的，则此标志为假。

57 ***ValueidD*** :: ***Mapvalue Sortqual-set Explicit***
58 ***Mapvalue*** = [***Qual***]

值标识描述 *ValueidD* 是一个公理变量（值标识符）的描述符。

Mapvalue	如果此标识符由字面值量化引入，该描述符则包含当前字面值。在量化的公理中，该值标识符由这个字面值替换。
Sortqual-set	类别的 *Qual* 的集合，在任何特定时间里，这些 *Qual* 对此值标识符都是合法的。如果此值标识符由显式量化引入，则这个集合仅包含一个类别。
Explicit	是一个标志，用来表示值标识符是否由显式量化所引入。

59 *ErrorD* :: ()

错误描述 *ErrorD* 用于检测同义词、信号表和生成程序的递归定义。在对这些构件的定义进行检测时，用一个 *ErrorD* 来代替它们的描述符使得在它们自己的定义内部对它们的任何使用都可以被检测到。它还可以用于屏蔽同义词，这些同义词不能用于选择定义的简单表达式中。

2.2 引用文字典 *Quotdict* 的说明

以下项目包含一些辅助信息，这些信息仅仅是由于一些实际的原因（为了减少 Meta-IV 函数的参数的数目和结果的数目）才被包括在字典 *Dict* 域中的。

1 *Quotdict* = SCOPEUNIT $\overset{m}{\rightarrow}$ *Qual* ∪
SCOPEUNIT $\overset{m}{\rightarrow}$ *Globalnames* ∪
LABELDICT $\overset{m}{\rightarrow}$ *Labeldict* ∪
STATEDICT $\overset{m}{\rightarrow}$ *Statedict* ∪
OUTSIGNALS $\overset{m}{\rightarrow}$ *Signalqual-set* ∪
DATATYPEDEF $\overset{m}{\rightarrow}$ *Data-type-definition*$_1$ ∪
SERVICES $\overset{m}{\rightarrow}$ (*Statetuplemap Servicetuple*) ∪
IMPLIED $\overset{m}{\rightarrow}$ *NameclosureD-set* ∪
IMPORTLIST $\overset{m}{\rightarrow}$ (*Name*$_0$ *Qual* [*Expr*$_0$])*

2 *Globalnames* = *Emptyqid Formuniquenm Formuniquenm*

3 *Labeldict* = *Label*$_0$ $\overset{m}{\rightarrow}$ *Transition*$_0$

4 *Statedict* = *Statename*$_0$ $\overset{m}{\rightarrow}$ (*StateD* | *ContenablestateD*)

5 *StateD* :: *Speclist* [*Importstateinf*]

6 *Importstateinf* = [*Statename*$_0$] [*Graph-node*$_1$ | *Decision-node*$_1$]

7 *ContenablestateD* :: *Transition*$_0$

8 *Statetuplemap* = *Statename*$_0$$^+$ $\overset{m}{\rightarrow}$ *Statename*$_0$

9 *Servicetuple* = *Servicequal**

10 *Speclist* = *Spec**

11 *Spec* = *Qual Statespec*$_0$

12 *Servicequal* = *Qual*

13 *Emptyqid* = *Qual*

14 *Formuniquenm* = *Name*$_0$

SCOPEUNIT 包含表示当前层次的限定表 *Qual*；即，此 *Qual* 表示所在的作用域单位的标识符。

GLOBALNAMES 包含一个含有一些实物名字的描述符 *Globalnames*，允许条件的变换和连续信号的变换需要这些实物名字。

emptyqid 以连续信号和允许条件方式，由进程发送给它们自己的信号。在 AS$_1$ 中的系统层声明此信号。

formuniquenm 连续信号中使用的两个变量的名字。在每一次进入一个连续信号状态时要更新第一个变量，它的新值与 *emptyqid* 信号一起被发送，这样每次形成一个新的唯一的值。第二个变量保持由 *emptyqid* 信号接受的值。在 AS$_1$ 中的每个进程中应声明这些变量。

LABELDICT 标号字典 *Labeldict* 拥有用于一个进程、过程或服务体内的每一个标号（连接符名字）的信息。它在体的实际变换发生之前就被构造出来了，它包含跟在所给标号（在连接符中）之后的跃迁 *Transition*$_0$。

STATEDICT

状态字典 *Statedict* 包含一个体的各个状态。状态的变换以状态字典 *Statedict* 为基础。如果状态描述符是一个连续允许状态描述符 *ContenablestateD*，则它表示一个含有连续信号或允许条件的状态。在 AS_1 中，这样的状态没有显式的表示法（它们由一些综合的状态来表示）。下一状态节点用导入综合状态的跃迁串来代替。

假如状态描述符是一个 *StateD*，它表示一个有 AS_1 表示法的状态。它包含：

speclist

该状态的输入描述符表。每个输入描述符（*Spec*）包含：

Qual

是一个限定词（*Qual*ifier），用以表示后继跃迁应被计算的范围。由于在状态体被变换到 AS_1 之前，多个服务已合并为一个状态字典 *Statedict*，如果这个状态字典 *Statedict* 来源于一个服务的分解，则此限定表 *Qual* 对各个输入描述符 *Spec* 是不同的。在其他的情况下 *Qual* 表示所在的进程或过程的标识符。

Statespec$_0$

AS_0 输入节点

Importstateinf

如果此状态是一个起源于进口表达式的隐式状态，则 *Importstateinf* 是存在的。它包含先前状态的名字 *Statename*$_0$，使得用一个划线跟在进口表达式后面的下一状态节点能够被恰当地替换。如果它是 **nil**，则这个进口表达式来源于一个初始跃迁。它也包含 AS_1 节点，在这节点里出现了含有进口表达式的表达式。如果此表达式含有不止一个进口表达式，那么只有一个进口状态描述符包含这样的一个节点（其它的为 **nil**）。

与正常的状态相反，在变换状态的同时，构成了进口状态 *Statedict* 的描述符。

OUTSIGNALS

包含一个信号集合。它用于从进程体中得出输出信号。

DATATYPEDEF

包含与每个作用域单位有关的 AS_1 数据类型定义 *Data-type-definition*$_1$。当遇到一个部分类型定义时，数据类型定义 *Date-type-definition*$_1$要用新的类别、运算符和等式来更新。

SERVICES

在由服务分解形式的一个状态字典 *Statedict* 的变换期间使用这个信息。把所有的服务在服务元组 *Servicetuple* 中进行排序。服务元组 *Servicetuple* 连同状态元组映射表 *Statetuplemap* 一起被用来把唯一的 AS_1 状态名与出现在各种服务之中的旧状态名联系起来。

IMPLIED

由变量类别和隐式变量名组成的偶对的集合，它是在进程定义的变换期间收集的。即，在公理变换期间收集隐式量化名字，在动作变换期间收集隐式进口变量名字。请注意，这里的名字闭包描述 *NameclosureD* 在概念上与出现在出口映射 *Exportmap* 中的名字闭包描述 *NameclosureD* 无关，尽管它们是相同的域。

IMPORTLIST

在表达式的变换期间，收集有关出现在这个表达式中的进口表达式的信息。在所得到的 AS_1表达式中，进口表达式由一个新的隐式名字来代替。在变换之后，生成（并加入到状态字典 *Statedict* 中）许多状态（每个进口表达式有一个状态）。跟随在一个新状态后面的跃迁从下列内容产生：使用该表达式的动作、隐式名字（*Name*$_0$）、进口变量的标识符（*Qual*）和用于进口表达式（*Expr*$_0$）中的任选的 Pid 表达式。由进口表 **IMPORTLIST** 表示的元组的长度与出现在构件中的进口表达式的数目相等。

2.3 其它域

1 *Decl*$_1$ = *Block-definition*$_1$ | *Channel-definition*$_1$ | *Signal-definition*$_1$ | *Signal-route-definition*$_1$ | *Procedure-definition*$_1$ | *Process-definition*$_1$ | *Signal-route-definition*$_1$ | *Syn-type-definition*$_1$ | *Data-type-definition*$_1$ | *Variable-definition*$_1$ | *View-definition*$_1$ | *Timer-definition*$_1$

2 *Context* = CONSTANT | AXIOMS | MAPPING | EXPRESSION

引入这些域（同义词）的目的是为了避免在函数类型规格中的累赘的重复。

Decl$_1$ 是一个速记符号，用来表示一个 AS$_1$定义。许多“定义变换”函数给出 AS$_1$定义作为结果。

Context 是一个速记符号，用来表示可能的上下文。在这个上下文中进行表达式的变换。这个上下文可以是常数（例如在回答的变换中）、公理、公理的映射部分或是任何其它情况。

3 *Processqual* = *Qual*

4 *Operatorqual* = *Qual*

这些域名字用于某些地方，以指出在所给定的上下文中此 *Qual*（标识符）是一特定的类型（即进程或运算符）。

5 *External-Information* = ...

外部信息 *External-Information* 包含了所需的附加信息来把语义赋给 SDL。这个信息是从外面被带进来的，因此把它作为函数 *definition-of-SDL*（与系统定义和预定义类别一起）的参数。由于 SDL 不规定信息是怎样构成的（比如说，类属的实在参数），因此使用外部信息 *External-Information* 的 Meta-IV 函数的定义是非形式的。

外部信息 *External-Information* 包含以下内容。

- 关于类属的实在参数的信息（相应的形式参数是外部同义词和任选节点中的非形式的正文）。
- 关于实际子集参数的信息，用来表明应该选择哪一个一致性的子集合；即，从外部信息 *External-Information* 中推出功能块标识符 *Block-identifier*$_1$的集合(参见定义 *SDL* 函数 *definition-of-SDL* 和选择一致性子集函数 *select-consistent-subset*)。
- 关于信道中的不确定延迟的信息（即一个“随机”函数）。
- 关于起动时间和时间单位的信息（用于模拟绝对时间）。

6 *Auxiliary-Information* = *Time-information Term-Information Is-expiredF DelayF*

7 *Time-information* = (*Ground-term*$_1$ → *Ground-term*$_1$) *Ground-term*$_1$

8 *Term-Information* = *Sort-identifier*$_1$ *Literal-operator-identifier*$_1$ *Literal-operator-identifier*$_1$ *Literal-operator-identifier*$_1$

9 *Is-expiredF* = *Ground-term*$_1$ *Ground-term*$_1$ → *Bool*

10 *DelayF* = () ⇒ *Bool*

辅助信息 *Auxiliary-Information* 包含一些用于解释的信息，系统定义 *System-definition* 和一致性子集的信息除外。辅助信息是在静态语义中构成的，并在系统处理器启动时作为参数传递给系统处理器（参见函数 *defi-*

nition-of-SDL)。辅助信息 *Auxiliary-Information* 由以下内容组成：

Time-information

时间信息 *Time-information* 中的函数用在动态语义中，用于更新当前的时间。给它一个 TIME 类别的 AS_1 字面值，它就返回另一个时间类别的字面值。此外，时间信息包含表示初始时间的基项 *Ground-term*$_1$。初始时间是在 SDL 系统之外定义的（即，它是从外部信息 *External-Information* 中得出的）。

Term-Information

AS_1 不包含有关标识符如何拼写的信息。但是，在动态语义中必须知道（即区别）四个 AS_1 标识符。这些标识符是 PID，NULL，TRUE 和 FALSE。项信息 *Term-information* 用来表示这些标识符，它们在静态语义中构成。

Is-expiredF

这个函数在静态语义中构成并在动态语义中用来测试一个给定的定时器是否已经到时。对于所给的两个时间类别的 AS_1 字面值，如果第一个参数的值大于或等于第二个参数，则返回 **true**（真）。

DelayF

这个函数用于模拟在通道中不确定的延迟（参见附件 *F*. 3 中的路径处理器）。这个函数从外部信息 *External-Information* 中导出，它具有一个重要的性质，也就是说它可以依赖于某些外部的物理参数。

3 AS_0 到 AS_1 的变换

3.1 主要函数

本节包含两个函数：

definition-of-SDL 这是从外面引用的最外层的函数。当把 SDL 系统（AS_0 形式）作为参数传递给它时，它就定义系统的语义(静态的和动态的)。它也形成静态语义和动态语义之间的连系。

transform-system 这是静态语义的入口函数。当把 SDL 系统（AS_0 形式）作为参数传给它时，如果 AS_0 形式在静态上是正确的（良形式的），它就返回一个基于 SDL 抽象语法形式（AS_1 形式）的规格。

***definition-of-SDL*(*extparms*, *systemdef*, *predefsorts*)** $\triangleq$ **(3.1.1)**

```
1  (let (as1, auxinf) = transform-system(systemdef, predefsorts, extparms) in
2   if as1 = nil then
3     undefined
4   else
5    (let subsetcut = select-consistent-subset(as1, extparms) in
6     start system(as1, subsetcut, auxinf)))
```

type: *External-Information* Sys_0 $Datadef_0{}^+ \Rightarrow$

目的 定义 SDL 的性质

参数

extparms	某些外部信息 *External-Information*（参见 2.3 节）
systemdef	表示 SDL 系统的 AS_0 树
predefsorts	AS_0 形式的预定义数据

算法

第 1 行	把系统变换为抽象语法形式（AS_1 形式）。
第 2 行	如果发现静态错误（即，如果导不出 AS_1 表示法）那么其行为是未定义的。
第 4 行	如果没有发现静态错误，则
第 5 行	选择功能块标识符 $Block\text{-}identifier_1$ 的集合用来表示一致性子集合
第 6 行	创建一个系统实例；即，创建一个象基础系统行为一样的 Meta-IV 进程。

transform-system(*genericsystem*, *predefdatasorts*, *extparms*) ≙ (3.1.2)

```
1  (let mk-Sys0(sysdef, refdeflist) = apply-generic-parameters(genericsystem, extparms) in
2   let mk-Sysdef0(snm, decllist, tnm) = sysdef in
3   let (as0global, globalentities) = as0-global-entities() in
4   let as1datadef = mk-Data-type-definition1(name-to-name1(create-unique-name()), {}, {}, {}, {}) in
5   (trap exit with (nil, nil) in
6    (let d = [⟨(SYSTEM, snm)⟩ ↦ mk-SystemD(),
7              SCOPEUNIT       ↦ ⟨(SYSTEM, snm)⟩,
8              GLOBALNAMES     ↦ globalentities,
9              DATATYPEDEF     ↦ as1datadef] in
10    let decllist' = replace-references(decllist, refdeflist)(⟨(SYSTEM, snm)⟩) in
11    let predefdict be s.t. (, predefdict) = transform-decllist(predefdatasorts)(predefdict + d) in
12    let decllist'' = remove-select(decllist', {})(predefdict) in
13    if (∃fulldict ∈ Dict)
14       ((trap exit with false in
15         (let (dict) = transform-decllist(decllist'' ⌢ ⟨as0global⟩ ⌢ predefdatasorts)(fulldict + d) in
16          fulldict = dict))) then
17      (let (as1dcl, dict) be s.t. (as1dcl, dict) =
18           transform-decllist(decllist'' ⌢ ⟨as0global⟩ ⌢ predefdatasorts)(dict + d) in
19       let as1 = make-as1tree(SYSTEM, snm, as1dcl, nil, nil, nil)(dict) in
20       let auxinf = generate-auxiliary-information(extparms)(dict) in
21       (tnm ∉ {snm, nil}
22          → exit("§2.2.2: 在系统定义中的结尾名不同于定义名"),
23        ¬(∃b ∈ elems decllist')(is-Blockdef0(b))
24          → exit("§2.4.2: 系统定义必须包含至少一个功能块"),
25        T → (as1, auxinf)))
26      else
27       (nil, nil))))
```

type: $Sys_0\ Datadef_0^+\ External\text{-}Information \rightarrow [System\text{-}definition_1]\ [Auxiliary\text{-}Information]$

目的

把 AS_0 形式的 SDL 系统变换为相应的 AS_1 形式

参数

genericsystem	一个类属系统定义
predefdatasort	也是 AS_0 形式的一连串预定义数据
extparameters	外部的信息（参见 2.3 节）

结果

结果是 AS_1 系统定义和某些将用于附件 F.3 中的信息。如果系统定义不是良形式的，对这两种结果都返回 **nil**。

算法

第 1 行　把类属系统定义变换为具体的系统定义，办法是通过为在系统中定义过的每个外部同义词提供一个表达式和通过把任选答案中的每个非形式正文变换为 AS_0 的条件表 *Conditionlist*$_0$。SDL 不定义怎样去进行这些变换。

第 2 行　令 *snm* 表示系统的名字，*decllist* 表示定义表以及令 *tnm* 表示尾名。

第 3 行　构成 AS_0 定义和相应于全局 emptyq 信号的全局名 *Globalnames* 描述符（见 *Quotdict*）。

第 4 行　为系统作用域单位构造初始的数据类型定义 *Data-type-definition*$_1$。在通过定义表（第 17 行）时，它包含所有定义在系统层次上的部分数据类型的信息。

第 5 行　如果变换声明表 *transform-decllist* 或任何其它函数被捕捉到，则规格说明不是良形式的。

第 6—9 行　构造一个初始的字典 *Dict*，这个字典 *Dict* 包括有系统描述符、表示系统层的作用域单位 **SCOPEUNIT** 项目、全局的名字和当前的数据类型定义 *Data-type-definition*$_1$。

第 10 行　在系统的定义表中插入所有的间接定义（*refdeflist*）。这个新的定义表（*decllist'*）不包含引用。

第 11 行　变换预定义的数据以构成仅由预定义数据类别组成的字典 *Dict*。这个结果仅在 SELECT 语句的删除期间使用（第 12 行）；即，如第 17 行所示，预定义类别在别处总被当作普通定义处理的。第 11 行可以这样来读：令 *predefdict* 是这样，使得在 *predefdict* 的作用域中（即与所有的预定义类别的词义信息一起）变换预定义类别的结果之一（其它结果是此处不用的 AS$_1$ 定义）就是 *predefdict* 自身。当预定义类别呈良形式时，总是存在这样一个字典 *Dict*。

第 12 行　从系统中删除所有的选择语句 SELECT。由于任选节点不包含任何定义，在适当的时候它们可以被删除。

第 13—16 行　如果存在一个字典 *Dict*（*fulldict*），它使得 *fulldict* 作用域中的 *decllist''* 能够被变换而不引起任何错误（即，在变换声明表函数 *transform-decllist* 中没有用 **exit**）并使得结果是 *fulldict* 自身，则 *decllist''* 就是良形式的。

第 17 行　如果是这样，就在这样一个字典 *Dict* 的作用域中变换 *decllist''*。其它结果（*as*$_1$ *dcl*）是 AS$_1$ 定义的集合。这种使用字典 *Dict* 的方法在模拟中是十分必要的，因此应该注意以下几点：

- 利用它来解决这个问题：SDL 中的名字在被正式定义之前就可以被引用。
- 变换定义的 Meta-IV 函数可以使用全部字典 *Dict*（所有的描述符）。例如，变换功能块定义函数 *transform-blockdef* 返回一个功能块定义 *Block-definition*$_1$和一个功能块的字典 *Dict* 的描述符，尽管在作为参数传给该函数的字典 *Dict* 中已经包含了这个功能块描述符也是如此。作为参数给出的这个字典 *Dict* 用于访问由功能块使用的实体的性质，但是字典 *Dict* 参数中的功能块描述符本身不能被使用（否则将不能保证字典 *Dict* 的解作为良形式规格而存在，这意味着 Meta-IV 规格可能是无效的）。
- 在这些定义的变换期间，返回的多个描述符被汇合起来，最后（当所有的定义全都变换了以后）函数 *transform-decllist* 的结果就与参数相同。
- 第 17 行的字典 *dict* 仅仅是一个描述字典 *Descriptordict*。在最后得到的结果 *Dict* 中没有来自于引用文字典 *Quotdict* 的贡献。
- 变换系统函数 *transform-system* 中的第 11 行和第 17 行是在完整的形式定义中所仅有的两个地方。在这里用了一个如此奥妙的方法来使用 **be such that** 构件。

第 19 行　构造系统定义 *System-definition*$_1$

第 20 行　生成将在解释期间使用的辅助信息 *Auxiliary-Information*

第 21 行　如果规定尾部名，则它必须等于系统名。

第 23 行　在系统中必须至少有一个功能块定义。

第 25 行　返回系统的 AS$_1$ 表示法和辅助信息。

as$_0$-*global-entities*() $\triangleq$ **(3.1.3)**

```
1  (let emptyqnm = create-unique-name(),
2       formunique1 = create-unique-name(),
3       formunique2 = create-unique-name() in
4   let emptyqid = mk-Id0(⟨⟩, emptyqnm) in
5   let globalnames = (emptyqid, formunique1, formunique2) in
6   let intgid = mk-Id0(⟨⟩, mk-Name0("INTEGER", nil)) in
7   let as0def = mk-Sigdef0(⟨mk-Sigelem0(emptyqnm, ⟨intgid⟩, nil)⟩) in
8   (as0def, globalnames))
```

type: () → *Sigdef*$_0$ *Globalnames*

目的

构造一个放在系统层上的全局的 AS$_0$ 定义和构造全*局名 Globalnames* 闭包（见*引用文字典 Quotdict* 的定义），这种闭包在变换期间是普通的。这个 AS$_0$ 定义定义了用于允许条件和连接信号中的 emptyq（空队列）信号。

算法

第 1 行　为 emptyq 信号创建一个唯一的名字。

第 2—3 行　为与连续信号一起用于连接中的两个变量创造独特的名字。在每个进程中都要定义这两个变量。

第 4 行　为 emptyq 信号构造一个 AS$_0$ 标识符。

第 5 行　构造全*局名 Globalnames* 闭包。

第 6 行　为整数类别构造一个 AS$_0$ 标识符。

第 7 行　为 emptyq 信号构造一个 AS$_0$ 定义。

第 8 行　返回 AS$_0$ 定义和全*局名 Globalnames* 闭包。

3.2 定义引用（reference）的替换

在这一节中，引用由间接定义来替换。入口函数 *replace-references*（替换引用函数）以系统层的定义和间接定义表作为变量。结果是一个不包含引用的定义表。

replace-references(*deflist*, *remotelist*)(*scopeunit*) $\triangleq$ **(3.2.1)**

```
1  if (∀i1, i2 ∈ ind remotelist)
2     (i1 ≠ i2 ∧
3       cases (remotelist[i1], remotelist[i2]):
4        ((mk-Blockdef0(mk-Id0(q1, nm1),,,), mk-Blockdef0(mk-Id0(q2, nm2),,,)),
5         (mk-Prdef0(mk-Id0(q1, nm1),,,,,,), mk-Prdef0(mk-Id0(q2, nm2),,,,,,)),
6         (mk-Procdef0(mk-Id0(q1, nm1),,,,), mk-Procdef0(mk-Id0(q2, nm2),,,,)),
7         (mk-Servicedef0(mk-Id0(q1, nm1),,,,), mk-Servicedef0(mk-Id0(q2, nm2),,,,)),
8         (mk-Chansubdef0(mk-Id0(q1, nm1),,), mk-Chansubdef0(mk-Id0(q2, nm2),,))
9           → nm1 = nm2 ⊃ (q1 ≠ q2 ∧ q1 ≠ ⟨⟩ ∧ q2 ≠ ⟨⟩),
10        T → true)) then
11    (let (deflist', remoteset) = remove-references(deflist, elems remotelist, false)(scopeunit) in
12     if remoteset = {} then
13       deflist'
14       else
15        exit("§2.4.1: 间接定义在系统定义中没有被引用 "))
16   else
17    exit("§2.4.1: 间接定义不是唯一的 ")
```

type: *Decl*$_0^+$ *Decl*$_0^*$ → *Qual* → *Decl*$_0^+$

目的

由间接定义替换系统中所有的引用

参数

deflist	用于系统层的定义表
remotelist	间接定义表

结果 不包含引用的系统层的定义表

算法

第 1—10 行	对于每两个不同的间接功能块定义、进程定义、过程定义、服务定义或子结构定义，必须满足以下要求：如果在它们的标识符内，它们有着相同的名字，则它们的限定词必须是不同的和非空的。
第 11—15 行	由它们的定义来替换引用。所得到的表是 *deflist'* 。在已经替换了所有的引用之后，不会再剩下间接定义。

remove-references(*dlist*, *remoteset*, *chansub*)(*unit*) $\triangleq$ (3.2.2)

```
1  if dlist = ⟨⟩ then
2    (⟨⟩, remoteset)
3   else
4    (let match-id(nm, id) = s-Name0(id) = nm ∧ s-Qualifier0(id) ∈ {⟨⟩, unit} in
5     cases hd dlist:
6      (mk-Blockref0(name)
7         → if (∃blkdef ∈ remoteset)(is-Blockdef0(blkdef) ∧ match-id(name, s-Blockid0(blkdef))) then
8             (let blkdef ∈ remoteset be s.t. is-Blockdef0(blkdef) ∧ match-id(name, s-Blockid0(blkdef)) in
9              let mk-Blockdef0(bid, decll, sub, tid) = blkdef in
10             if tid ∉ {nil, bid} then
11               exit("§2.4.1: 在定义中的结尾标识符和起始标识符不同 ")
12              else
13               (let blkdef' =
14                    mk-Blockdef0(mk-Id0(⟨⟩, name), decll, sub, mk-Id0(⟨⟩, name)) in
15                remove-references(⟨blkdef'⟩ ⌢ tl dlist, remoteset \ {blkdef}, chansub)(unit)))
16            else
17             exit("§2.4.1: 没有间接定义和引用相匹配 "),
18      mk-Prref0(name, inst)
19         → if (∃prdef ∈ remoteset)(is-Prdef0(prdef) ∧ match-id(name, s-Prid0(prdef))) then
20             (let prdef ∈ remoteset be s.t. is-Prdef0(prdef) ∧ match-id(name, s-Prid0(prdef)) in
21              let mk-Prdef0(pid, inst', b, d, e, f, tid) = prdef in
22              let inst'' = select-remote-number-of-instances(inst, inst') in
23              if tid ∉ {nil, pid} then
24                exit("§2.4.1: 在定义中的结尾标识符和起始标识符不同 ")
25               else
26                (let prdef' =
27                     mk-Prdef0(mk-Id0(⟨⟩, name), inst'', b, d, e, f, mk-Id0(⟨⟩, name)) in
28                 remove-references(⟨prdef'⟩ ⌢ tl dlist, remoteset \ {prdef}, chansub)(unit)))
29            else
30             exit("§2.4.1: 没有间接定义和引用相匹配 "),
31      mk-Procref0(name)
32         → if (∃procdef ∈ remoteset)(is-Procdef0(procdef) ∧ match-id(name, s-Procid0(procdef))) then
33             (let procdef ∈ remoteset be s.t. is-Procdef0(procdef) ∧ match-id(name, s-Procid0(procdef)) in
34              let mk-Procdef0(pid, parm, decll, body, tid) = procdef in
35              if tid ∉ {nil, pid} then
36                exit("§2.4.1: 在定义中的结尾标识符和起始标识符不同 ")
37               else
38                (let procdef' =
39                     mk-Procdef0(mk-Id0(⟨⟩, name), parm, decll, body, mk-Id0(⟨⟩, name)) in
40                 remove-references(⟨procdef'⟩ ⌢ tl dlist, remoteset \ {procdef}, chansub)(unit)))
41            else
42             exit("§2.4.1: 没有间接定义和引用相匹配 "),
43      mk-Serviceref0(name)
44         → if (∃servicedef ∈ remoteset)(is-Servicedef0(servicedef) ∧
45                                        match-id(name, s-Serviceid0(servicedef))) then
46             (let servicedef ∈ remoteset be s.t. is-Servicedef0(servicedef) ∧
47                  match-id(name, s-Serviceid0(servicedef)) in
48              let mk-Servicedef0(sid, a, b, d, tid) = servicedef in
49              if tid ∉ {nil, sid} then
50                exit("§2.4.1: 在定义中的结尾标识符和起始标识符不同 ")
51               else
52                (let servicedef' =
53                     mk-Servicedef0(mk-Id0(⟨⟩, name), a, b, d, mk-Id0(⟨⟩, name)) in
54                 remove-references(⟨servicedef'⟩ ⌢ tl dlist, remoteset \ {servicedef}, chansub)(unit)))
55            else
56             exit("§2.4.1: 没有间接定义和引用相匹配 "),
```

```
57      mk-Chansubref0(name)
58        → if (∃chansubdef ∈ remoteset)(is-Chansubdef0(chansubdef) ∧
59                                       s-Chansubid0(chansubdef) ≠ nil ∧
60                                       match-id(name, s-Chansubid0(chansubdef))) then
61            (let chansubdef ∈ remoteset be s.t. is-Chansubdef0(chansubdef) ∧
62                 match-id(name, s-Chansubid0(chansubdef)) in
63             let mk-Chansubdef0(cid, decll, tid) = chansubdef in
64             if tid ∉ {nil, cid} then
65               exit("§2.4.1: 在定义中的结尾标识符和起始标识符不同 ")
66               else
67                (let chansubdef' =
68                     mk-Chansubdef0(mk-Id0(⟨⟩, name), decll, mk-Id0(⟨⟩, name)) in
69                 remove-references(⟨chansubdef'⟩ ⌢ tl dlist, remoteset \ {chansubdef}, true)(unit)))
70          else
71           exit("§2.4.1: 没有间接定义和引用相匹配 "),
72      mk-Blocksubref0(name)
73        → if (∃chansubdef ∈ remoteset)(is-Chansubdef0(chansubdef) ∧
74                                       s-Chansubid0(chansubdef) ≠ nil ∧
75                                       match-id(name, s-Chansubid0(chansubdef))) then
76            (let chansubdef ∈ remoteset be s.t. is-Chansubdef0(chansubdef) ∧
77                 match-id(name, s-Chansubid0(chansubdef)) in
78             let mk-Chansubdef0(bid, decll, tid) = chansubdef in
79             if tid ∉ {nil, bid} then
80               exit("§2.4.1: 在定义中的结尾标识符和起始标识符不同 ")
81               else
82                (let blksubdef' =
83                     mk-Blocksubdef0(mk-Id0(⟨⟩, name), decll, mk-Id0(⟨⟩, name)) in
84                 remove-references(⟨blksubdef'⟩ ⌢ tl dlist, remoteset \ {chansubdef}, chansub)(unit)))
85          else
86           exit("§2.4.1: 没有间接定义和引用相匹配 "),
87      mk-Decomposition0(list)
88        → (let (list', rrest) = remove-references(list, remoteset, false)(unit) in
89           (mk-Decomposition0(list'), rrest)),
90      mk-Blockdef0(mk-Id0(q, nm), decll, sub, tid)
91        → (let (decll', rrest) = remove-references(decll, remoteset, false)(unit ⌢ ⟨(BLOCK, nm)⟩) in
92           let (sub', rrest') =
93               if sub = nil
94                 then (nil, rrest)
95                 else remove-references(⟨sub⟩, rrest, false)(unit ⌢ ⟨(BLOCK, nm)⟩) in
96           let (drest', rrest'') = remove-references(tl dlist, rrest', chansub)(unit) in
97           (⟨mk-Blockdef0(mk-Id0(q, nm), decll', sub', tid)⟩ ⌢ drest', rrest'')),
98      mk-Blocksubdef0(id, decll, tid)
99        → (let (, scnm) = unit[len unit] in
100          let mk-Id0(q, nm) = if id = nil then mk-Id0(⟨⟩, scnm) else id in
101          let unit' = unit ⌢ ⟨(SUBSTRUCTURE, nm)⟩ in
102          let (decll', rrest) = remove-references(decll, remoteset, false)(unit') in
103          (mk-Blocksubdef0(mk-Id0(q, nm), decll', tid), rrest)),
104     mk-Chandef0(nm, a, b, sub, tnm)
105       → (let (sub', rrest) =
106              if sub = nil
107                then (nil, remoteset)
108                else remove-references(⟨sub⟩, remoteset, chansub)(unit) in
109          let (drest', rrest') = remove-references(tl dlist, rrest, chansub)(unit) in
110          (⟨mk-Chandef0(nm, a, b, sub', tnm)⟩ ⌢ drest', rrest')),
```

```
111     mk-Chansubdef0(id, decll, tid)
112       → (let (, scnm) = unit[len unit] in
113          let mk-Id0(q, nm) = if id = nil then mk-Id0(⟨⟩, scnm) else id in
114          let unit' = unit ⌢ ⟨(SUBSTRUCTURE, nm)⟩ in
115          let (decll', rrest) = remove-references(decll, remoteset, true)(unit') in
116          (mk-Chansubdef0(mk-Id0(q, nm), decll', tid), rrest)),
117     mk-Prdef0(mk-Id0(q, nm), a, b, d, decll, body, tid)
118       → (let (decll', rrest) =
119              remove-references(decll, remoteset, false)(unit ⌢ ⟨(PROCESS, nm)⟩) in
120          let (body', rrest') =
121             if is-Body0(body) then
122               (body, rrest)
123              else
124               remove-references(⟨body⟩, rrest, false)(unit ⌢ ⟨(PROCESS, nm)⟩) in
125          let (drest', rrest'') = remove-references(tl dlist, rrest', chansub)(unit) in
126          (⟨mk-Prdef0(mk-Id0(q, nm), a, b, d, decll', body', tid)⟩ ⌢ drest', rrest'')),
127     mk-Procdef0(mk-Id0(q, nm), p, decll, body, tid)
128       → (let (decll', rrest) =
129              remove-references(decll, remoteset, false)(unit ⌢ ⟨(PROCEDURE, nm)⟩) in
130          let (drest', rrest') = remove-references(tl dlist, rrest, chansub)(unit) in
131          (⟨mk-Procdef0(mk-Id0(q, nm), p, decll', body, tid)⟩ ⌢ drest', rrest')),
132     mk-Servicedef0(mk-Id0(q, nm), a, decll, body, tid)
133       → (let (decll', rrest) =
134              remove-references(decll, remoteset, false)(unit ⌢ ⟨(SERVICE, nm)⟩) in
135          let (drest', rrest') = remove-references(tl dlist, rrest, chansub)(unit) in
136          (⟨mk-Servicedef0(mk-Id0(q, nm), a, decll', body, tid)⟩ ⌢ drest', rrest')),
137     mk-Select0(expr, decll)
138       → if is-wf-entities(decll, chansub)(unit) then
139            (let (decll', rrest) = remove-references(decll, remoteset, chansub)(unit) in
140             let (drest', rrest') = remove-references(tl dlist, rrest, chansub)(unit) in
141             (⟨mk-Select0(expr, decll')⟩ ⌢ drest', rrest'))
142           else
143            exit("§4.3.3: 选择中的定义在那个作用域单位中是不允许的"),
144     T → (let (drest, rrest) = remove-references(tl dlist, remoteset, chansub)(unit) in
145          (⟨hd dlist⟩ ⌢ drest, rrest))))
```

type: $(Decl_0^* \mid Blocksubdef_0 \mid Chansubdef_0 \mid Decomposition_0)\ Refdecl_0^*\ Bool \rightarrow Qual \rightarrow (Decl_0^* \mid Blocksubdef_0 \mid Chansubdef_0 \mid Decomposition_0)\ Refdecl_0^*$

目的

由一个间接定义来替换定义表中的每个引用

参数

dlist	AS_0 定义表
remoteset	间接定义的集合
chansub	为 **true**，仅当 *dlist* 表示一个信道子结构的定义表时。这个参数用于检查正确的连接 *Connect*。（参见是良形式实体函数 *is-wf-entities*）。
unit	是定义表所在的上下文（由一个限定词来表示）

结果

是不包含引用的 AS_0 定义表，而且表中删除了没有找到引用的间接定义的集合

算法

第 1 行

当定义集合（*dlist*）为空时，不返回定义和经过修改的间接定义集合。（删除引用函数 *remove-references* 递归地返回完整的定义表）

第 4 行　定义一个实用函数，如果引用中的一个 *nm*（$Name_0$）与一个 *id*（来自一个定义）匹配时，即如果 *id* 中的那个名字等于引用中的 $Name_0$，并且限定词或者是空的，或者与放置在该引用的作用域单位相等时，实用函数返回 **true**。

第 5 行　考虑定义表中的第一个定义。

第 6 行　如果这个定义是一个功能块引用，则必须存在一个间接功能块定义，该定义在它的标识符中有着与功能块引用相同的名字（由 *match-id* 来检查，参见第 4 行）并且对它来说，它的标识符内的限定词或者是空的，或者与放置在该引用的作用域单位相等。

第 8 行　设 *blkdef* 表示那个功能块定义。

第 10—11 行　在间接功能块定义中的结束标识符必须或是被省略或是等于起始标识符。

第 13 行　重新构造功能块定义，使其在开始标识符与结束标识符中带有空限定词。

第 15 行　把它加到定义表中使得功能块中所包围的定义在下一个递归层中被处理，并且从间接定义的集合里删除这个功能块定义。

第 18—30 行　对引用功能块定义采用相同的处理方法如上所示，但第 22 行除外。在第 22 行中，用于引用的实例数 $Instances_0$ 要和用于间接定义的实例数 $Instances_0$ 进行比较。

第 31—42 行　对引用功能块定义采用相同的处理方法如上所示。

第 43—56 行　对引用功能块定义采用相同的处理方法如上所示。

第 57—71 行　对引用功能块定义采用相同的处理方法如上所示。在第 59 行要检查：如果信道子结构定义是间接的话，信道子结构标识符应该出现。

第 72—86 行　对引用功能块定义采用相同的方法如上所示。除非功能子结构定义在语法上作为更一般的信道子结构定义出现，功能块子结构就这样转换。

第 87—89 行　如果该定义是一个分解，则替换分解中所包含的定义表中的引用。

第 90 行　如果此定义是一个功能块定义，则

第 91 行　删除功能功中所包含的定义表中的引用。

第 92 行　删除功能块子结构中的引用。

第 96 行　删除包含功能块定义的定义表余下部分中的引用，并且

第 97 行　返回与其它修改过的定义拼接起来的修改过的功能块定义，并返回那个修改过的间接定义表。

第 98—103 行　删除功能块子结构定义中的引用。

第 104—110 行　删除信道定义中的引用。

第 111—116 行　删除信道子结构定义中的引用。

第 117—126 行　删除进程定义中的引用。如果该进程包含一个分解（在第 120 行测试的），则这个分解中的引用也必须被替换。

第 127—131 行　删除过程定义中的引用。

第 132—136 行　删除服务定义中的引用。

第 137—141 行　删除选择定义中包含的引用并且检查选择中的每个定义在包围的作用域单位中是否允许。

第 144 行　在其它情况下，把定义不加改变地返回。

select-remote-number-of-instances(*inst*, *reminst*) ≙ (3.2.3)

```
1  (inst = nil ∧ reminst ≠ nil
2     → reminst,
3   reminst = inst ∨ reminst = nil
4     → inst,
5   T → exit ("§2.4.4: 间接的实例个数的规格与进程引用不匹配"))
```

type: $[Instances_0]\ [Instances_0] \rightarrow [Instances_0]$

目的

在一进程被引用的情况下为该进程选择实例数 $Instances_0$

参数

inst 为进程引用指定的实例数 $Instances_0$

reminst 为间接定义指定的实例数 $Instances_0$

算法

第 1 行 如果仅为间接定义指定实例数 $Instances_0$，则返回那个实例数 $Instances_0$ 规格。

第 3 行 如果进程引用的实例数 $Instances_0$ 等于间接定义的实例数 $Instances_0$，或者仅为引用指定实例数 $Instances_0$，则返回那个实例数 $Instances_0$ 规格。

第 5 行 否则该进程的实例数 $Instances_0$ 规格是不一致的。

is-wf-entities(*decllist*, *ischannelsub*)(*level*) ≙ (3.2.4)

```
1  if decllist = ⟨⟩ then
2    true
3   else
4    (let (q, ) = level[len level] in
5     is-wf-entities(tl decllist, ischannelsub)(level) ∧
6      cases hd decllist:
7      (mk-Blockdef0(, , ),
8       mk-Chandef0(, , , )
9         → q ∈ {SYSTEM, SUBSTRUCTURE},
10      mk-Connect0(id, )
11        → q ∈ {SUBSTRUCTURE, BLOCK} ∧ (id = ENV ⊃ (q = SUBSTRUCTURE ∧ ischannelsub)),
12      mk-Sigroutedef0(, , ),
13      mk-Prdef0(, , , , , , )
14        → q = BLOCK,
15      mk-Signallistdef0(, ),
16      mk-Sigdef0()
17        → q ∉ {SERVICE, PROCEDURE},
18      mk-Procdef0(, , , , ),
19      mk-Vardef0(, , )
20        → q ∈ {SERVICE, PROCESS, PROCEDURE},
21      mk-Importdef0(),
22      mk-Viewdef0(),
23      mk-Timerdef0()
24        → q ∈ {SERVICE, PROCESS},
25      T → true))
```

type: $Decl_0^*\ Bool \rightarrow Qual \rightarrow Bool$

目的

检查在所给上下文中一个起始于选择定义的定义表在语法上是否允许。但是，这里不处

理分解。

参数

decllist	要进行检查的定义表
ischannelsub	这是一个标志，如果 *decllist* 属于一个信道子结构，则此标志为 **true**。

结果 假如被允许的话，则结果为真。

算法

第 1 行	当通过时，则返回 **true**。
第 4 行	从表示当前作用域单位的限定表 *Qual* 中抽取作用域类型（*q*）。
第 5—25 行	如果第一个定义是良形式的（第 7—25 行）且定义表的其余部分也是良形式的（第 5 行），则这个定义表就是良形式的。
第 7—8 行	如果第一定义是一个功能块定义或一个信道定义，则这个上下文的作用域单位必须是系统或是一个功能块子结构，或是一个信道子结构。
第 10 行	如果它是一个连接，则这个作用域单位必须是一个功能块子结构或一个信道子结构，或是一个功能块。
第 12—13 行	如果它是一个进程定义的信号路由定义，则这个作用域单位必须是一个功能块。
第 15—16 行	如果它是一个信号表定义或一个信号定义，则这个作用域单位一定不能是一个服务或一个过程。
第 18—19 行	如果它是一个过程定义或一个变量定义，则这个作用域单位必须是一个服务或一个进程，或是一个过程。
第 21—23 行	如果它是一个进口定义或一个视见定义，或是一个定时器定义，则这个作用域单位必须是一个服务或一个进程。

3.3 选择定义的消除

在这节中，对选择定义求结果。入口函数是在变换系统函数 *transform-system* 中采用的消除选择函数 *remove-select*。对每个定义表采取以下步骤：

1. 收集定义表中所定义的类别名字，包括那些在一个被包含的选择定义中定义的类别。把这些类别加到周围作用域单位中所收集的类别中。这是必要的，这样做可以查找出由于预定义类别被再次定义所造成的混乱。
2. 收集所有定义表中定义的同义词或包含选择定义的定义表中定义的同义词。在 *Dict* 中为这些同义词放入一个 *ErrorD* 描述符来表示它们还不能使用。要标识一些同义词：它们是用所收集到的（可见的）类别来定义的同义词或者它们是拥有不止一个定义（如果它们在不同的选择定义中被定义，那么它们也是多重定义的）的同义词。由于使用它们会导致不同的结果，这与选择的次序有关，所以它们根本不能用于选择定义之中。注意，仅仅是在选择定义的消除期间不能使用这些同义词。以后，它们不必多重定义。
3. 消除这些选择定义。每次一个选择定义由它所包含的定义表替换时，定义表中同义词的描述符由一个 *ErrorD* 变为一个 *SynD*，除了那些不允许使用的同义词而外。
4. 当所有的选择定义已被消除后，为结果定义表中的所有同义词（也为先前不能使用的那些同义词）构造 *SynD* 描述符，并且收集结果定义表中定义的类别。
5. 采用更新了的 *dict* 和采用所收集的类别来消除包围的作用域单位中的选择定义。

由于任选的跃迁不包含任何定义，因此在状态体被变换为 AS 时，可以求出它们的结果。

remove-select(*decllist*, *sursorts*)(*dict*) $\triangleq$ (3.3.1)

```
1  (let sorts = sursorts ∪ collect-sorts(decllist) in
2   let (allsyn, illsyn) = collect-illegal-synonyms(decllist, sorts) in
3   let dict' = [dict(SCOPEUNIT) ↷ ⟨(VALUE, nm)⟩ ↦ mk-ErrorD() | nm ∈ allsyn] in
4   let decllist' = remove-select-in-decllist(decllist, illsyn)(dict') in
5   let dict'' = repeat-collecting-synonyms(decllist', {})(dict) in
6   let sorts' = sursorts ∪ collect-sorts(decllist') in
7   ⟨remove-select-in-enclosed-scopeunit(decllist[i], sorts')(dict'') | 1 ≤ i ≤ len decllist⟩)
```

type: $(Decl_0 \mid Blocksubdef_0 \mid Decomposition_0)^*$ $Name_0$-**set** $\rightarrow$ *Dict* $\rightarrow$
$(Decl_0 \mid Blocksubdef_0 \mid Decomposition_0)^*$

目的

为一个作用域单位扩展选择定义以及为定义表内包含的多个作用域单位中的所有选择定义扩展选择定义

参数

decllist
这个定义表包含选择定义。为了方便，功能块子结构与服务分解在这里的上下文中也被当作定义。

sursorts
（不包括预定义的）类别的集合，它在定义表中是潜在可见的（即，有可能依赖于他们是否被所包含的选择定义所选择）。在预定义的数据重新被定义的情况下，为了获得正确的处理，需要这种信息。

dict
是字典 *Dict*，它仅包含预定义类别的描述符和可见的但非局部的预定义类别的同义词的描述符。

结果

是一个定义表，表中已扩展了所有的选择定义

算法

第 1 行　收集在定义表中定义的或在选择定义中定义的类别。

第 2 行　令 *allsyn* 表示在定义表中定义的或在所包含的选择定义中定义的同义词。令 *illsyn*（它是 *allsyn* 的一个子集合）表示在为这个定义表消除选择定义时不能用的同义词。

第 3 行　把错误描述符 *ErrorD* 放入字典 *Dict*，以表示局部定义的同义词还不能使用。

第 4 行　删除这些选择定义。

第 5 行　为了结果定义表内的同义词把一个 *SynD* 描述符放入 *Dict* 中。

第 6 行　再一次收集类别，这次要使用结果定义表。

第 7 行　删除所包含的作用域单位中的选择定义。

remove-select-in-decllist(*decllist*, *illsyn*)(*dict*) ≙ **(3.3.2)**

```
1  (let dict' = repeat-collecting-synonyms(decllist, illsyn)(dict) in
2   if (∃d ∈ elems decllist)(is-Select0(d)) then
3     (if (∃d ∈ elems decllist)(is-Select0(d) ∧ is-wf-simple-expr(s-Expr0(d))(dict')) then
4        (let d ∈ elems decllist be s.t. is-Select0(d) ∧ is-wf-simple-expr(s-Expr0(d))(dict') in
5         let mk-Select0(expr, decllist') = d in
6         let selected = eval-simple-expr(expr, "BOOLEAN")(dict') in
7         let decllist'' = ⟨d' ∈ elems decllist | d' ≠ d⟩ in
8         if selected then
9           remove-select-in-decllist(decllist' ⌢ decllist'', illsyn)(dict)
10          else
11          remove-select-in-decllist(decllist'', illsyn)(dict))
12       else
13        exit "§4.3.2: 简单表达式不是预定义类别的或其中包含未定义标识符的") )
14    else
15     decllist)
```

type: $(Decl_0 \mid Blocksubdef_0 \mid Decomposition_0)^*$ $Name_0$-**set** → *Dict* → $(Decl_0 \mid Blocksubdef_0 \mid Decomposition_0)^*$

目的　　为一个定义表删除选择定义

参数

decllist	包含选择定义的定义表
illsyn	在选择定义的简单表达式中不能采用的同义词
dict	是字典 *Dict*，它仅含预定义类别的描述符或可见同义词的描述符（*SynD* 或 *ErrorD*）。

结果　　已删除所有选择定义的定义表

算法

第 1 行	用（预定义类别的）同义词的描述符来更新 *dict*，这些同义词是在定义表中定义的（但不包含在选择定义中）。
第 2 行	如果该定义表中（仍然）存在一个选择定义，则
第 3 行	如果该选择定义中的表达式是良形式的，则
第 4 行	令 *d* 表示那个选择定义。
第 6 行	求 *d* 中的简单表达式的值。
第 7 行	构造一个已经删除了 *d* 的定义表。
第 8 行	如果选择了该选择定义中的定义，则
第 9 行	继续删除选择定义，在这里定义表被扩展，以包括来自于选择定义的定义。
第 11 行	如果没有选择此选择定义，则继续处理定义表，在这个定义表中，该选择定义已被删除。
第 15 行	如果定义表中的全部选择定义已经被删除，则返回这个定义表。

eval-simple-expr(*expr*, *sort*)(*dict*) ≙ **(3.3.3)**

```
1  (let predefqual = get-predef-sort(sort)(dict) in
2   let (as1tree, , ) = transform-expr(expr, CONSTANT, {predefqual})(dict) in
3    eval-expr(as1tree, sort)(dict))
```

type: $Expr_0$ $Char^+$ → *Dict* → (*Intg* | *Bool*)

目的　　求一个简单表达式的值

参数

expr AS_0 简单表达式

sort 表达式类别的拼字，此类别或是“Boolean”或是“*Integer*”

结果 如果 SDL 类别是“Integer”，则结果是 Meta-IV 类型 *Intg*；如果 SDL 类别是“Boolean”，则结果是 Meta-IV 域 *Bool*。

算法

第 1 行 构成类别的限定表 *Qual*

第 2 行 将表达式变换为 AS_1，指明表达式是（必须是）一个常数（基）表达式并宣布表达式是（必须是）指定的类别。

第 3 行 求 AS_1 表达式的值（这个函数没有被形式地定义）。

***repeat-collecting-synonyms*(*decllist*, *illsyn*)(*dict*)** $\triangleq$ **(3.3.4)**

```
1  (let dict' = collect-synonyms(decllist, illsyn)(dict) in
2   if dict' = dict then dict' else repeat-collecting-synonyms(decllist, illsyn)(dict'))
```

type: $Decl_0^*$ $Name_0$-**set** → *Dict* → *Dict*

目的 收集所有良形式同义词的描述符（即，在这一阶段上的这些同义词不依赖在选择定义中定义的同义词）

参数

decllist 要被检查的定义表

illsyn 来自于 *decllist* 的同义词，它们不能被使用，即，它们的 *ErrorD* 描述符不应该变成一个 *SynD* 描述符。

结果 用良形式同义词的描述符来更新的 *Dict*

算法

第 1 行 通过遍历定义表 *decllist* 来收集该表中的良形式同义词的描述符。

第 2 行 如果在 *dict'* 中没有更多的同义词可以被改变，则返回 *dict*。否则

第 2 行 再次遍历那个表。由于同义词可以是相互依赖的，所以更多的同义词定义现在可以是良形式的了（在 *dict'* 的上下文中）。

collect-synonyms(*decllist*, *illsyn*)(*dict*) ≙ **(3.3.5)**

```
1  if decllist = ⟨⟩ then
2    dict
3   else
4    (if is-Synonymdef0(hd decllist) then
5       (let mk-Synonymdef0(nm, sort, expr) = hd decllist in
6        let qualset = {qual ∈ dom dict | is-SortD(dict(qual))} in
7        let sortset =
8            if sort = nil then
9              qualset
10           else
11            (trap exit with {} in
12             {get-parent(get-visible-qual(sort, TYPE)(dict))(dict)}) in
13       let synqual = dict(SCOPEUNIT) ⌢ ⟨(VALUE, nm)⟩ in
14       let (as1tree, qset, ) =
15           if sortset = {} then
16             (nil, {}, )
17           else
18            (trap exit with (nil, {}, ) in
19             transform-expr(expr, CONSTANT, sortset)(dict + [synqual ↦ mk-ErrorD()])) in
20       if as1tree = nil ∨ card qset ≠ 1 ∨ nm ∈ illsyn then
21         collect-synonyms(tl decllist, illsyn)(dict)
22        else
23         (let sortqual ∈ qset in
24          let d = [synqual ↦ mk-SynD(sortqual, expr)] in
25          collect-synonyms(tl decllist, illsyn)(dict + d)))
26     else
27      collect-synonyms(tl decllist, illsyn)(dict))
```

type: $Decl_0^*$ $Name_0$-**set** → *Dict* → *Dict*

目的　遍历一个定义表并收集所有良形式的同义词的描述符

参数

decllist　包含同义词定义的定义表

illsyn　来自于 *decllist* 的不能被使用的同义词；即，它们的 *ErrorD* 描述符不应该变成一个 *SynD* 描述符。

dict　是一个字典 *Dict*，它包含预定义类别的描述符和可见同义词的描述符。

结果　是一个字典 *Dict*，对于良形式的同义词用 *SynD* 描述符来更新。

算法

第 1 行　遍历之后，则返回（已被更新的）字典 *Dict*。

第 4 行　如果表中的下一个定义是个同义词定义，则

第 6 行　从 *dict* 中抽取预定义类别的 *Qual*（标识符）。

第 7—12 行　如果同义词定义中指定了一个类别，则表达式的合法类别的集合仅包含那个类别（如果被指定的类别不是良形式的，则这个集合为空），否则，这个合法类别的 集合包括所有的预定义类别。

第 13 行　令 *synqual* 表示同义词的 *Qual*。

第 14—19 行　试把表达式变换成 AS_1。如果不成功，那么或者是一个静态的错误（以后会被抓到的），或者那个表达式包含还没有考虑到的同义词（它们可能是在一个选择定义中被定义或者在 tl *decllist* 中被定义）。

第 20 行　　如果失败了，即，如果 AS_1 表达式 as_1 *tree* 不存在或者那个表达式的类别是不可确定的（被返回的类别集合的基数与 1 不同），或者如果该同义词是不能被使用的同义词之一，则继续处理定义表的其余部分，否则

第 23 行　　令 *Sortqual* 表示表达式的类别。

第 24—25 行　　用同义词描述符来更新 *dict* 并遍历余下的定义表。

第 27 行　　如果定义表内的第一个定义不是一个同义词定义，则继续处理表中的其余部分。

***is-wf-simple-expr*(*expr*)(*dict*)** $\triangleq$ **(3.3.6)**

```
1  (let boolqual = get-predef-sort("BOOLEAN")(dict) in
2   trap exit with false in
3   (let (,,) =transform-expr(expr, CONSTANT, {boolqual})(dict) in
4    true))
```

type: $Expr_0 \rightarrow Dict \rightarrow Bool$

目的　　检查一个简单的布尔表达式是否是良形式的。如果它不是良形式的，这可能是因为它是用还没有被编入 *dict* 中的同义词来定义的原故。

参数

expr　　要检查的 AS_0 表达式

结果　　如果此 AS_0 表达式是良形式的，则结果为真。

算法

第 1 行　　抽取布尔类别的 *Qual*（标识符）。表达式的合法类别的集合仅包括这一个类别。

第 2—3 行　　如果此变换函数被捕捉到错误，则返回假。否则

第 4 行　　返回真。

remove-select-in-enclosed-scopeunit(*decl*, *sorts*)(*dict*) ≜ (3.3.7)

```
1  (let level = dict(SCOPEUNIT) in
2   cases decl:
3    (mk-Blockdef0(mk-Id0(q, nm), decll, blksub, tid)
4       → (let dict' = dict + [SCOPEUNIT ↦ level ⌢ ⟨(BLOCK, nm)⟩] in
5          (let (decll', blksub') =
6               if blksub = nil then
7                 (remove-select(decll, sorts)(dict'), nil)
8               else
9                (let dlist = remove-select(decll ⌢ ⟨blksub⟩, sorts)(dict') in
10                let i be s.t. is-Blocksubdef0(dlist[i]) in
11                (⟨dlist[n] | 1 ≤ n ≤ len dlist ∧ n ≠ i⟩, dlist[i])) in
12           mk-Blockdef0(mk-Id0(q, nm), decll', blksub', tid))),
13    mk-Servicedef0(mk-Id0(q, nm), sigl, decll, body, tid)
14       → (let dict' = dict + [SCOPEUNIT ↦ level ⌢ ⟨(SERVICE, nm)⟩] in
15          let decll' = remove-select(decll, sorts)(dict') in
16          mk-Servicedef0(mk-Id0(q, nm), sigl, decll', body, tid)),
17    mk-Chandef0(nm, p1, p2, chansub, tnm)
18       → if chansub = nil then
19            decl
20          else
21           (let mk-Chansubdef0(id, decll, tid) = chansub in
22            let nm' = if id = nil then nm else s-Name0(id) in
23            let dict' = dict + [SCOPEUNIT ↦ level ⌢ ⟨(SUBSTRUCTURE, nm')⟩] in
24            let decll' = remove-select(decll, sorts)(dict') in
25            mk-Chandef0(nm, p1, p2, mk-Chansubdef0(id, decll', tid), tnm)),
26    mk-Prdef0(mk-Id0(q, nm), inst, parm, inpset, decll, body, tid)
27       → (let dict' = dict + [SCOPEUNIT ↦ level ⌢ ⟨(PROCESS, nm)⟩] in
28          let (decll', body') =
29              if is-Body0(body) then
30                (remove-select(decll, sorts)(dict'), body)
31               else
32                (let dlist = remove-select(decll ⌢ ⟨body⟩, sorts)(dict') in
33                 let i be s.t. is-Decomposition0(dlist[i]) in
34                 (⟨dlist[n] | 1 ≤ n ≤ len dlist ∧ n ≠ i⟩, dlist[i])) in
35          mk-Prdef0(mk-Id0(q, nm), inst, parm, inpset, decll', body', tid)),
36    mk-Blocksubdef0(id, decll, tid)
37       → (let (, nm) = level[len level] in
38          let nm' = if id = nil then nm else s-Name0(id) in
39          let dict' = dict + [SCOPEUNIT ↦ level ⌢ ⟨(SUBSTRUCTURE, nm')⟩] in
40          let decll' = remove-select(decll, sorts)(dict') in
41          mk-Blocksubdef0(id, decll', tid)),
42    mk-Decomposition0(decll)
43       → (let decll' = remove-select(decll, sorts)(dict) in
44          if (∀d ∈ elems decll')(is-Sigroutedef0(d) ∨
45                                 (is-Connect0(d) ∧ is-Id0(s-Connectpoint0(d))) ∨
46                                 is-Servicedef0(d)) then
47            mk-Decomposition0(decll')
48          else
49           exit("§4.3.3: 所选择的定义在那个作用域单位中是不允许的"));
50    T → decl))
```

type : (*Decl*$_0$ | *Blocksubdef*$_0$ | *Decomposition*$_0$) *Name*$_0$-set → *Dict* → (*Decl*$_0$ | *Blocksubdef*$_0$ | *Decomposition*$_0$)

目的

删除一被包围的作用域单位中的选择定义

参数

decl 一个定义，它可能是一个作用域单位

sort　　可见的类别（不包括预定义的）

结果　　一个定义，在这个定义中选择定义已被删除

算法

第 1 行　　用 *level* 表示此范围（定义在这个范围中出现）。

第 3—12 行　　如果定义是一个功能块定义，则删除这个功能块的定义表中的选择定义（第 7 行）。如果这个功能块包含一个子结构，则它也被当作是一个定义。选择定义在这个功能块的范围中被删除，即，*Dict* 中的作用域单位 **SCOPEUNIT** 被更新用以表示此功能块。

第 13—41 行　　对于其它种类的作用域单位的处理与在定义是功能块的情况中的处理一样。

第 42—49 行　　如果定义是一个分解（尽管它不是一个作用域单位，但它可以包含选择定义），则检查那个结果定义表（在删除选择定义之后）是否仅包含服务信号路由定义、服务信号路由连接和服务定义。服务信号路由连接中的 $Connectpoint_0$ 一定不能是 **ENV**（如果该连接包含在一个选择定义中，在语法上就不检查这种情形）。

collect-illegal-synonyms(*decllist*, *sorts*) $\triangleq$ **(3.3.8)**

```
1  (if decllist = ⟨⟩ then
2     ({},{})
3   else
4    (let (restsyn, restillsyn) = collect-illegal-synonyms(tl decllist, sorts) in
5     let (synset, illsynset) =
6          cases hd decllist:
7           (mk-Synonymdef0(nm, sort, )
8              → if (sort ≠ nil ∧ s-Name0(sort) ∈ sorts ∧ s-Qualifier0(sort) ≠ ⟨(SYSTEM, )⟩) then
9                   ({nm},{nm})
10                 else
11                  ({nm},{}),
12            mk-Select0(, dlist)
13              → collect-illegal-synonyms(dlist, sorts),
14            T → ({},{})) in
15     (restsyn ∪ synset, restillsyn ∪ illsynset ∪ (restsyn ∩ synset))))
```

type:　$Decl_0^*$ $Name_0$-**set** → $Name_0$-**set** $Name_0$-**set**

目的　　收集定义表中（潜在地）所定义的同义词并且辨认出那些不能被用于选择定义的任何简单表达式中的同义词

参数

decllist　　定义表

sorts　　在定义表中（潜在地）定义了的类别

结果　　被潜在定义的同义词和不能被使用的同义词所组成的集合

算法

第 1 行　　当处理完定义表时，返回空集。

第 4 行　　收集定义表剩余部分的同义词集合。

第 5 行　　令 *synset* 表示这个定义中所定义的同义词，而用 *illsynset* 表示这些同义词中的那些不能被使用的同义词。

第 7—11 行　　如果定义是一个同义词定义，则这个同义词名字是 *synset*；如果类别名是可见类别中的一个，就不能使用这个同义词名字，并且其限定词不表示系统层。请注意，*Dict* 仅包含预定义类别的描述符。所以仅当一个预定义的类别已被再次定义时，这个检查才是重要的。

第 12 行　　如果定义是一个选择定义，则为所包含的定义收集同义词集合。

第 15 行　　被定义同义词名字的结果集合是这个定义的名字与余下定义的名字的联合（∪）。不可用的同义词的结果集合是这个定义的名字联合余下定义的不可用的同义词，再联合既在这个定义中定义又在余下的定义中定义的名字。

collect-sorts(*decllist*) ≜ **(3.3.9)**

```
1  (if decllist = ⟨⟩ then
2     {}
3   else
4    (collect-sorts(tl decllist) ∪
5      cases hd decllist:
6      (mk-Syntypedef0(nm,,,,),
7       mk-Partialtypedef0(nm,,,,)
8         → {nm},
9       mk-Select0(, dlist)
10        → collect-sorts(dlist),
11      T → {})))
```

type：　$Decl_0^* \rightarrow Name_0$-**set**

目的　　收集在一个定义表中潜在定义的类别的类别名。

参数

decllist　　定义表

结果　　潜在定义的类别名的集合（它们实际上是否被定义要取决于选择定义）

算法

第 1 行　　当处理完定义表时，返回空集。

第 6—7 行　　如果定义是一个部分类型定义或一个同义类型定义，则抽取这个类别名。

第 9 行　　如果定义是一个 选择定义，则收集所含定义表中被潜在定义的类别名。

3.4 定义的变换

transform-decllist(*decllist*)(*dict*) $\triangleq$ **(3.4.1)**

```
1  (if decllist = ⟨⟩ then
2     ({}, [])
3   else
4    (let (as1 dcl, d) = transform-decl(hd decllist)(dict) in
5     let (as1 dcll, d') = transform-decllist(tl decllist)(dict) in
6     if dom d ∩ dom d' ≠ {} then
7       exit("§2.2.2: 在相同作用域单位和相同实体类中的两个定义定义了相同的名字")
8      else
9       (as1 dcl ∪ as1 dcll, d + d')))
```

type: $Decl_0^* \rightarrow Dict \rightarrow Decl_1$-**set** $Dict$

目的 把 AS_0 定义表变换为一个 AS_1 定义的集合

参数 一个 AS_0 定义表

结果 是从 AS_0 定义表得到的 AS_1 定义和对字典 *Dict* 的贡献。注意，与表达式和图形变换函数相反，这不是被返回的全部的 *dict*。这意味着返回的描述符（除了在公理中严格用于局部的 *ValueidD*、生成程序参数和局部用于捕捉递归定义的 *ErrorD* 之外）没有影响 *dict* 的内容。返回的描述符和 *dict* 之间的等价由前面提到的函数 *transform-system* 中的 **be such that** 构件完成。

算法

第 4 行 变换定义表中的第一个定义。

第 5 行 变换其余的定义。

第 6 行 第一个定义的名字和实体类（反映在 *Qual* 中）组成的偶对必须与其余定义的偶对不相同。

第 9 行 返回对字典 *Dict* 的贡献和 AS_1 定义。

transform-decl(*decl*)(*dict*) ≜ (3.4.2)

```
 1  ( cases decl:
 2    (mk-Blockdef0(,,,)
 3       → transform-blockdef(decl)(dict),
 4     mk-Chandef0(,,,,)
 5       → transform-channeldef(decl)(dict),
 6     mk-Prdef0(,,,,,,)
 7       → transform-processdef(decl)(dict),
 8     mk-Sigdef0()
 9       → transform-signaldef(decl)(dict),
10     mk-Procdef0(,,,,)
11       → transform-proceduredef(decl)(dict),
12     mk-Partialtypedef0(,,,,)
13       → transform-partial-typedef(decl)(dict),
14     mk-Syntypedef0(,,,,)
15       → transform-syntype(decl)(dict),
16     mk-Sortgenerator0(,,,,)
17       → transform-sortgenerator(decl)(dict),
18     mk-Synonymdef0(,,)
19       → transform-synonymdef(decl)(dict),
20     mk-Vardef0(,,)
21       → transform-vardef(decl)(dict),
22     mk-Viewdef0()
23       → transform-viewdef(decl)(dict),
24     mk-Importdef0()
25       → transform-importdef(decl)(dict),
26     mk-Sigroutedef0(,,)
27       → if is-BlockD(dict(dict(LEVEL)))
28            then transform-signalroutedef(decl)(dict)
29            else transform-servicesigroutedef(decl)(dict),
30     mk-Signallistdef0(,)
31       → transform-signallistdef(decl)(dict),
32     mk-Timerdef0()
33       → transform-timerdef(decl)(dict),
34     mk-Servicedef0(,,,,)
35       → build-service-descriptor(decl)(dict),
36     T → ({}, [])))
```

type: $Decl_0 \rightarrow Dict \rightarrow Decl_1$-**set** *Dict*

目的 把一个 AS_0 定义变换为一个 AS_1 定义

结果 参见变换声明函数 *transform-decllist*

算法 变换下列其中之一

第 2 行 一个功能块定义或

第 4 行 一个信道定义或

第 6 行 一个进程定义或

第 8 行 一个信号定义或

第 10 行 一个过程定义或

第 12 行 一个部分数据类型定义或

第 14 行 一个同义类型定义或

第 16 行 一个类别生成程序或

第 18 行	一个同义词定义或
第 20 行	一个变量定义或
第 22 行	一个视见定义或
第 24 行	一个进口定义或
第 26—29 行	根据包围作用域单位变换的一个信号路由定义或一个服务信号路由或
第 30 行	一个信号表定义或
第 32 行	一个定时器定义或
第 29 行	一个服务信号路由定义或
第 34 行	一个服务定义
第 19 行	不作处理，因为其它种类的定义（即连接）在别的地方处理。

3.4.1 功能块定义

transform-blockdef(**mk-***Blockdef*$_0$(*bid*, *decllist*, *subdef*, *tid*))(*dict*) $\triangleq$ (3.4.1.1)

```
 1  (let mk-Id0(q, bnm) = bid in
 2   let bqual = dict(SCOPEUNIT) ⌢ ⟨(BLOCK, bnm)⟩ in
 3   let dict' = dict + [SCOPEUNIT ↦ bqual] +
 4       [DATATYPEDEF ↦ initialdatadef(dict)] in
 5   let explicit = (∃d ∈ elems decllist)(is-Sigroutedef0(d)) in
 6   let (chandefl, routedefl, connects, exp, imp) =
 7       implicit-channels-and-signal-routes(decllist)(dict') in
 8   let (outerchannels, cdi) = transform-decllist(chandefl)(dict) in
 9   let (as1 dcl, di) = transform-decllist(decllist ⌢ routedefl)(dict') in
10   let (as1 connect, connectmap) = transform-block-connect(decllist, {}, [])(dict') in
11   if ((∃i ∈ elems decllist)(is-Prdef0(i)) ∨ subdef ≠ nil) ∧ q = ⟨⟩ ∧ tid ∈ {bid, nil} then
12     if subdef = nil then
13       (let descr = [bqual ↦ mk-BlockD(exp, imp, explicit, connectmap)] in
14        let as1 block = make-as1 tree(BLOCK, bnm, as1 dcl ⌢ as1 connect, nil, nil, nil)(dict) in
15        (outerchannels ∪ {as1 block}, descr + di + cdi))
16      else
17       (let mk-Blocksubdef0(subid, subdecll, tailid) = subdef in
18        let mk-Id0(q', bnm') = if subid = nil then mk-Id0(⟨⟩, bnm) else subid in
19        let chansubqual = dict(SCOPEUNIT) ⌢ ⟨(SUBSTRUCTURE, bnm')⟩ in
20        let subqual = if chansubqual ∈ dom dict then
21                        chansubqual
22                      else
23                        bqual ⌢ ⟨(SUBSTRUCTURE, bnm')⟩ in
24        let dict'' = [SCOPEUNIT     ↦ subqual,
25                      DATATYPEDEF ↦ initialdatadef(dict')] in
26        let (as1 subdcl, subdi) = transform-decllist(subdecll)(dict'') in
27        let as1 connect' = transform-substructure-connect(subdecll ⌢ connects, {}, [])(dict'') in
28        let as1 tree = make-as1 tree(SUBSTRUCTURE, bnm',
29                                     as1 subdcl ∪ as1 connect', nil, nil, nil)(subdi) in
30        let as1 block' = make-as1 tree(BLOCK, bnm, as1 dcl, as1 tree, nil, nil)(dict) in
31        let descr = [bqual ↦ mk-BlockD(exp, imp, explicit, connectmap)] +
32            (if chansubqual ∈ dom dict then [] else [subqual ↦ mk-BlocksubD()]) in
33        (q' ≠ ⟨⟩
34           → exit("§2.4.1: 定义的名字仅仅可以在间接定义中被限定 "),
35         tailid ∉ {nil, subid}
36           → exit("§2.2.2: 功能块子结构定义中的结尾名与定义名不同"),
37         ¬(∃d ∈ elems subdecll)(is-Blockdef0(d))
38           → exit("§3.2.2: 功能块子结构不包含功能块定义 "),
39         T → (outerchannels ∪ {as1 block'}, descr + di + subdi + cdi)))
40    else
41     (¬(∃i ∈ elems decllist)(is-Prdef0(i) ∨ subdef ≠ nil)
42        → exit("§2.4.3: 功能块必须包含一个或多个进程，或一个子结构定义"),
43      q ≠ ⟨⟩
44        → exit("§2.4.1: 定义的名字仅仅可以在间接定义中被限定 "),
45      T → exit("§2.2.2: 功能块定义中的结尾名与定义名不同")))
```

type: *Blockdef*$_0$ → *Dict* → *Decl*$_1$-**set** *Dict*

目的

把一个 AS$_0$ 功能块定义变换为一个 AS$_1$ 功能块定义

参数

AS$_0$ 功能块定义包含

bid 未加限定词的功能块标识符

decllist 它的定义表

subdef 它的任选的子结构

tid 它的任选的尾标识符

结果　　参见变换声明表函数 *transform-decllist*

算法

第 2—3 行　构成表示功能块标识符（*bqual*）的限定表 *Qual* 并更新字典 *Dict* 的作用域单位 **SCOPEUNIT** 项目来表示这个功能块的范围。构成这个功能块的初始的数据类型定义 $Data\text{-}type\text{-}definition_1$。

第 5 行　如果为这个功能块指定了显式信号路由，则令 *explicit* 为真。

第 6 行　创建隐式的 AS_0 信道定义，它是功能块的出口与进口所隐含的；还创建进出口信道描述 *ExpimpchanD* 映射表（见 *ExpimpchanD* 的域定义）它用于将信道定义接口到子结构和周围环境上。

第 8 行　将隐式信道定义变换为 AS_1。

第 9 行　变换所含的定义表，并更新 *dict* 的项目 **SCOPEUNIT** 以表示此功能块。

第 10 行　把信道变换成功能块中的信号路由连接，并返回功能块的连接描述 *BlockconnectionD*。

第 11 行　如果省略子结构，则功能块中必须存在至少一个进程定义。这个功能块标识符一定不能被限定。如果规定了结尾标识符，则它必须与功能块标识符相同。

第 12—15 行　如果功能块不包含一个功能块子结构，则返回隐式信道（*outerchannels*）和功能块（as_1-*block*）的 AS_1 定义以及隐式信道（*cdi*）、功能块（*descr*）和所含的定义（*di*）对字典 *Dict* 的贡献。

第 17 行　分解功能块子结构。

第 18 行　如果忽略该功能块子结构名，则它与这个功能块同名。令 *bnm'* 表示这个功能块子结构名。

第 19—20 行　令 *subqual* 表示子结构的限定表 *Qual*。如果从一个信道子结构中导出这个功能块子结构，则 *subqual* 表示信道子结构的 *Qual*。

第 24 行　与第 3 行中一样，更新作用域单位 **SCOPEUNIT** 和数据类型定义 **DATATYPEDEF**。

第 26 行　变换子结构的定义表。

第 27 行　创建功能块（as_1block）的 AS_1 定义。

第 28 行　从 *subdecll* 中的 AS_0 连接创建 AS_1 连接。

第 30 行　功能块子结构（as_1tree）的 AS_1 定义。

第 31—32 行　登记功能块（*bqual*）的 *Dict* 项目。也登记功能块子结构（*subqual*）的 *Dict* 项目，除非该子结构表示一个信道子结构。

第 33 行　这个功能块子结构名（标识符）一定不能限定（因为它不是一个间接定义）。

第 35 行　如果指定了结尾标识符，则它一定要与子结构标识符（*subid*）相同。

第 37 行　这个子结构必须包含至少一个功能块定义。

第 39 行　与第 15 行一样，返回 AS_1 定义和隐式信道对字典 *Dict* 的贡献（*cdi*），以及功能块和子结构对字典 *Dict* 的贡献（*descr*），功能块中所含的定义（*di*）和子结构中所含定义（*subdi*）对字典 *Dict* 的贡献。

第 41 行　在此功能块中必须存在至少一个进程定义，除非指定了一个子结构，并且

第 33 行　功能块标识符一定不要限定，并且

第 45 行　如果指定了结尾名，则它必须与功能块名相同。

initialdatadef(*dict*) ≙ **(3.4.1.2)**

```
1  (let mk-Data-type-definition1(unm, , , , ) = dict(DATATYPEDEF),
2       qual = dict(SCOPEUNIT) ↷ ⟨(TYPE, unm)⟩ in
3   let typeid = make-as1-identifier(qual)(dict) in
4   mk-Data-type-definition1(create-unique-name(), {typeid}, {}, {}, {}))
```

type: *Dict* → *Data-type-definition*$_1$

目的

为一个作用域单位构造 AS_1 数据类型定义，在这个作用域单位中把表示外围作用域单位的数据类型定义的标识符包括在类型的并中。在作用域单位中的部分数据类型定义的变换期间把字面值、运算符和等式增加到数据类型定义中。

算法

第 1 行　抽取外围作用域单位的 AS_1 数据类型定义。

第 2 行　构造外围作用域单位的类型标识符的 *Qual*。

第 3 行　构造此类型的 AS_1 标识符。

第 4 行　返回一个新的数据类型定义，这个定义有一个唯一的名字；在这个定义中，外围作用域单位的类型标识符被包括在类型的并中；这个定义不包含字面值、运算符或等式。

3.4.2 信道定义

transform-channeldef(**mk-***Chandef*$_0$(*cnm*, *chanpath*, *ochanpath*, *csub*, *tnm*))(*dict*) $\triangleq$ (3.4.2.1)

```
(let mk-Chanpath0(ori, dest, siglist) = chanpath in
 let orid = get-visible-qual(ori, BLOCK)(dict),
     destid = get-visible-qual(dest, BLOCK)(dict),
     sigidset = transform-signallist(siglist)(dict),
     sigidset' =
     if ochanpath = nil then
       {}
      else
       (let mk-Chanpath0(orig', dest', osiglist) = ochanpath in
        let orid' = get-visible-qual(orig', BLOCK)(dict),
            destid' = get-visible-qual(dest', BLOCK)(dict) in
        if orid = destid' ∧ destid = orid' then
          transform-signallist(osiglist)(dict)
         else
          exit("§2.5: 信道定义中的第二条路径必须和第一条路径方向相反 ")) in
 if orid = destid then
   exit("§2.5: 信道的两个端点必须是不同的 ")
  else
   (let as1orid = if ori = ENV then ENVIRONMENT else make-as1-identifier(orid)(dict),
        as1destid = if dest = ENV then
                      ENVIRONMENT
                    else
                     make-as1-identifier(destid)(dict),
        as1sigs = make-as1idset(sigidset)(dict),
        as1sigs' = make-as1idset(sigidset')(dict),
        cqual = dict(SCOPEUNIT) ↷ ⟨(CHANNEL, cnm)⟩ in
    (¬is-local(orid)(dict) ∨ ¬is-local(destid)(dict)
       → exit("§2.5: 信道的端点被定义在本信道以外的作用域单位中 "),
     tnm ∉ {cnm, nil}
       → exit("§2.2.2: 信道定义中的结尾名不同于定义名 "),
     csub ≠ nil
       → transform-channel-sub(csub, cnm, sigidset, sigidset', orid, destid)(dict),
     T → (let descr = [cqual ↦ mk-ChannelD(orid, destid, sigidset, sigidset', nil)] in
          let path1 = mk-Channel-path1(as1orid, as1destid, as1sigs),
              path2 = if as1sigs' = {} then
                        nil
                      else
                       mk-Channel-path1(as1destid, as1orid, as1sigs') in
          ({mk-Channel-definition1(name-to-name1(cnm), path1, path2)}, descr)))))
```

type: *Chandef*$_0$ → *Dict* → *Decl*$_1$-**set** *Dict*

目的 把一个信道定义变换为 AS_1

结果 是 AS_1 信道定义和一个对 *dict* 的贡献，包含信道描述符和信道子结构描述符（如果有的话）。

参数 AS_0 信道定义包含

cnm 信道名

chanpath 第一条路径

ochanpath 任选的第二条信道路径

csub 任选的信道子结构

tnm	结束定义的名字

算法

第 1 行	分解第一条路径
第 2—15 行	构造起始功能块（*orid*）、目的功能块（*destid*）、信号集合（*sigidset*）和在反方向传输的信号集合（*sigidset'*）的 *Qual*。
第 9—15 行	如果信道是双向的，则一个方向的起始端点必须是另一方向的终止端点。
第 16 行	起始功能块与目的功能块一定不能是同一个功能块。
第 19—25 行	构造起始功能块的标识符（as_1orid）、目的功能块的标识符（$as_1destid$）、信号标识符集合（$as_1sigidset$）和相反方向的信号标识符集合（$as_1sigidset'$）。
第 26 行	更新作用域单位 **SCOPEUNIT** 来表示此信道标识符。
第 27—28 行	这个起始功能块的定义和这个目的功能块的定义一定要与此信道的定义处于相同的层次上。而且
第 29 行	如果规定的话，结束信道定义的名字必须是此信道的名字。
第 31 行	如果有信道子结构，则变换此信道子结构。否则
第 33 行	构造此信道对 *Dict* 的贡献。
第 34—35 行	构造这两条 AS_1 路径。
第 39 行	返回 AS_1 信道定义和它对字典 *Dict* 的贡献。

transform-channel-sub(*csub*, *nm*, *sigset*, *osigset*, *endpoint1*, *endpoint2*)(*dict*) $\triangleq$ (3.4.2.2)

```
1  (let mk-Chansubdef0(subid, decllist, tailid) = csub in
2   let mk-Id0(q, nm') = if subid = nil then mk-Id0(⟨⟩, nm) else subid in
3   let mk-Id0(qtail, nmtail) = if tailid = nil then mk-Id0(q, nm') else tailid in
4   if q ≠ ⟨⟩ ∨ qtail ≠ ⟨⟩ then
5     exit("§2.4.1: 定义的名字仅仅可以在间接定义中被限定")
6    else
7     if nmtail ≠ nm' then
8       exit("§2.2.2: 在信道子结构定义中的结尾名不同于定义名")
9      else
10      (let level = dict(SCOPEUNIT) in
11       let newbnm = create-unique-name() in
12       let newbqual = level ⌢ ⟨(BLOCK, newbnm)⟩ in
13       let newc1 = create-unique-name(),
14           newc2 = create-unique-name() in
15       let path1 = mk-Chanpath0(as0-id(endpoint1), as0-id(newbqual), ⟨as0-id(q) | q ∈ sigset⟩),
16           opath1 =
17           if osigset = {}
18             then nil
19             else mk-Chanpath0(as0-id(newbqual), as0-id(endpoint1), ⟨as0-id(q) | q ∈ osigset⟩),
20           path2 = mk-Chanpath0(as0-id(newbqual), as0-id(endpoint2), ⟨as0-id(q) | q ∈ sigset⟩),
21           opath2 =
22           if osigset = {}
23             then nil
24             else mk-Chanpath0(as0-id(endpoint2), as0-id(newbqual), ⟨as0-id(q) | q ∈ osigset⟩) in
25       let chandef1 = mk-Chandef0(newc1, path1, opath1, nil, newc1),
26           chandef2 = mk-Chandef0(newc2, path2, opath2, nil, newc2) in
27       let substructurequal = level ⌢ ⟨(SUBSTRUCTURE, nm')⟩ in
28       let dict' = dict + [SCOPEUNIT ↦ substructurequal] in
29       let decllist' = replace-connects(decllist, newc1, endpoint1, newc2, endpoint2)(dict') in
30       let newchannels = (endpoint1, newc1, endpoint2, newc2) in
31       let blockdef =
32           mk-Blockdef0(mk-Id0(⟨⟩, newbnm), ⟨⟩, mk-Blocksubdef0(mk-Id0(⟨⟩, nm'), decllist', nil), nil) in
33       let channelqual = level ⌢ ⟨(CHANNEL, nm)⟩ in
34       let di = [channelqual      ↦ mk-ChannelD(endpoint1, endpoint2, sigset, osigset, newchannels),
```

```
35              substructurequal ↦ mk-ChannelsubD(newbqual)] in
36      let (as1 declset, dict') = transform-decllist((chandef1, chandef2, blockdef))(dict) in
37      (as1 declset, dict' + di)))
```

type: *Chansubdef*$_0$ *Name*$_0$ *Signalqual*-**set** *Signalqual*-**set** *Endpoint Endpoint* → *Dict* → *Decl*$_1$-**set** *Dict*

目的

把一个信道子结构定义变换为两个信道定义和一个功能块定义

参数

csub	AS_0 信道子结构定义
nm	包围的信道定义的名字
siglist	由包围的信道传送的那些信号
osiglist	任选的指向相反方向的那些信号
*endpoint*1	是一个功能块，从这个功能块中发送出 *siglist*。
*endpoint*2	是一个功能块，信号 *siglist* 发送到这个功能块。*osiglist* 信号从 *endpoint*2 发送到 *endpoint*1。

结果

参见变换声明表函数 *transform-decllist*

算法

第 1 行	令 *subid* 表示信道子结构标识符，令 *decllist* 表示包围在子结构中的定义表，令 *tailid* 表示结束这个子结构定义的标识符。
第 2 行	如果没有为子结构指定标识符（名字），就继承外围信道的名字。
第 3 行	令 *qtail* 表示结尾标识符（*subid*）中的限定词，而令 *nmtail* 表示结尾标识符中的名字。
第 4 行	子结构的名字一定不要限定（因为它不是一个间接定义）；如果规定的话，结束这个定义的标识符必须与开始这个定义的标识符相同。
第 7 行	结尾名必须与信道子结构名相同。
第 10 行	令 *level* 表示定义此信道的功能块的限定表 *Qual*。
第 11 行	为被构造的功能块起一个单独的新名字。
第 12 行	为被构造的功能块构造字典 *Dict* 项目。
第 13—14 行	为被构造的两个信道各起一个新名字。
第 15—16 行	为第一个新信道构造两个方向的路径。
第 20—21 行	为第二个新信道构造两个方向的路径。
第 25—26 行	构造这两个新的 AS_0 信道。
第 29 行	由信道子结构的定义表中的功能块子结构的连接（*Connect*$_0$）替换信道端点连接（*Connect*$_0$）。
第 31 行	构造此新功能块。它有一个空的定义表和一个含有修改过的定义表的功能块子结构。
第 34 行	令 *di* 表示字典 *Dict*，它由包围信道的描述符和信道子结构的描述符组成。
第 36 行	把这两个构造好的信道定义和功能块定义变换为 AS_1。
第 37 行	返回这三个 AS_1 定义和字典 *Dict*，该 *Dict* 包括这三个定义的描述符、所含定义的描述符、包围信道的描述符和信道子结构的描述符。

replace-connects(*decllist*, *newc*1, *endpoint*1, *newc*2, *endpoint*2)(*dict*) ≙ **(3.4.2.3)**

```
1  (let conset = {d ∈ elems decllist | is-Connect0(d)} in
2   if card conset = 2 then
3     (let {mk-Connect0(blkid1, clist1), mk-Connect0(blkid2, clist2)} = conset in
4      let bqual1 = get-visible-qual(blkid1, BLOCK)(dict),
5          bqual2 = get-visible-qual(blkid2, BLOCK)(dict) in
6      if {bqual1, bqual2} = {endpoint1, endpoint2} then
7        (let (c1, c2) = if bqual1 = endpoint1 then (newc1, newc2) else (newc2, newc1) in
8         let decllist' = ⟨decllist[i] | i ∈ ind decllist ∧ ¬is-Connect0(decllist[i])⟩ in
9         decllist' ⌢
10        ⟨mk-Connect0(mk-Id0(⟨⟩, c1), clist1),
11         mk-Connect0(mk-Id0(⟨⟩, c2), clist2)⟩)
12       else
13        exit("§3.2.3: 在连接中的功能块标识符不表示信道端点"))
14    else
15     exit("§3.2.3: 必须正好是两个信道端点的连接"))
```

type: $Decl_0^*$ *Channame*$_0$ *Endpoint* *Channame*$_0$ *Endpoint* → *Dict* → $Decl_0^+$

目的

把一个信道子结构定义表变成一个功能块子结构定义表，使得这个表含有信道的连接而不是信道端点的连接。

参数

decllist	要被改变的定义表
*newc*1，*newc*2	两个综合信道的名字，要用它们取代包含一信道子结构的信道
*endpoint*1，*endpoint*2	是包含一个信道子结构的信道的两个端点。信道 *newc*1 与端点 *endpoint*1 相连，信道 *newc*2 与端点 *endpoint*2 相连。

结果

新的定义表，将被包围在一个综合的功能块子结构中

算法

第 1 行	从定义表中抽取信道的端点的连接。
第 2 和 13 行	必须正好是两个这样的信道端点的连接。
第 3 行	分解包含这两个信道端点的连接的集合。
第 4—5 行	为在两个信道端点的连接中提到的功能块标识符构造限定符表 *Quals*。
第 6 行	这两个功能块标识符必须与信道定义中提到的功能块标识符相同。
第 7 行	令 *c*1 表示 *clist*1 对应的信道的名字，用 *c*2 表示 *clist*2 对应的信道的名字。
第 8 行	构造定义表，表中的两个信道端点的连接已被删除。并且
第 9—11 行	返回与两个综合信道连接拼接后的这个定义表。

3.4.3 进程定义

transform-processdef(**mk-***Prdef*$_0$(*pid*, *ins*, *pl*, *sigl*, *decll*, *body*, *tid*))(*dict*) $\triangleq$ **(3.4.3.1)**

```
1  (let mk-Id0(q, name) = pid in
2   let mk-Instances0(ini, maxi) = if ins = nil then mk-Instances0(nil, nil) else ins in
3   let ini' = if ini = nil then 1 else eval-simple-expr(ini, "INTEGER")(dict),
4       max' = if maxi = nil then infinite else eval-simple-expr(maxi, "INTEGER")(dict) in
5   (q ≠ ⟨⟩
6      → exit("§2.4.1: 定义的名字仅仅可以在间接定义中被限定"),
7    tid ∉ {pid, nil}
8      → exit("§2.2.2: 进程定义中的结尾名不同于定义名"),
9    ini' > max'
10     → exit("§2.2.3: 实例的初始数大于实例的最大数"),
11   ini' < 0
12     → exit("§2.2.3: 实例的初始数小于0"),
13   max' = 0
14     → exit("§2.2.3: 实例的最大数等于0"),
15   T → (let d = [IMPLIED      ↦ {},
16                 OUTSIGNALS   ↦ {},
17                 SCOPEUNIT    ↦ dict(SCOPEUNIT) ⌢ ⟨(PROCESS, name)⟩,
18                 DATATYPEDEF  ↦ initialdatadef(dict)] in
19        let (as1 list, pd, dict') = transform-processparm(pl)(dict + d),
20            sigset = transform-validinputset(sigl)(dict + d),
21            (as1 declset, ddict) = transform-decllist(decll)(dict + d) in
22        if dom dict' ∩ dom ddict ≠ {} then
23          exit("§2.2.2: 形式参数的名字必须区别于变量名")
24        else
25          (let (servicedeclset1, as1 body, bdict, connectmap) =
26               transform-process-body(body)(dict + d),
27               outsigs = bdict(OUTSIGNALS),
28               number = mk-Number-of-instances1(ini', max') in
29           if (∃q1, q2 ∈ outsigs ∪ sigset)(len q1 > len q2 ∧ ⟨q1[i] | 1 ≤ i ≤ len q2⟩ = q2) then
30             exit("§3.3: 进程使用相同信号在不同具体化层次上的信号")
31           else
32             (let (as1 declset', vdict) = make-implicit-vardecl(bdict),
33                  as1 totset = as1 declset ∪ as1 declset' ∪ servicedeclset1,
34                  as1 tree =
35                  make-as1 tree(PROCESS, name, as1 totset, number, as1 list, as1 body)(ddict) in
36              let delem = [d(SCOPEUNIT) ↦ mk-ProcessD(pd, sigset, outsigs, connectmap)] +
37                  dict' + ddict + vdict in
38              ({as1 tree}, delem))))))
```

type: *Prdef*$_0$ → *Dict* → *Process-definition*$_1$-**set** *Dict*

目的 把一个进程定义变换为 AS$_1$

参数 AS$_0$ 进程定义包含

- *pid* 进程标识符
- *ins* “实例数” 构件
- *pl* 形式参数表
- *sigl* 有效的输入信号集合
- *decll* 所含的定义表
- *body* 进程状态体
- *tid* 结尾标识符

结果 参见变换声明表函数 *transform-decllist*

算法

第 1 行 令 *name* 表示此进程的名字。

第 2 行　　如果省略了“实例数”构件，那么就等于是省略了初始实例数和省略了最大实例数。

第 3—4 行　　如果没有规定初始实例数，就让它等于 1。如果没有规定最大实例数，那么就允许无穷多个实例。预定义常数 *infinite* 表示一个“不受限制的”数（也参见附件 F.1 的 5.8 节）。如果规定了一个表达式，就求出它的值，而且这个值必须是预定义的数据类别 INTEGER；即，整数 *Qual*（*iqual*）作为类别集里唯一的有效类别，用作传给简单表达式求值函数 *eval-simple-expr* 的参数。

第 5 行　　进程的名字（标识符）一定不能限定（因为它不是一个间接定义）。

第 7 行　　如果规定的话，结束这个进程定义（*tid*）的标识符必须与进程名（标识符）相同。

第 9 行　　最大实例数必须大于或等于初始的实例数。

第 11 行　　初始的实例数一定不能小于零。

第 13 行　　最大实例数一定不能是零。

第 15—16 行　　初始化 **IMPLIED** 项目，使它不包含隐式变量；初始化 **OUTSIGNALS**，使它不包含输出信号。在进程体的变换期间把项目填入这两个集合。

第 17—18 行　　更新作用域单位 **SCOPEUNIT** 来表示进程的层次并构造进程的初始数据类型定义 *Data-type-definition*$_1$。

第 19 行　　变换进程的形式参数。

第 20 行　　抽取完整的有效输入信号集合。

第 21 行　　变换所含的定义。

第 22 行　　形式参数的名字必须不同于进程中定义的变量名和同义词名（即，它们必须有不同的限定表 *Quals*）。

第 25 行　　变换进程体并返回从所含服务（如果有的话）得到的 AS$_1$ 定义表、AS$_1$ 进程图、字典 *Dict*（其中只用到 **IMPLIED** 项目和 **OUTSIGNALS** 项目）、以及信号路由和服务信号路由之间的关系（如果进程中没有服务，则 *connectmap* 是空的）。

第 29—30 行　　一定不允许存在两个输出信号，其中第一个信号在第二个信号中定义。对于输入信号也必须如此。

第 32 行　　构造隐含的 AS$_1$ 变量定义。

第 33—38 行　　返回 AS$_1$ 进程定义（*as$_1$tree*）和对字典 *Dict* 的贡献。对字典 *Dict* 的贡献由进程描述符（*delem*）、形式参数描述符（*dict'*）和在进程内被定义的实体描述符（*ddict*）组成。

transform-validinputset(*sigset*)(*dict*) ≜ (3.4.3.2)

```
1  (let qual = dict(SCOPEUNIT) in
2   let mk-BlockD(,, explicit,) = dict(get-sur(process-level(qual))) in
3   let routesigset =
4       if ¬explicit then
5         {}
6        else
7         {squal | (∃mk-SignalrouteD(p1, p2, s1, s2) ∈ rng dict)
8                  ((p1 = qual ∧ squal ∈ s2) ∨ (p2 = qual ∧ squal ∈ s1))} in
9   let squalset =
10      if sigset = nil then
11        {}
12       else
13        (let mk-Inputset0(sigl) = sigset in
14         if sigl = nil then {} else transform-signallist(sigl)(dict)) in
15  let implicit =
16      if is-ServiceD(dict(qual)) then
17        {}
18       else
19        (let (expimpinput,) = import-export-signals(dict) in
20         expimpinput) in
21  let timers = {tqual ∈ dom dict | is-TimerD(dict(tqual)) ∧ is-local(tqual)(dict)} in
22  let locals = {tqual ∈ dom dict | is-SignalD(dict(tqual)) ∧ is-local(tqual)(dict)} in
23  if squalset ∩ timers = {} ∧ locals ⊆ squalset then
24    routesigset ∪ squalset ∪ implicit ∪ timers
25   else
26    exit("§2.5.2: 有效输入信号必须包含所有局部信号并且不包含定时器"))
```

type: $[Signallist_0] \rightarrow Dict \rightarrow Signalqual\text{-}set$

目的

为一个进程或为一个服务抽取完整的有效输入信号集合

参数

sigset 任选的 AS_0 信号标识符表

算法

第 1 行 令 *qual* 表示此服务或进程的限定表 *Qual*。

第 2 行 如果为外围功能块指定了显式的信号路由，则 *explicit* 为真。

第 3—8 行 令 *routeset* 表示指向该进程或服务的信号路由（隐式的或显式的）中包含的信号集合。如果没有为外围功能块指定显式的信号路由，这个集合就是空的。

第 9—14 行 令 *squalset* 表示此进程或服务的有效输入信号集合构件中包含的信号。

第 15—19 行 如果作用域单位不是一个服务就抽取指向这个作用域单位的隐含信号的限定表 *Qual*。

第 21 行 令 *timers* 表示在此进程或服务中定义的定时器信号。

第 22 行 令 *locals* 表示在此进程或服务中定义的信号。

第 23 行 此进程或服务的有效输入信号集合一定不包括任何定时器，并且它必须包含所有的局部定义的信号。

第 24 行 完整的有效输入信号集合由信号路由中的信号、有效输入信号集合构件中的信号、隐式信号和定时器组成。

transform-processparm(*pl*)(*dict*) ≙ (3.4.3.3)

```
1  (if pl = ⟨⟩ then
2     (⟨⟩, ⟨⟩, [])
3   else
4    (let mk-Parm0(nml, tid) = hd pl in
5     let (as1 list, tp', d') = transform-processparm(tl pl)(dict) in
6     let as0 vardef = mk-Vardefelem0(nml, tid, nil) in
7     let (,d) = transform-vardef(mk-Vardef0(nil, nil, ⟨as0 vardef⟩))(dict) in
8     let tq = get-visible-qual(tid, TYPE)(dict) in
9     let tq' = get-parent(tq)(dict) in
10    let tql = ⟨tq' | 1 ≤ i ≤ len nml⟩ in
11    let as1 id = make-as1-identifier(tq')(dict) in
12    let as1 nml = ⟨name-to-name1(nml[i]) | 1 ≤ i ≤ len nml⟩ in
13    if card elems as1 nml ≠ len as1 nml then
14      exit("§2.2.2: 在相同作用域单位中的两个定义使用相同的名字")
15     else
16      (let as1 tree = mk-Process-formal-parameter1(as1 nml, as1 id) in
17       (⟨as1 tree⟩ ⌢ as1 list, tql ⌢ tp', d + d'))))
```

type: $Parm_0^* \rightarrow Dict \rightarrow Process\text{-}formal\text{-}parameter_1^*\ Sortqual^*\ Dict$

目的

把一个进程的形式参数表变换到 AS_1

参数

pl　　AS_0 进程形式参数表

结果

AS_1 形式参数、对应的类别表（该表用于检查创建请求节点中的实在参数）和包含形式参数描述符对字典 *Dict* 的贡献

算法

函数递归地遍历形式参数表。

第 1 行　当全部处理完时，返回空表和空映射表。

第 4 行　令 *nml* 表示表中第一个元素的名字表，令 *tid* 表示他们的类别。

第 5 行　变换其余的形式参数。

第 6 行　为手头现有参数构造一个 AS_0 变量定义，并且

第 7 行　变换这个变量定义。

第 8 行　构造此变量类别的限定表 *Qual*

第 9 行　如果是同义类型则抽取其父辈

第 10 行　构造与变量表对应的类别 *Qual* 表（*tql*）。

第 11 行　为变量表（*nml*）的类别构造父辈部分类型（*tq'*）的 AS_1 标识符（$as_1 id$）

第 12 行　把此名字表变换为一个 AS_1 名字表

第 13 行　表中的名字必须是不同的

第 16 行　构造手头现有（*nml*）参数的 AS_1 形式参数定义。

第 17 行　与其余形式参数一起返回 AS_1 形式参数定义，返回有关的类别表并返回它们对字典 *Dict* 的贡献。

3.4.4 信号定义

transform-signaldef(**mk-***Sigdef*$_0$(*elemlist*))(*dict*) $\triangleq$ **(3.4.4.1)**

```
 1  (let mk-Sigelem0(nm, tidl, refinement) = hd elemlist in
 2   let dlev = dict + [SCOPEUNIT ↦ dict(SCOPEUNIT) ⌢ ⟨(SIGNAL, nm)⟩] in
 3   let (as1 refinement, subsig1, subsig2, d) =
 4       if refinement = nil then
 5         (nil, {}, {}, [])
 6        else
 7         (let mk-Refinement0(subsiglist) = refinement in
 8          let (as1 set, s1, s2, d') = transform-refinement(subsiglist)(dlev) in
 9          (mk-Signal-refinement1(as1 set), s1, s2, d')) in
10   let quall = ⟨get-visible-qual(tidl[i], TYPE)(dict) | 1 ≤ i ≤ len tidl⟩ in
11   let d' = [dict(SCOPEUNIT) ⌢ ⟨(SIGNAL, nm)⟩ ↦ mk-SignalD(quall, subsig1, subsig2)] in
12   let (as1 set', d'') =
13       if tl elemlist = ⟨⟩
14         then ({}, [])
15         else transform-signaldef(mk-Sigdef0(tl elemlist))(dict) in
16   if dom d' ∩ dom d'' ≠ {} then
17     exit("§2.2.2: 在相同作用域单位中的两个定义使用相同的名字")
18    else
19     (let as1 idlist = ⟨make-as1-identifier(quall[i])(dict) | i ∈ ind quall⟩ in
20      let as1 tree = mk-Signal-definition1(name-to-name1(nm), as1 idlist, as1 refinement) in
21      ({as1 tree} ∪ as1 set', d + d' + d'')))
```

type: *Sigdef*$_0$ → *Dict* → *Signal-definition*$_1$-**set** *Dict*

目的　把一个信号定义变换到 AS_1

参数　组合的 AS_0 信号定义，它引入基本信号定义表（*elemlist*）

结果　参见变换声明表函数 *transform-decllist*

算法

第 1 行　分解组合的 AS_0 信号定义中的第一个信号定义

第 2 行　更新层次以表示此信号的结构。

第 3—9 行　变换信号定义的具体化部分并返回所含的 AS_1 定义（as_1 *refinement*）、与定义的信号方向相同的子信号（*subsig*1）、与定义的信号方向相反的子信号（*subsig*2）和从新定义得到的对字典 *Dict* 的贡献（*d*）。

第 10 行　从规定的 AS_0 类别标识符表构造类别限定表 *Qual*

第 11 行　构造此信号对字典 *Dict* 的贡献。

第 12—15 行　变换组合信号定义中的其余的信号定义。

第 16 行　在组合定义中不允许两个信号有相同的名字。

第 19 行　构造 AS_1 类别标识符表。

第 20—21 行　返回 AS_1 信号定义集合和从这些信号得到的对字典 *Dict* 的贡献。

transform-refinement(*decllist*)(*dict*) ≜ (3.4.4.2)

```
1  (if decllist = ⟨⟩ then
2     ({},{},{},[])
3   else
4    (let (as1 declrest, sig1rest, sig2rest, drest) = transform-refinement(tl decllist)(dict) in
5     let mk-Subsignal0(reverse, sigdef) = hd decllist in
6     let (as1 decl, d) = transform-signaldef(sigdef)(dict) in
7     if dom d ∩ dom drest ≠ {} then
8       exit("§2.2.2: 在相同作用域单位和相同实体类中的两个定义定义相同的名字")
9      else
10      (let as1 tot = as1 declrest ∪ {mk-Subsignal-definition1(reverse, decl) | decl ∈ as1 decl} in
11       if reverse = nil then
12         (as1 tot, sig1rest ∪ dom d, sig2rest, d + drest)
13        else
14         (as1 tot, sig1rest, sig2rest + dom d, d + drest))))
```

type: $Subsignal_0^* \to Dict \to Subsignal\text{-}definition_1\text{-}\mathbf{set}\ Signalqual\text{-}\mathbf{set}\ Signalqual\text{-}\mathbf{set}\ Dict$

目的

把信号具体化中的子信号变换成 AS_1 子信号定义。

参数

decllist　AS_0 子信号定义表

结果

包括下列各项：AS_1 子信号定义的集合、包含与父辈信号方向相同的信号的信号限定表 *Qual* 的集合、包含与父辈信号方向相反的信号的信号限定表 *Qual* 的集合（这两个集合在变换信号定义函数 *transform-signaldef* 中被放入父辈信号的描述符中）和子信号对字典 *Dict* 的贡献。

算法

第 1 行　当全部处理完时，不返回任何贡献。

第 4 行　变换其余的子信号。

第 5 行　分解表中的第一个子信号。

第 6 行　变换此子信号，把它看成是一个普通信号。

第 7 行　检查子信号定义是否唯一。

第 10 行　从 AS_1 信号定义构造 AS_1 子信号定义的集合，并且

第 11—14 行　返回这个集合。如果规定了 REVERSE 则把信号的限定表 *Qual*（取自于 *d* 的域）加到第一个 *Signalqual* 集合中，否则把它们加到第二个 *Signalqual* 集合中。还返回子信号对字典 *dict* 的贡献（*d*）和其余子信号对字典 *dict* 的贡献（*drest*）。

3.4.5 过程定义

transform-proceduredef(**mk-***Procdef*$_0$(*pid*, *pl*, *decll*, *body*, *tid*))(*dict*) ≜ (3.4.5.1)

```
1  (let mk-Id0(q, name) = pid in
2   if q = ⟨⟩ ∧ tid ∈ {pid, nil} then
3     (let name' = if is-ServiceD(dict(dict(SCOPEUNIT))) then
4                    create-unique-name()
5                  else
6                    name in
7      let nqual = dict(SCOPEUNIT) ⌢ ⟨(PROCEDURE, name')⟩ in
8      let d = [SCOPEUNIT  ↦ dict(SCOPEUNIT) ⌢ ⟨(PROCEDURE, name)⟩,
```

```
 9              DATATYPEDEF ↦ initialdatadef(dict)] in
10     let (as1 list, pd, dict') = transform-procedureparml(pl)(dict + d),
11         (as1 declset, ddict) = transform-decllist(decll)(dict + d),
12         (mk-Procedure-graph1(sta, stateset), bdict) = transform-body(body)(dict + d) in
13     if dom dict' ∩ dom ddict ≠ {} then
14       exit("§2.2.2: 形式参数的名字必须不同于变量的名字")
15      else
16       if (∃squal ∈ dom dict)(is-ServiceD(dict(squal)) ∧
17                              get-sur(squal) = process-or-service-level(dict(SCOPEUNIT))) ∧
18         stateset ≠ {}
19         then exit("§4.10.2: 在分解或服务中的过程具有状态或进口表达式")
20         else (let as1 tree =
21                 make-as1 tree(PROCEDURE, name', as1 declset,
22                               as1 list, mk-Procedure-graph1(sta, stateset), nil)(ddict) in
23               let delem = [d(SCOPEUNIT) ↦ mk-ProcedureD(pd, nqual)] + dict' + ddict in
24               ({as1 tree}, delem + bdict)))
25    else
26     (q ≠ ⟨⟩
27        → exit("§2.4.1: 定义的名字仅仅可以在间接定义中被限定"),
28      T → exit("§2.2.2: 过程定义中的结尾名不同于定义名")))
```

type: $Procdef_0 \rightarrow Dict \rightarrow Procedure\text{-}definition_1\text{-}\mathbf{set}\ Dict$

目的

把一个过程定义变换到 AS_1

参数

过程定义包括

pid 过程标识符（必须只含一个名字）

pl 形式参数表

decll 定义表

body 状态体

tid 结束此定义的标识符（名字）

结果

仅由一个定义组成的 AS_1 定义集合和此过程及其所含定义对字典 *dict* 的贡献

算法

第 1 行 令 *name* 表示过程名

第 2 行 过程名（标识符）一定不能限定（因为它不是一个间接定义），如果规定的话，则结束这个过程定义的名字（标识符）一定要与此过程的名字相同。

第 3 行 如果这个过程定义出现在一个服务中，就要为它构造一个唯一的名字。

第 7 行 构造变量的限定表 *Qual* 以便用于 AS_1 中

第 8—9 行 更新 **SCOPEUNIT** 以表示此过程的作用域单位，构造过程的初始的数据类型定义 *Data-type-definition*$_1$。

第 10 行 变换过程的形式参数以便得到 AS_1 参数表（as_1list）、它们的类别描述符（在过程被引用时要用到的）和它们对字典 *Dict* 的贡献。

第 11 行 变换所含的定义

第 12 行 变换过程体并返回 AS_1 过程图和一个字典 *Dict*，在此 *Dict* 中仅使用 **IMPLIED** 项目和 **OUTSIGNALS** 项目。

第 13 行 形式参数的名字不可以与在过程中定义的变量或同义词的名字相同。

第 16—19 行 如果在一个进程中或在一个进程内的过程中定义了一个包含有状态或进口的过程，则这个进程一定不能包含服务。

第 20—24 行　　返回 AS_1 过程定义（as_1tree）和对字典 *Dict* 的贡献，它们由此过程描述符、形式参数描述符与过程中定义的那些实体的描述符（*bdict*）组成。

transform-procedureparml(*pl*)(*dict*) ≜ **(3.4.5.2)**

```
1  (if pl = ⟨⟩ then
2     (⟨⟩, ⟨⟩, [])
3   else
4    (let (as1 elems, tpelems, delems) =
5          cases hd pl:
6           (mk-Inoutparm0(varlist, tid)
7              → transform-varparm(true, varlist, tid)(dict),
8            mk-Inparm0(varlist, tid)
9              → transform-varparm(false, varlist, tid)(dict)) in
10   let (as1 list, tplist, d) = transform-procedureparml(tl pl)(dict) in
11   if dom delems ∩ dom d ≠ {} then
12     exit("§2.2.2: 在相同作用域单位中的两个定义使用相同的名字")
13    else
14     (as1 elems ⌢ as1 list, tpelems ⌢ tplist, delems + d)))
```

type: $Procparm_0^* \to Dict \to Procedure\text{-}formal\text{-}parameter_1^*\ FormparmD^*\ Dict$

目的　　把一个过程的形式参数表变换到 AS_1

参数

pl　　AS_0 的形式参数表

结果　　是 AS_1 形式参数、这些参数的描述符（*FormparmD**），（这些描述符被放入过程描述符 *ProcedureD* 中并用于检查实在参数的类型）最后是这些参数对字典 *Dict* 的贡献。

算法

第 1 行　　当全部处理完时，返回空表和空字典 *Dict*。

第 4—9 行　　变换第一个参数，这个参数或者是一个 **INOUT** 变量参数（第 6 行），或者是一个 **IN** 变量参数（第 8 行）。

第 10 行　　变换剩下的形式参数表

第 11 行　　*pl* 中引入的名字必须是不同的

第 14 行　　返回由第一个形式参数和其余部分组成的实物。

transform-varparm(*isout*, *varlist*, *tid*)(*dict*) ≙ (3.4.5.3)

```
 1  if card elems varlist ≠ len varlist then
 2    exit("§2.2.2: 在相同作用域单位中的两个定义使用相同的名字")
 3   else
 4    (let tqual = get-visible-qual(tid, TYPE)(dict) in
 5     let tqual' = get-parent(tqual)(dict) in
 6     let as1 varlist = ⟨name-to-name1(varlist[i]) | 1 ≤ i ≤ len varlist⟩ in
 7     let as1 tree = if isout then
 8                      mk-Inout-parameter1(as1 varlist, make-as1-identifier(tqual)(dict))
 9                    else
10                     mk-In-parameter1(as1 varlist, make-as1-identifier(tqual)(dict)),
11         parmdescrlist = if isout then
12                           ⟨mk-InoutDescr(tqual) | 1 ≤ i ≤ len varlist⟩
13                         else
14                           ⟨mk-InDescr(tqual') | 1 ≤ i ≤ len varlist⟩ in
15     let nmlist =
16         ⟨if is-ServiceD(dict(dict(SCOPEUNIT)))
17            then create-unique-name()
18            else varlist[i] | 1 ≤ i ≤ len varlist⟩,
19         newquallist = ⟨dict(SCOPEUNIT) ⌢ ⟨(VALUE, nmlist[i])⟩ | 1 ≤ i ≤ len varlist⟩ in
20     let vquall = ⟨dict(SCOPEUNIT) ⌢ ⟨(VALUE, varlist[i])⟩ | 1 ≤ i ≤ len varlist⟩,
21         delem = [vquall[i] ↦ mk-VarD(tqual, nil, nil, nil, newquallist[i]) | 1 ≤ i ≤ len varlist] in
22     (⟨as1 tree⟩, parmdescrlist, delem))
```

type: *Bool* $Name_0^*$ Id_0 → *Dict* → *Procedure-formal-parameter*$_1^+$ *FormparmD*$^+$ *Dict*

目的

把一个形式参数变换到 AS_1

参数

isout 为真，如果形式参数是一个 INOUT 参数。

tarlist 这些形式参数的名字

tid 它们的类别

结果

参见变换过程参数表函数 *transform-procedureparml*

算法

第 1 行 形式参数中的名字必须是不同的

第 4 行 抽取参数类别的限定表 *Qual*。

第 5 行 抽取此类别的父辈部分类型。

第 6 行 把名字表中的名字变换成 AS_1 名字

第 7—10 行 构造 AS_1 形式参数。

第 11—14 行 构造实在参数类别描述符。

第 21 行 构造形式参数对字典 *dict* 的贡献。所含的描述符是 *VarD* 描述符，(象普通变量的描述符一样) 因为在静态性质方面形式变量参数相当于普通变量。

3.4.6 类别生成程序

transform-sortgenerator(**mk-***Sortgenerator*$_0$(*nm*, *parml*, *geninstl*, *prop*, *tnm*))(*dict*) $\triangleq$ **(3.4.6.1)**

```
1  (if tnm ∈ {nm, nil} ∧ gen-formparm-unique(parml, {nm}, {}, {}) then
2     (let gqual = dict(SCOPEUNIT) ↷ ⟨(GENERATOR, nm)⟩ in
3      let prop' = transform-geninst(nm, geninstl, prop)(dict + [gqual ↦ mk-ErrorD()]) in
4      let descr = mk-GeneratorD(parml, prop') in
5      ({}, [gqual ↦ descr]))
6    else
7     exit("§5.4.1.12: 在生成程序定义中的结尾名不同于定义名"))
```

type: *Sortgenerator*$_0$ → *Dict* → *Decl*$_1$-**set** *Dict*

目的 构造对应于一个数据类别生成程序的字典 *Dict* 的贡献。不返回 AS$_1$ 实物。

参数 一个 AS$_0$ 数据类别生成程序包含

nm	生成程序名
parml	形式参数表
geninstl	所含的生成程序实例表
prop	通常的性质
tnm	结束这个定义的名字

结果 为了方便起见，此函数返回一个 AS$_1$ 定义集合，尽管这个表总是空的（见变换声明表函数 *transform-decllist*）。也返回此数据类别生成程序对字典 *Dict* 的贡献。

算法

第 1 行 如果规定的话，结束定义的名字必须与生成程序名相同，并且在各种参数类中的形式参数必须都是唯一的。

第 2 行 构造生成程序的限定表 *Qual*。

第 3 行 变换生成程序定义中所包含的生成程序实例。生成程序的 qual（*gqual*）用一个 *ErrorD* 来表示，这样使得递归能够被检测出来。

第 4—7 行 返回包括生成程序形式参数表（*parml*）和生成程序体（*prop*）对字典 *Dict* 的贡献。在生成程序体被生成程序实例构件（不在这个地方）替换之后要把良形式性运用到体上。

gen-formparm-unique(*parml*, *sset*, *lset*, *oset*) $\triangleq$ **(3.4.6.2)**

```
1  if parml = ⟨⟩ then
2    true
3   else
4    cases hd parml:
5     (mk-Sortparm0(nmlist)
6        → card elems nmlist = len nmlist ∧ elems nmlist ∩ sset = {} ∧
7          gen-formparm-unique(tl parml, sset ∪ elems nmlist, lset, oset),
8      mk-Termparm0(nmlist),
9      mk-Litparm0(nmlist)
10       → card elems nmlist = len nmlist ∧ elems nmlist ∩ lset = {} ∧
11         gen-formparm-unique(tl parml, sset, lset ∪ elems nmlist, oset),
12     mk-Opparm0(nmlist)
13       → card elems nmlist = len nmlist ∧ elems nmlist ∩ oset = {} ∧
14         gen-formparm-unique(tl parml, sset, lset, oset ∪ elems nmlist))
```

type: *Genparm*$_0$ *Name*$_0$-**set** *Name*$_0$-**set** *Name*$_0$-**set*** → *Bool*

目的　　检查生成程序的各形式参数是否都是唯一的

参数

parml　　形式参数表

sset, *lset*, *oset*　　分别为类别参数的集合、字面值或项参数的集合和运算符参数集合。这些集合是迄今为止所处理的（函数是递归的）。

结果　　如果成功的话，结果为真。

算法

第 5—7 行　　对一个类别参数来说，所含各名字必须是唯一的，并且这些名字不允许出现在另一个类别参数中；生成程序其余的形式参数也必须是唯一的。

第 8—11 行　　对于一个项参数或字面值参数，所含的名字必须是唯一的，并且这些名字不允许出现在另一个项参数或字面值参数中；生成程序其余的形式参数也必须是唯一的。

第 12—14 行　　对于一个运算符参数，所含的名字必须是唯一的，并且这些名字不允许出现在另一个运算符参数中；生成程序的其余的形式参数也必须是唯一的。

3.4.7　类别定义

transform-partial-typedef(mk-*Partialtypedef*$_0$(*nm*, *extprop*, *prop*, *vlist*, *tnm*))(*dict*) ≜　　(3.4.7.1)

```
1  (if tnm ∈ {nm, nil} then
2     (let tqual = dict(SCOPEUNIT) ⌢ ⟨(TYPE, nm)⟩ in
3      let (implicitdecl, delem) = make-implicit-sigdecls(tqual)(dict) in
4      if (∃opqual ∈ dom dict)((is-OperatorD(dict(opqual)) ∨ is-LiteralD(dict(opqual))) ∧
5                              s-Result(dict(opqual)) = tqual ∧
6                              get-sur(get-sur(opqual)) = dict(SCOPEUNIT)) then
7        if vlist = nil then
8          (let dict' = cases extprop:
9                       (nil
10                        → transform-sortdef(nm, prop, nil)(dict),
11                      mk-Struc$_0$()
12                        → transform-struc(nm, extprop, prop)(dict),
13                      mk-Inherited$_0$(, , )
14                        → transform-inherited(nm, extprop, prop)(dict),
15                      mk-Geninstlist$_0$(instl)
16                        → (let prop' = transform-geninst(nm, instl, prop)(dict) in
17                           transform-sortdef(nm, prop', nil)(dict))) in
18           (implicitdecl, dict'))
19         else
20          (let unm = create-unique-name() in
21           let (, dnew) =
22               transform-partial-typedef(mk-Partialtypedef$_0$(unm, extprop, prop, nil, unm))(dict),
23               id = mk-Id$_0$(dict(SCOPEUNIT), unm) in
24           let (as$_1$ syn, dsyn) =
25               transform-syntype(mk-Syntypedef$_0$(nm, id, vlist, nil, tnm))(dict) in
26           (implicitdecl ∪ as$_1$ syn, delem + dnew + dsyn))
27       else
28        exit("§5.2.1: 不存在一个运算符，它返回那个类别的一个值"))
29    else
30     exit("§5.2.1: 在部分类型定义中的结尾名不同于定义名"))
```

type:　*Partialtypedef*$_0$ → *Dict* → *Decl*$_1$-**set** *Dict*

目的 把一个部分数据类型定义变换到 AS_1

参数 数据类别定义包含

nm	类别名
extprop	扩展的性质
prop	性质（字面值、运算符、等式等）
vlist	在同义类型情况下表示一个值表
tnm	结束这个定义的名字

结果 参见变换声明表函数 *transform-decllist*

算法

第 1 行	如果规定的话，结束数据类别定义的名字必须与此数据类别的名字相同。
第 2 行	构造表示此类别标识符的限定表 *Qual*。
第 3 行	构造 AS_1 定义，它规定了附属于被定义类别的所有 IMPORT - EXPORT 变量的隐式信号。
第 4 行	必须有一个运算符或字面值，它
第 5 行	以这个类别作为结果类别，并且它
第 6 行	定义在外围作用域单位中。
第 7 行	如果省略了 CONSTANTS 构件，则它是一个实际的类别定义
第 8 行	构造字典 *Dict* 的贡献，它包括类别描述符、运算符和字面值描述符以及更新了的数据类型定义 $Data\text{-}type\text{-}definition_1$。此 AS_0 类别定义或者是
第 9 行	一个（简单）类别定义（没有扩展的性质）。或
第 11 行	是一个结构定义，或
第 13 行	是一个基于继承的类别，或
第 15 行	是一个生成程序实例（的序列）。在这种情形下，这些实例在用普通方法变换（第 17 行）之前就要进行扩展（第 15 行）。
第 20—26 行	如果规定了 CONSTANTS，则它是一个同义类型定义，这意味着
第 20 行	生成了一个唯一的隐式类别名，并且
第 21 行	变换那个部分类型定义 $Partialtypedef_0$，它不带有任何 CONSTANTS，但却带有那个新的名字，并且
第 24 行	变换同义类型定义 $Syntypedef_0$，它带有 CONSTANTS（*vallist*）和作为父辈的隐式类别

$$make\text{-}implicit\text{-}sigdecls(tqual)(dict) \triangleq \qquad (3.4.7.2)$$

```
1  (let qset = {(tq, ) ∈ dom dict | tq = tqual} in
2   transform-decllist({make-implicit-decl(tqual, dict(t)) | t ∈ qset})(dict))
```

type: $Sortqual \to Dict \to Decl_1\text{-set}\ Dict$

目的 构造 AS_1 信号定义，它附属于所有包含类别 *tqual* 的名字闭包描述符 *NameclosureD*（见出口映射表 *Exportmap* 的定义）。

算法

第 1 行	构造其类别限定符 *Sortqual* 等于 *tqual* 的名字闭包描述符 *NameclosureD* 的集合，并且
第 2 行	构造 AS_0 定义表，它通过联合附属于该集合内的每一个元素的定义表构成，并把这个表变换成 AS_1。

make-implicit-decl(*tqual*, mk-*SignalnamesD*(, *xqnm*, *xrnm*)) ≜ (3.4.7.3)

```
1  (let as0tid = as0-id(tqual) in
2   mk-Sigdef0(⟨mk-Sigelem0(xqnm, ⟨⟩, nil),
3               mk-Sigelem0(xrnm, ⟨as0tid⟩, nil)⟩))
```

type: *Sortqual SignalnamesD* → *Sigdef*$_0$

目的 构造附属于某个名字和某个类别的 EXPORT - IMPORT 的 AS_0 信号定义（*tqual*）

算法

第 1 行 构造由 xtREPLY 信号带有的类别的 AS_0 标识符

第 2—3 行 构造信号定义，包含

第 2 行 xtQUERY 信号和

第 3 行 xtREPLY 信号。

transform-inherited(*nm*, mk-*Inherited*$_0$(*pid*, *lrenam*, *ops*), *prop*)(*dict*) ≜ (3.4.7.4)

```
1  (let pqual = get-inherited-parent( et-visible-qual(pid, TYPE)(dict))(dict) in
2   if is-recursive-sort(pqual, {})(dict) then
3     exit("§5.4.1.11: 类别基于自身")
4    else
5     (let pqual' = s-Newqual(dict(pqual)) in
6      let sortdict = transform-sortdef(nm, prop, pqual)(dict) in
7      let typedef = sortdict(DATATYPEDEF) in
8      let mk-Data-type-definition1(typename, union, sorts, sigs, eqs) = typedef in
9      let iqual be s.t. iqual ∈ dom sortdict ∧ is-SortD(sortdict(iqual)) in
10     let mk-SortD(eqs', pq, exp1, nqual) = sortdict(iqual) in
11     let (litmap, litd) = transform-literal-renaming(lrenam, iqual, pqual)(dict) in
12     let concaxioms1 = make-as1-concaxioms(dom litd)(dict) in
13     let (opmap, opd, opset) = transform-operator-renaming(ops, iqual, pqual)(dict) in
14     if dom sortdict ∩ dom litd ∩ dom opd ≠ {} then
15       exit("§5.2.2: 用显式定义和继承性定义两种方式来定义运算符或字面值")
16      else
17       (let inhax = extract-inherited-axioms(litmap, opmap, nqual, pqual')(dict) in
18        let iid1 = make-as1-identifier(nqual)(dict) in
19        let litsig = {mk-Literal-signature1(nm1, iid1) | mk-Identifier1(, nm1) ∈ rng litmap} in
20        let datatypedef' =
21            mk-Data-type-definition1(typename, union, sorts,
22                                     sigs ∪ litsig ∪ opset, eqs ∪ inhax ∪ concaxioms1) in
23        let sortdescr = mk-SortD(eqs' ∪ inhax ∪ concaxioms1, pq, exp1, nqual) in
24        (sortdict + litd + opd + [iqual          ↦ sortdescr,
25                                  DATATYPEDEF ↦ datatypedef'])))
```

type: *Name*$_0$ *Inherited*$_0$ *Properties*$_0$ → *Dict* → *Dict*

目的 变换一个基于继承的部分类型定义

参数

nm 被定义的类别名

Inherited AS_0 继承构件

prop ADDING 构件中定义的 AS_0 性质。

结果 是字典 *Dict*，在这个 *Dict* 中 DATATYPEDEF 项目已被更新以包括被定义的类别，而且类别和

运算符的描述符已被加到该 *Dict* 中。

算法

第 1 行　抽取表示这个父辈类别标识符的限定表 *Qual*。

第 2—3 行　这个类别一定不能自己继承自己

第 5 行　抽取父辈唯一的限定表 *Qual*

第 6 行　用被定义的类别和用它的 ADDING 性质来更新数据类型定义 **DATATYPEDEF** 项目。

第 8 行　分解更新过的数据类型定义 *Data-type-definition*$_1$。

第 9—10 行　抽取被定义类别的限定表 *Qual*（刚加上去的）并分解它。

第 11 行　把字面值再命名部分变换为一个映射表 *litmap* 并构造字面值再命名中引入的字面值名字的描述符。

第 12 行　从字面值再命名中引入的任何字符串字面值构造隐含的公理。

第 13 行　把运算符再命名部分变换成一个映射表 *opmap* 并构成新运算符名的描述符（*opd*）和运算符的 AS$_1$ 标记（*opset*）。这个函数也处理隐含继承的运算符（不可见的运算符）。

第 14—15 行　所有在附加部分 *sortdict* 中定义的运算符和字面值以及在继承部分定义的运算符（*opd*）和字面值（*litd*）都必须有不同的标记。

第 17 行　抽取所有的公理，这些公理使用了隐含继承的运算符。

第 18 行　构造所定义的类别的 AS$_1$ 标识符。

第 19 行　构造再命名的字面值的 AS$_1$ 标记。

第 20—23 行　把在数据类型定义 *Data-type-definition* 和此类别描述符中的继承部分的性质包括进来。

第 24 行　返回对字典 *Dict* 的贡献，它包括附加部分的（*sortdict*）、再命名字面值的（*litd*）、再命名运算符的（*opd*）、类别描述符的和更新过的数据类型定义 *Data-type-definition*$_1$ 的。

is-recursive-sort(*qual*, *qset*)(*dict*) ≙ **(3.4.7.5)**

```
1  (if qual ∈ qset then
2     true
3   else
4    cases dict(qual):
5     (mk-SortD(, pa, , )
6        → if pa = nil then false else is-recursive-sort(pa, qset ∪ {qual})(dict),
7      mk-SyntypeD(pa, , , )
8        → is-recursive-sort(pa, qset ∪ {qual})(dict)))
```

type: *Sortqual Sortqual*-**set** → *Dict* → *Bool*

目的　检查一个同义类型或一个继承类别是不能够基于它自身的。本函数递归地遍历通过该 *dict*，直到找到一个部分类型描述符不基于继承为止（即，直到符合假条件为止）。

参数

qual　表示当前的类别

qset　表示已经引用过的类别的集合。当开始运用此函数时，这个集合是空的。

结果　如果此类别递归地定义，则结果为真。

算法

第 1 行　　如果当前的类别（*qual*）已引用过，则为真；否则

第 5—6 行　　如果当前的类别是一个没有父辈的部分类型，则这个类别不是递归地定义的，否则把此部分类型加入到这个类别集合中，并且继续处理它的父辈。

第 7 行　　把当前的同义类型加入到这个类别集合中并且继续处理它的父辈。

transform-literal-renaming(*lrenam*, *qual*, *pqual*)(*dict*) ≙ **(3.4.7.6)**

```
1  (let plitset = {lq ∈ dom dict | is-LiteralD(dict(lq)) ∧ s-Sortqual(dict(lq)) = pqual} in
2   let defaultmap = [qual ↦ qual | qual ∈ plitset] in
3   let litmap = defaultmap + build-literal-renaming(lrenam, qual, pqual, plitset) in
4   ([make-as1-identifier(lq)(dict) ↦ make-as1-identifier(litmap(lq))(dict) | lq ∈ dom litmap],
5    [lq ↦ mk-LiteralD(qual) | lq ∈ rng litmap]))
```

type: *Literalrenaming*$_0$ *Sortqual Sortqual* → *Dict* →
(*Literal-operator-identifier*$_1$ ⤖ *Literal-operator-identifier*$_1$) *Dict*

目的　　构造一个映射表，从父辈字面值映射到新字面值。当要抽取所继承的公理时要用到这些新字面值(在抽取继承公理函数 *extract-inherited-axioms* 中)，并构造新字面值的 *Dict* 描述符。

参数

lrenam　　AS$_0$ 字面值再命名

qual　　定义的类别的限定表 *Qual*

pqual　　继承性所基于的类别的限定表 *Qual*

结果　　构成的映射和包含字面值描述符的对字典 *Dict* 的贡献

算法

第 1 行　　从字典 *Dict* 中抽取，为父辈类别定义的字面值的集合。

第 2 行　　构造缺省的映射表；即，适用于一些字面值的关系，在再命名部分没有提到这些字面值。

第 3 行　　通过重写那个缺省映射表来构造字面值再命名的映射表（在这个映射表中，所含的字面值是限定表 *Qual*）。

第 4 行　　在把 *Qual* 转换为 AS$_1$ 标识符之后返回字面值再命名的映射表，并返回

第 5 行　　字面值描述符（*literalDs*），它们全都包含其类别的限定表 *Qual*。

build-literal-renaming(*lrenam*, *qual*, *pqual*, *plitset*) ≙ **(3.4.7.7)**

```
1  if lrenam = ⟨⟩ then
2    []
3   else
4    (let mk-Literalpair0(nlit, olit) = hd lrenam in
5     let nqual = qual ⌢ ⟨(LITERAL, nlit)⟩,
6         oqual = pqual ⌢ ⟨(LITERAL, olit)⟩ in
7     if oqual ∈ plitset then
8       (let restmap = build-literal-renaming(tl lrenam, qual, pqual, plitset) in
9        if nqual ∈ rng restmap then
10         exit("§5.4.1.11: 在重新命名中两次定义的字面值")
11       else
12        if oqual ∈ dom restmap then
13          exit("§5.4.1.11: 字面值被重新命名两次")
14         else
15          restmap + [oqual ↦ nqual])
16      else
17       exit("§5.4.1.11: 在字面值重新命名中的字面值没有在父辈类别中定义"))
```

type: *Literalrenaming*$_0$ *Sortqual Sortqual Qual*-**set** → (*Qual* ⇸ *Qual*)

目的

从 AS$_1$ 再命名部分构造一个映射表，从父辈的限定表 *Qual* 映射到新字面值名字的限定表 *Qual*

参数

lrenam	AS$_0$ 字面值的再命名
qual	定义的类别的限定表 *Qual*
pqual	继承性所基于的类别的限定表 *Qual*
plitset	限定表 *Qual* 的集合，包含为父辈类别定义的所有的字面值

结果

构造的字面值的映射表

算法

第 1 行	在全部处理完时，什么也不返回（本函数是递归的）。
第 4 行	分解在字面值的再命名中的下一对字面值名。
第 5—6 行	构造新字面值名的限定表 *Qual*（*nqual*）和父辈（旧的）字面值名的限定表 *Qual*（*oqual*）。
第 7 行	在字面值偶对 *literalpair*$_0$ 中右边的字面值必须是父辈类别的字面值。
第 8 行	构造字面值的再命名的其余部分的映射表。
第 9—12 行	如果左边字面值（第 9 行）或右边字面值（第 12 行）是在字面值再命名的其余部分的映射之中，则它在字面值再命名中再命名两次
第 15 行	否则返回余下的字面值再命名的字面值映射表，表中已经增加了这一对字面值的贡献

transform-operator-renaming(*oprenam*, *iqual*, *pqual*)(*dict*) ≜ (3.4.7.8)

```
1  (let opset = {qual ∈ dom dict | is-OperatorD(dict(qual)) ∧
2                                  (let mk-OperatorD(argl, res, , ) = dict(qual) in
3                                   pqual ∈ elems argl ∨ pqual = res)} in
4   let explicitset = {qual ∈ opset | s-Explicit(dict(qual))} in
5   let allnameset = {name | (∃q ∈ explicitset)((, (name, , )) = q[len q])} in
6   let oprenam' = if oprenam = ALL then ⟨mk-Operatorpair0(nil, nm) | nm ∈ allnameset⟩ else oprenam in
7   let (opmap, opd) = build-operator-renaming(oprenam', iqual, pqual, explicitset)(dict) in
8   let (opmap', opd') = build-implicit-operators(iqual, pqual, opset \ explicitset)(dict) in
9   let (qmap, opdict) = (opmap + opmap', opd + opd') in
10  let idmap = [make-as1-identifier(qual)(dict) ↦ make-as1-identifier(qmap(qual))(dict) |
11               qual ∈ dom qmap] in
12  let as1typing = {make-as1-typing(op)(dict) | op ∈ dom opdict} in
13  (idmap, opdict, as1typing))
```

type: (ALL | *Operatorrenaming*$_0$) *Sortqual Sortqual* → *Dict* →
(*Operator-identifier*$_1$ $\overrightarrow{m}$ *Operator-identifier*$_1$) *Dict Operator-signature*$_1$-**set**

目的

从 AS$_1$ 运算符再命名部分构造一个映射表，从一些父辈运算符标识符映射到这个被定义类别的一些运算符标识符。该映射表包括隐含继承的运算符的标识符（如果一个运算符在其标记中包括父辈类别，它就被隐含地继承了）

参数

oprenam	AS$_0$ 运算符的再命名
iqual	被定义类别的限定表 *Qual*
pqual	父辈的限定表 *Qual*

结果

构造好的运算符映射表

算法

第 1 行	从字典 *Dict* 构造运算符限定表 *Qual* 的集合，其中父辈限定表 *Qual*（*pqual*）应出现在这个类别的变元表（*argl*）中或者是这个运算符的结果。
第 4 行	构造 *opset* 的子集合，它包含那些被显式地定义（或继承）的运算符。
第 5 行	为被显式定义的运算符构造运算符名字的集合。
第 6 行	如果规定了 ALL，则令 *oprenam'* 表示 *Operatorrenaming*$_0$，它包括所有的显示运算符。否则令 *oprenam'* 等于 *oprenam*。
第 7 行	为显式运算符构造运算符的 *Qual* 的映射表和字典 *Dict* 的描述符。
第 8 行	为隐式运算符构造运算符的 *Qual* 的映射表和字典 *Dict* 的描述符。
第 9 行	完整的运算符的 *Qual* 的映射表是 *qmap*，完整的字典 *Dict* 的贡献是 *opdict*。
第 10 行	把运算符的 *Qual* 的映射表转换为一个运算符标识符映射表。
第 12 行	为对字典 *Dict* 的贡献中的每个运算符构造 AS$_1$ 运算符标记。
第 13 行	返回运算符标识符映射表、对字典 *Dict* 的贡献和这些运算符的 AS$_1$ 标记。

make-as$_1$*-typing*(*qual*)(*dict*) ≙ **(3.4.7.9)**

```
1  (let mk-OperatorD(sl, res, qual', ) = dict(qual) in
2   let (, (nm, , )) = qual[len qual'] in
3   let sl_1 = ⟨make-as_1-identifier(sl[i])(dict) | 1 ≤ i ≤ len sl⟩,
4       res_1 = make-as_1-identifier(res)(dict) in
5   mk-Operator-signature_1(name-to-name_1(nm), sl_1, res_1))
```

type: *OperatorD* → *Dict* → *Operator-signature*$_1$

目的

从一个运算符的限定表 *Qual* 构造这个运算符的 AS_1 标记

参数

qual 运算符的限定表 *Qual*

结果

构造好的运算符标记

算法

第 1 行 分解运算符描述符。*qual'* 表示新的限定符 *Newqual*，它包含 AS_1 中使用的名字

第 2 行 通过分解限定表 *Qual* 中末尾的元素来抽取运算符名。对于一个运算符，末尾的元素总是一个 *Operatorqualelem*（见此域）。

第 3 行 构造表示变元类别的 AS_1 标识符表。

第 4 行 构造表示结果类别的 AS_1 标识符。

第 5 行 返回组合的运算符标记

build-operator-renaming(*renlist*, *qual*, *pqual*, *opset*)(*dict*) ≙ **(3.4.7.10)**

```
1  (if renlist = ⟨⟩ then
2     ([], [])
3   else
4    (let mk-Operatorpair_0(nop, oop) = hd renlist in
5     let oset = {oqual ∈ opset | (let (, (nm, , )) = oqual[len oqual] in
6                                  get-sur(oqual) = pqual ∧
7                                  nm = oop)} in
8     if oset = {} ∨ (is-Name_0(oop) ∧ (let mk-Name_0(, exc) = oop in
9                                       exc ≠ nil)) then
10      if oset = {} then
11        exit("§5.4.1.11: 在运算符重新命名中的运算符没有在父辈类别中定义")
12       else
13        exit("§5.4.1.11: 带有感叹号的运算符在一个运算符重新命名中被描述")
14      else
15       (let (maprest, drest) =
16            build-operator-renaming(tl renlist, qual, pqual, opset \ oset)(dict) in
17        let (map, d) = rename-operator(qual, pqual, oset, nop)(dict) in
18        if dom d ∩ dom drest = {} then
19          (map + maprest, d + drest)
20         else
21          exit("§5.4.1.11: 运算符重新命名两次"))))
```

type: *Operatorrenaming*$_0$ *Sortqual Sortqual Qual-set* → *Dict* → (*Qual* ⇀ *Qual*) *Dict*

目的

从一个运算符的再命名构造一个映射表，从父辈运算符限定表 *Qual* 映射到这个被定义类别的运算符限定表 *Qual*。还构造这些新运算符对字典 *Dict* 的贡献。

参数

renlist	AS_0 运算符再命名表
qual	被定义类别的限定表 *Qual*
pqual	父辈类别的限定表 *Qual*
opset	包含所有被显式定义的运算符的限定表 *Qual* 的集合

结果　　构造好的运算符限定表 *Qual* 的映射和对字典 *Dict* 的贡献

算法

第 1 行	当全部处理完时，什么都不返回（本函数是递归的）。
第 4 行	分解再命名表中的下一个再命名偶对。
第 5 行	抽取运算符的限定表 *Qual* 的集合，定义此运算符的地方的外围作用域单位就是父辈的 *Qual*，并且它的名字部分等于在重新命名中的父辈运算符名。
第 8—11 行	在这个集合中至少必须有一个 *Qual*。
第 8—13 行	父辈运算符名一定不能包含一个感叹号。
第 15 行	为再命名表的其余部分构造运算符映射表和对 *Dict* 的贡献。
第 17 行	重新命名手头上的运算符。*map* 表示对运算符映射表的贡献，*d* 表示对字典 *Dict* 的贡献。
第 18—19 行	如果手头上的运算符名的一些限定表 *Qual*（此运算符名可以有几个 *Qual*，因为在运算符再命名中提到的父辈运算符名可以表示几个（超载的）运算符，所以此运算符名可以有几个 *Qual*）不在再命名表的其余部分的对字典 *Dict* 的贡献中，则返回手头上的运算符名的运算符映射表的贡献（*map*）和再命名表的其余部分的运算符映射表的贡献，并返回相应的对字典 *Dict* 的贡献。

$rename\text{-}operator(iqual, pqual, oset, nop)(dict) \triangleq$ **(3.4.7.11)**

```
1  if oset = {} then
2    ([], [])
3   else
4    (let qual ∈ oset in
5     let (, (nm, , )) = qual[len qual] in
6     let mk-OperatorD(arglist, result, , ) = dict(qual) in
7     let arglist' = ⟨if get-parent(arglist[i])(dict) = pqual then iqual else arglist[i] | 1 ≤ i ≤ len arglist⟩ in
8     let parglist = ⟨get-parent(arglist'[i])(dict) | 1 ≤ i ≤ len arglist'⟩ in
9     let result' = if get-parent(result)(dict) = pqual then iqual else result in
10    let presult = get-parent(result')(dict) in
11    let (maprest, drest) = rename-operator(iqual, pqual, oset \ {qual}, nop)(dict) in
12    if result = result' ∧ arglist = arglist' then
13      (maprest, drest)
14     else
15      (let nm' = if nop = nil then nm else nop in
16       let newnm = create-unique-name() in
17       let opqual = iqual ⌢ ⟨(OPERATOR, (nm', parglist, presult))⟩ in
18       let newqual = iqual ⌢ ⟨(OPERATOR, (newnm, parglist, presult))⟩ in
19       (maprest + [iqual ↦ newqual], drest + [opqual ↦ mk-OperatorD(arglist', result', newqual, true)])))
```

type: $Sortqual\ Sortqual\ Qual\text{-}\mathbf{set}\ Newoperator_0 \to Dict \to (Qual \xrightarrow{m} Qual)\ Dict$

目的　　构造一个映射表，从父辈运算符的限定表 *Qual* 映射到一个给定了运算符名的运算符的限定表 *Qual*。还构造表示这个名字的运算符对字典 *Dict* 的贡献。

参数

iqual	继承类别的限定表 *Qual*

pqual	父辈类别的限定表 *Qual*
oset	父辈类别中定义的运算符的限定表 *Qual* 的集合，这些运算符的名字是在运算符再命名中提到的。
nop	新的运算符名

结果 构造好的映射表和相应的对字典 *Dict* 的贡献

算法

第 1 行	当全部处理完时，什么都不返回（本函数是递归的）。
第 4—5 行	在集合中取出下一个运算符限定表 *Qual* 并抽取这个运算符名。
第 6 行	分解当前的父辈运算符的运算符描述符。
第 7 行	用继承的类别来替换当前的运算符变元类别表中父辈类别的每一次出现。注意，函数 *get-parent* 与继承的类别的父辈没有关系。*get-parent* 用同义类型的（父辈）类别来取代此同义类型。
第 8 行	从变元表中删除任何同义类型。这个标定的变元表用于运算符的 *Qual* 中。
第 9—10 行	对结果类别的处理与此相同。
第 11 行	遍历父辈运算符的限定表 *Qual* 的集合的其余部分。
第 12—13 行	如果在变元表中或运算符的结果中没有用到父辈的 *Qual*，则返回这个表的其余部分的映射和对字典 *Dict* 的贡献。否则
第 15 行	如果在 $Operatorpair_0$ 中没有规定新的运算符名，则被继承的运算符与父辈运算符有相同的名字。
第 16 行	构造一个唯一的名字，用于运算符的 *Newqual* 中（见 *OperatorD* 的定义）。
第 17 行	为此运算符构造其限定表 *Qual*，用作字典 *Dict* 的项目。
第 18 行	构造运算符的 *Newqual*。
第 19 行	返回运算符的 *Qual* 映射，包括当前运算符的和其余运算符的，并且返回对字典 *Dict* 的贡献。

$build\text{-}implicit\text{-}operators(iqual, pqual, opset)(dict) \triangleq$ **(3.4.7.12)**

```
1  if opset = {} then
2    ([],[])
3   else
4    (let qual ∈ opset in
5     let (,(,arglist,result)) = qual[len qual] in
6     let arglist' = ⟨if get-parent(arglist[i])(dict) = pqual then iqual else arglist[i] | 1 ≤ i ≤ len arglist⟩ in
7     let result' = if get-parent(result)(dict) = pqual then iqual else result in
8     let (restmap, drest) = build-implicit-operators(iqual, pqual, opset \ {qual})(dict) in
9     if arglist = arglist' ∧ result = result' then
10      (restmap, drest)
11     else
12      (let newnm = create-unique-name() in
13       let opqual = iqual ↷ ⟨(OPERATOR, (newnm, arglist', result'))⟩ in
14       (restmap + [qual ↦ opqual], drest + [opqual ↦ mk-OperatorD(arglist', result', opqual, false)])))
```

type: *Sortqual Sortqual Qual*-**set** → *Dict* → (*Qual* $\overset{m}{\rightarrow}$ *Qual*) *Dict*

目的 构造一个映射表，从父辈运算符的 *Qual* 映射到隐式运算符的 *Qual*。还构造这些运算符对字典 *Dict* 的贡献。

参数

iqual 继承类别的限定表 *Qual*

pqual 父辈类别的限定表 *Qual*

oset 使用父辈类别的运算符限定表 *Qual* 的集合

结果 构造好的映射和相应的对字典 *Dict* 的贡献

算法

第 1 行 当全部处理完时，什么都不返回（该函数是递归的）。

第 4 行 在集合中取出一个运算符限定表 *Qual* 并抽取变元类别表和结果类别。

第 6—7 行 构造变元类别表，表中父辈类别的每一次出现由继承类别（*iqual*）来替换并且对结果类别作同样的处理。

第 8 行 为在 *opset* 中其余的运算符构造 *Qual* 映射和对字典 *Dict* 的贡献。

第 9—10 行 如果在标记中没有用 *pqual*，则返回构造好的实物。

第 12—13 行 为隐含运算符建造一个新的唯一的名字，并构造包含这个名字的限定表 *Qual*。

第 14 行 返回在 *opset* 中的其余运算符的 *Qual* 映射表和对字典 *Dict* 的贡献，并返回此运算符的映射表的贡献和它对字典 *Dict* 的贡献

$$\textit{extract-inherited-axioms}(\textit{litmap}, \textit{opmap}, \textit{qual}, \textit{pqual})(\textit{dict}) \triangleq \qquad \textbf{(3.4.7.13)}$$

```
1  (let qualset = all-visible-sorts(dict) \ {qual} in
2   let id1 = make-as1-identifier(qual)(dict) in
3   let pid1 = make-as1-identifier(pqual)(dict) in
4   let axset = union {s-Equations1(dict(qual)) | qual ∈ qualset} in
5   let axset' = {convert-axiom(axiom, litmap, opmap, id1, pid1) | axiom ∈ axset} in
6   axset' \ axset)
```

type: $(\textit{Literal-operator-identifier}_1 \xrightarrow{m} \textit{Literal-operator-identifier}_1)$
$(\textit{Operator-identifier}_1 \xrightarrow{m} \textit{Operator-identifier}_1)\ \textit{Sortqual}\ \textit{Sortqual} \rightarrow$
$\textit{Dict} \rightarrow \textit{Equations}_1$

目的 抽取所有继承的公理；即，所有使用了某些运算符或字面值的公理。这些运算符或字面值在其标记中含有父辈类别。

参数

litmap 一个映射表，从多个父辈字面值标识符映射到为一个继承类别定义的多个字面值标识符

opmap 一个映射表，从一些父辈运算符标识符映射到为一个继承类别定义的（隐式的或显式的）一些运算符标识符

qual 继承类别的限定表 *Qual*

pqual 父辈类别的限定表 *Qual*

结果 使用任何父辈运算符（或字面值）的等式的集合。在这个集合中，父辈运算符（或字面值）标识符已被继承类别的运算符（或字面值）标识符替换。

算法

第 1 行 构造包围作用域单位中可见的类别限定表 *Qual* 的集合。把继承类别排除在这个集合之外。

第 2—3 行 分别构造继承类别和父辈类别的 AS_1 标识符。

第 4 行 构造各种（可见的）类别中定义的等式集合的并集。

第 5 行　　构造在一个可见类别中定义的所有等式组成的集合，并且转换每一个等式使之包括新的运算符（或字面值）而不是父辈运算符（或字面值）。

第 6 行　　返回这个集合，但排除那些还没有改变的等式。

convert-axiom(*axiom*, *lmap*, *omap*, *id*, *pid*) ≙ (3.4.7.14)

```
1   cases axiom:
2   (mk-Unquantified-equation1(t1, t2)
3       → mk-Unquantified-equation1(convert-term(t1, lmap, omap), convert-term(t2, lmap, omap)),
4    mk-Quantified-equations1(nms, sort, eqs)
5       → mk-Quantified-equations1(nms, if sort = pid then id else sort,
6                                  {convert-axiom(eq, lmap, omap, id, pid) | eq ∈ eqs}),
7    mk-Conditional-equation1(eqs, eq)
8       → mk-Conditional-equation1({convert-axiom(a, lmap, omap, id, pid) | a ∈ eqs},
9                                  convert-axiom(eq, lmap, omap, id, pid)),
10   T → axiom)
```

type: *Equation*$_1$ (*Literal-operator-identifier*$_1$ $\overrightarrow{m}$ *Literal-operator-identifier*$_1$)
(*Operator-identifier*$_1$ $\overrightarrow{m}$ *Operator-identifier*$_1$) *Sort-identifier*$_1$ *Sort-identifier*$_1$ → *Equation*$_1$

目的　　修改一个继承的公理，以便包括由一个继承类别定义的运算符和字面值

参数

axiom　　要修改的公理

lmap　　一个映射表，从一些父辈字面值标识符映射到为一个继承类别定义的一些字面值标识符

omap　　一个映射表，从一些父辈运算符标识符映射到为一个继承类别定义的（隐式的或显式的）一些运算符标识符

id　　继承类别的 AS_1 标识符

pid　　父辈类别的 AS_1 标识符

结果　　修改过的公理（等式）

算法

第 2 行　　如果等式是一个简单的非量化的等式，则转换左边项（*t*1）和右边项（*t*2）。

第 4—6 行　　如果等式是一个量化的等式，则替换量化中的类别，如果它是父辈类别的话，并转换量化的等式中所含的等式。

第 7—9 行　　如果等式是个条件等式，则转换限制条件（第 8 行）并转换被限制的等式（第 9 行）。

第 10 行　　非形式的正文维持不变。

convert-term(*term*, *lmap*, *omap*) ≜ **(3.4.7.15)**

```
 1  if is-Error-term1(term) then
 2    term
 3   else
 4   (let t = cases term:
 5            (mk-Ground-term1(te)     → te,
 6             mk-Composite-term1(te) → te) in
 7    let t' = cases t:
 8            (mk-Identifier1(,)
 9               → if is-Ground-term1(term) ∧ t ∈ dom lmap then lmap(t) else t,
10             mk-Conditional-term1(t1, t2, t3)
11               → mk-Conditional-term1(convert-term(t1, lmap, omap),
12                                      convert-term(t2, lmap, omap),
13                                      convert-term(t3, lmap, omap)),
14             T → (let (opid, arglist) = t in
15                  let arglist' = ⟨convert-term(arglist[i], lmap, omap) | 1 ≤ i ≤ len arglist⟩ in
16                  if opid ∈ dom omap then
17                    (omap(opid), arglist')
18                  else
19                    (opid, arglist'))) in
20    if is-Ground-term1(term) then
21      mk-Ground-term1(t')
22     else
23      mk-Composite-term1(t'))
```

type: $Term_1$ ($Literal\text{-}operator\text{-}identifier_1 \overset{m}{\rightarrow} Literal\text{-}operator\text{-}identifier_1$)
($Operator\text{-}identifier_1 \overset{m}{\rightarrow} Operator\text{-}identifier_1$) → $Term_1$

目的

修改一个项，以便包括由一个继承类别定义的运算符和字面值

参数

term 要修改的项

lmap 一个映射表，从一些父辈字面值标识符映射到为一个继承类别定义的一些字面值标识符

omap 一个映射表，从一些父辈运算符标识符映射到为一个继承类别定义的（隐式的或显式的）一些运算符标识符

结果

修改过的项

算法

第 1 行 如果这是错误项，则不作处理。

第 4—6 行 分解这个项。进行替换处理，不管它是一个基项还是一个合成的项。

第 7 行 如果这个项包含一个标识符并且这个项是一个基项，则这个标识符表示一个字面值；如果它是在字面值映射表之中就把它替换掉。

第 10—13 行 如果这个项包含一个条件项，则转换其布尔项、“then”项和“else”项。

第 14—19 行 如果该项是一个运算符的应用，则

第 14 行 分解这个运算符的应用。

第 15 行 转换这些变元项。

第 16—19 行 如果此运算符标识符在运算符映射表之中，则从映射表（*omap*）中取出新继承的运算符标识符，否则不改变此运算符标识符。再把运算符的应用组合起来。

第 20—23 行 重新组合这个项（与第 4—6 行相反）。

transform-geninst(*nm*, *instlist*, *adding*)(*dict*) ≜ (3.4.7.16)

```
1  (if instlist = ⟨⟩ then
2     adding
3    else
4     (let mk-Geninst0(id, parm) = hd instlist in
5      let mk-Properties0(lit, op, ax, ma, init) = transform-geninst(nm, tl instlist, adding)(dict) in
6      let tqual = get-visible-qual(id, GENERATOR)(dict) in
7      let mk-Id0(, genname) = id in
8      let mk-GeneratorD(fparm, prop) = dict(tqual) in
9      if len fparm ≠ len parm then
10       exit("§5.4.1.12.2: 在生成程序中实参表和形参表的长度必须相同")
11      else
12       (let (tm, litm, opm, constm) = collect-genparms(fparm, parm) in
13        let tm' = tm + [genname ↦ mk-Id0(⟨⟩, nm)] in
14        let mk-Properties0(lit', op', ax', ma', init') = prop in
15        let lit'' = ⟨insert-genparms(lit'[i])(tm', litm, opm, constm) | 1 ≤ i ≤ len lit'⟩,
16            op'' = ⟨insert-genparms(op'[i])(tm', litm, opm, constm) | 1 ≤ i ≤ len op'⟩,
17            ax'' = ⟨insert-genparms(ax'[i])(tm', litm, opm, constm) | 1 ≤ i ≤ len ax'⟩,
18            ma'' = ⟨insert-genparms(ma'[i])(tm', litm, opm, constm) | 1 ≤ i ≤ len ma'⟩ in
19        if init ≠ nil ∧ init' ≠ nil then
20          exit("§5.5.3.3: 不止一个缺省赋值语句")
21         else
22          (let init'' = if init = nil then init' else init in
23           mk-Properties0(lit ⌢ lit'', op ⌢ op'', ax ⌢ ax'', ma ⌢ ma'', init'')))))
```

type: $Name_0$ $Geninst_0$ $Properties_0$ → *Dict* → $Properties_0$

目的　把一系列生成程序实例变换成 AS_0 性质 $Properties_0$

参数

nm	包含生成程序实例的 AS_0 类别定义的名字
instlist	AS_0 生成程序实例表
adding	ADDING 构件中规定的附加的性质

结果　AS_0 性质 $Properties_0$，将在变换部分类型定义 *transform-partial-typedef* 中变换到 AS_1

算法

第 1 行	当生成程序实例表为空时，则返回 ADDING 的性质（本函数是递归的）。
第 4 行	分解表中的下一个生成程序实例。
第 5 行	变换其余的生成程序实例并返回性质 *properties*，其中已经增加了表其余部分中的生成程序的性质。
第 6—7 行	构造生成程序标识符的限定表 *Qual* 并分解这个生成程序标识符。
第 8 行	分解生成程序描述符。
第 9—10 行	生成程序实在参数表的长度必须与生成程序形式参数表的长度相等。
第 12 行	构造四个映射表，包含形式参数与实在参数之间的关系。*tm* 包含类别形式参数与类别实在参数之间的关系，*litm* 包含字面值形式参数与字面值实在参数之间的关系，*opm* 包含运算符形式参数与运算符实在参数之间的关系，*constm* 包含常数形式参数与常数实在参数之间的关系。

第 13 行　　把生成程序名当作一个形式参数并把它包括在类别映射表中。

第 15—18 行　　由相应的实在参数来替换生成程序体内的每一个形式参数，即替换下列各项中的形式参数：字面值标记（第 15 行）、运算符标记（第 16 行）、等式（第 17 行）和等式的映射部分（第 18 行）。

第 19 行　　如果这个生成程序实例包含一个缺省赋值，则其它的生成程序实例或 ADDING 性质一定不能包含一个缺省赋值。

第 23 行　　把从这个生成程序实例得到的性质联合从其它实例得到的性质组合成所要求的性质。

$$\mathit{insert\text{-}genparms}(\mathit{node})(\mathit{tmap}, \mathit{lmap}, \mathit{omap}, \mathit{constmap}) \triangleq \qquad \textbf{(3.4.7.17)}$$

```
 1  cases node:
 2   (mk-Opspec0(nm, sortl, sort)
 3     → mk-Opspec0(insert-parm(nm, omap), ⟨insert-parm(sortl[i], tmap) | 1 ≤ i ≤ len sortl⟩,
 4                  insert-parm(sort, tmap)),
 5    mk-Equation0(t1, t2)
 6     → mk-Equation0(insert-genparms(t1)(tmap, lmap, omap, constmap),
 7                    insert-genparms(t2)(tmap, lmap, omap, constmap)),
 8    mk-Condequation0(eql, eq)
 9     → mk-Condequation0(⟨insert-genparms(eql[i])(tmap, lmap, omap, constmap) | 1 ≤ i ≤ len eql⟩,
10                        insert-genparms(eq)(tmap, lmap, omap, constmap)),
11    mk-Mappingaxiom0(vl, sid, al)
12     → mk-Mappingaxiom0(vl, insert-parm(sid, tmap),
13                        ⟨insert-genparms(al[i])(tmap, lmap, omap, constmap) | 1 ≤ i ≤ len al⟩),
14    mk-Quantequation0(vl, sid, al)
15     → mk-Quantequation0(vl, insert-parm(sid, tmap),
16                        ⟨insert-genparms(al[i])(tmap, lmap, omap, constmap) | 1 ≤ i ≤ len al⟩),
17    mk-Name0(,)
18     → insert-parm(node, lmap),
19    mk-Id0(,)
20     → insert-parm(node, lmap + constmap),
21    mk-Operatorterm0(x, tli)
22     → mk-Operatorterm0(insert-parm(x, omap),
23                        ⟨insert-genparms(tli[i])(tmap, lmap, omap, constmap) | 1 ≤ i ≤ len tli⟩),
24    mk-Condterm0(e1, e2, e3)
25     → mk-Condterm0(insert-genparms(e1)(tmap, lmap, omap, constmap),
26                    insert-genparms(e2)(tmap, lmap, omap, constmap),
27                    insert-genparms(e3)(tmap, lmap, omap, constmap)),
28    mk-Monadterm0(op, e)
29     → mk-Monadterm0(op, insert-genparms(e)(tmap, lmap, omap, constmap)),
30    mk-Infixterm0(e1, op, e2)
31     → mk-Infixterm0(insert-genparms(e1)(tmap, lmap, omap, constmap),
32                    op, insert-genparms(e2)(tmap, lmap, omap, constmap)),
33    mk-Spellingterm0(x)
34     → mk-Spellingterm0(insert-parm(x, lmap)),
35    T → node)
```

type : $(Literal_0 \mid Op_0 \mid Axiom_0 \mid Mappingaxiom_0) \to (Name_0 \overrightarrow{m} Id_0)\ (Name_0 \overrightarrow{m} Literal_0)$
$(Name_0 \overrightarrow{m} (Name_0 \mid Quotedop_0))\ (Name_0 \overrightarrow{m} Term_0) \to (Literal_0 \mid Op_0 \mid Axiom_0 \mid Mappingaxiom_0)$

目的　　在生成程序的体内，用生成程序实在参数来替换生成程序形式参数

参数

node　　出现在生成程序体内的一个实物节点。函数递归地遍历 *node* 的子树

tmap　　一个映射表，从类别形式参数映射到类别实在参数

lmap	一个映射表，从字面值形实参数映射到字面值实在参数
omap	一个映射表，从运算符形实参数映射到运算符实在参数
constmap	一个映射表，从常数形式参数映射到常数实在参数

结果　　已经替换了形式参数的节点 *node*

算法

第 2—4 行	在一个运算符标记中，考虑运算符名（*nm*）、变元类别表（*sortl*）和结果类别（*sort*）。
第 5—7 行	在一个等式中，考虑它的两个项 *t*1 和 *t*2。
第 8—10 行	在一个条件等式中，考虑限制条件（*t*1）和受限制的等式（*t*2）。
第 11—13 行	在一个映射等式中，考虑类别标识符（*sid*）和所包含的等式（*al*）。
第 14—16 行	在一个量化的等式中，考虑类别标识符（*sid*）和所包含的等式（*al*）
第 17 行	如果此节点是一个字面值名字 $Literal_0$ $Name_0$，则考虑它时要使用字面值映射表。
第 19 行	如果此节点是一个标识符（在一个等式中出现），则考虑它时要使用字面值映射表和常数映射表（在等式中也可以使用字面值形式参数）。
第 21—23 行	在一个运算符应用中，考虑运算符标识符（*x*）和变元项（*tli*）。
第 24—27 行	在一个条件项中，考虑布尔项（*e*1）、“then”项（*e*2）和“else”项（*e*3）。
第 28—32 行	在一元项或二元项中，考虑所包含的项。
第 33 行	在一个拼字项中，考虑所包含的标识符。
第 35 行	其它种类的节点 *node* 不能包含或一定不包含任何形式参数。

insert-parm(x, map) $\triangleq$ **(3.4.7.18)**

```
 1  cases x:
 2   (mk-Id0(q, nm)
 3      → if nm ∈ dom map then
 4           if q = ⟨⟩ then
 5              cases map(nm):
 6               (mk-Name0(,)
 7                  → mk-Id0(⟨⟩, map(nm)),
 8                mk-Quotedop0()
 9                  → mk-Qualop0(⟨⟩, map(nm)),
10                mk-Nmclass0()
11                  → exit("§5.4.1.12: 名字类不能用于等式中")),
12                mk-String0()
13                  → mk-Stringterm0(⟨⟩, map(nm)),
14                T → map(nm))
15             else
16              exit("§5.4.1.12: 限定了生成程序的形式名字")
17           else
18            x,
19    mk-Name0(,)
20      → if x ∈ dom map then
21           cases map(x):
22            (mk-Name0(,),
23             mk-Nmclass0(),
24             mk-String0()
25               → map(x),
26             mk-Quotedop0()
27               → map(x),
28             T → exit("§5.4.1.12: 用于字面值标记的生成程序的常数参数"))
29           else
30            x,
31    T → x)
```

type: $(Id_0 \mid Name_0 \mid Quotedop_0)\ (Name_0 \xrightarrow{m} (Term_0 \mid Literal_0 \mid Quotedop_0)) \rightarrow (Term_0 \mid Literal_0 \mid Quotedop_0)$

目的

测试一个标识符或一个名字是否是一个生成程序的形式参数。如果是的，就用相应的实在参数来替换它。

参数

x　是要测试的标识符或名字。为了简化使用函数 *insert-genparms*，*x* 也可以是一个引用文运算符 $Quotedop_0$，在这种情况下返回未改变的 *x*。

map　一个映射表，包含某参数类的形式参数和实在参数之间的联系。这个参数类依赖于使用函数 *insert-parm* 的上下文。该参数类可以是类型形式参数、运算符形式参数或是字面值和常数形式参数的组合。

结果

如果 *x* 是一个形式参数，结果就是相应的实在参数，否则结果就是 *x*。

算法

第 2 行　如果当前的实物是一个标识符并且其名字部分在映射表（*map*）中，则这个标识符表示一个形式参数，在这种情形下此形式参数一定不能被限定（第 4 行和第 16 行）。

第 6 行　如果实在参数是一个名字（即一个字面值或运算符名），则把它放入一个标识符中返回（因为它用于使用的上下文中）。

第 8 行　如果实在参数是一个引用文运算符，则把它放入一个运算符标识符（$Qualop_0$）中返回。

第 10 行　　在使用的上下文中不能用名字类。

第 12 行　　如果实在参数是一个字符串，则把它放入一个字符串字面值标识符（$Stringterm_0$）中返回。

第 14 行　　否则实在参数是一个项，在这个情形下返回此项。

第 18 行　　如果标识符不是一个形式参数，则返回未改变的标识符。

第 19 行　　如果当前的实物是一个名字并且是在映射表中，则它是一个形式参数，在定义的上下文中用到它。

第 22—24 行　　如果实在参数是一个名字或一个名字类，或者是一个字符串，则返回此实在参数。

第 26 行　　如果实在参数是一个引用文中缀运算符，则返回所含的中缀运算符。

第 28 行　　在其它情况下，实在参数表示一个项，这个项是在定义的上下文中不允许的参数。

第 30 行　　如果名字不是一个形式参数，则返回这个未改变的名字。

collect-genparms(*fparm*, *parm*) ≜ (3.4.7.19)

```
 1  (if fparm = ⟨⟩ then
 2     ([], [], [], [])
 3   else
 4    (let (trest, litrest, oprest, constrest) = collect-genparms(tl fparm, tl parm) in
 5     let exclamation =
 6        if is-Id0(hd parm) then
 7          (let mk-Name0(, exc) = s-Name0(hd parm) in
 8           exc ≠ nil)
 9         else
10          false in
11      cases hd fparm:
12      (mk-Sortparm0(nm)
13         → if is-Id0(hd parm) ∧ ¬exclamation then
14              (trest + [nm ↦ hd parm], litrest, oprest, constrest)
15            else
16             exit("§5.4.1.12.2: TYPE 生成程序的实在参数必须是一个标识符"),
17       mk-Opparm0(nm)
18         → if (is-Id0(hd parm) ∧ s-Qualifier0(hd parm) = ⟨⟩) then
19              (trest, litrest, oprest + [nm ↦ s-Name0(hd parm)], constrest)
20            else
21             if is-Quotedop0(hd parm) then
22               (trest, litrest, oprest + [nm ↦ hd parm], constrest)
23              else
24               exit("§5.4.1.12.2: OPERATOR 生成程序的实在参数必须是一个运算符标记"),
25       mk-Litparm0(nm)
26         → cases hd parm:
27            (mk-String0(str)
28               → (trest, litrest + [nm ↦ str], oprest, constrest),
29             mk-Id0(q, nam)
30               → if q = ⟨⟩ ∧ ¬exclamation then
31                    (trest, litrest + [nm ↦ nam], oprest, constrest)
32                  else
33                   exit("§5.4.1.12.2: LITERAL 生成程序的实在参数必须是一个字面值标记"),
34             mk-Nmclass0()
35               → (trest, litrest + [nm ↦ hd parm], oprest, constrest),
36             T → exit("§5.4.1.12.2: LITERAL 生成程序的实在参数必须是一个字面值标记")),
37       mk-Termparm0(nm)
38         → if is-Nmclass0(hd parm) ∨ is-Quotedop0(hd parm) ∨ exclamation then
39              exit("§5.4.1.12.2: CONSTANT 生成程序的实在参数必须是一个项")
40            else
41             (trest, litrest, oprest, constrest + [nm ↦ hd parm]))))
```

type: $Genparm_0^*$ $Genactparm_0 \to (Name_0 \xrightarrow{m} Id_0)$ $(Name_0 \xrightarrow{m} Literal_0)$
$(Name_0 \xrightarrow{m} (Name_0 \mid Quotedop_0))$ $(Name_0 \xrightarrow{m} Term_0)$

目的

构造四个映射表，对一个生成程序实例，把四种形式参数和相应的实在参数联系起来。在应用这个函数以前，要检查形参表和实参表应有相同的长度。

参数

fparm	生成程序的形式参数表
parm	生成程序的实在参数表

算法

第 1 行	当处理完形式参数表时，返回空的映射表（本函数是递归的）。
第 4 行	构造其余的（除第一个以外全部的）形式参数的映射表。
第 5—10 行	如果实在参数是一个 Id_0，其中 $Name_0$ 带有一个惊叹号，则令 *exclamation* 为真。

第 12—16 行　　如果形式参数是一个类别参数，则实在参数必须是一个不带有惊叹号的标识符。把形式名与标识符之间的联系加到类别映射表中。

第 17—24 行　　如果形式参数是一个运算符参数，则这个实在参数必须是一个非限定的标识符（即一个名字）或一个引用文中缀运算符。

第 19 和 22 行　　依次对每个实在的中缀运算符，把形式参数与实在参数的联系加到运算符映射表中。

第 25 行　　如果形式参数是一个字面值参数，则实在参数必须是一个非限定的标识符或一个非限定的字符串，或者是一个名字类。

第 27 行和 34 行　　把形式名和实在名（或字符串，或名字类）之间的联系加到字面值映射表中。

第 37—41 行　　如果形式参数是一个常数参数，则实在参数必须是一个项（即，不是一个名字类或一个引用文中缀运算符）；如果这个项是一个 Id_0，则它一定不能包含一个惊叹号。

第 41 行　　把形式名和这个项之间的联系加到这个常数映射表中。

transform-syntype(mk-*Syntypedef*$_0$(*nm*, *pid*, *initial*, *vallist*, *tnm*))(*dict*) $\triangleq$ **(3.4.7.20)**

```
 1  (let tq = get-visible-qual(pid, TYPE)(dict) in
 2   if is-recursive-sort(tq, {})(dict) then
 3     exit("§5.4.1.9: Syntype is based on itself")
 4    else
 5     if tnm ∉ {nm, nil} then
 6       exit("§5.4.1.9: 在同义类型定义中的结尾名不同于定义名")
 7      else
 8       (let tqual = get-parent(tq)(dict) in
 9        let initial1 = cases dict(tq):
10                         (mk-SortD(, , init, )
11                            → init,
12                          mk-SyntypeD(, , init, )
13                            → init) in
14        let expr1 = if initial = nil then
15                      initial1
16                    else
17                      (let (as1 tree, , ) =
18                           transform-expr(initial, CONSTANT, {tqual})(dict) in
19                       as1 tree) in
20        let (, as1 valset) = transform-valueset({tqual}, vallist)(dict) in
21        let as1 id = make-as1-identifier(tqual)(dict) in
22        let nm' = if is-ServiceD(dict(dict(SCOPEUNIT))) then
23                    create-unique-name()
24                  else
25                    nm in
26        let as1 dcl = mk-Syn-type-definition1(name-to-name1(nm'), as1 id, as1 valset) in
27        let synqual = dict(SCOPEUNIT) ⁀ ⟨(TYPE, nm')⟩ in
28        let d = [dict(SCOPEUNIT) ⁀ ⟨(TYPE, nm)⟩ ↦ mk-SyntypeD(tq, synqual, expr1, as1 valset)] in
29        ({as1 dcl}, d)))
```

type: *Syntypedef*$_0$ → *Dict* → *Syn-type-definition*$_1$-**set** *Dict*

目的　　把一个同义类型定义变换到 AS_1

参数　　一个同义类型定义包含

nm　　同义类型的名字

pid　　父辈类别标识符

initial　　同义类型变量的初始值

vallist　　此类别变量的允许值集合

tnm　　结束这个定义的名字

结果　　参见变换声明表函数 *transform-decllist*

算法

第 1 行　　抽取指定的父辈类别的限定表 *Qual*。

第 2 行　　父辈类别一定不能基于该同义类型。

第 5 行　　如果规定的话，结尾名必须等于该同义类型名。

第 8 行　　抽取指定的父辈类别的部分类型。

第 9 行　　令 $initial_1$ 表示 *pid* 的初始表达式。

第 14—19 行　　构造该初始表达式的 AS_1 形式。如果它被省略，则采用 *pid* 的初始表达式。

第 20 行　　变换这个值集合

第 22 行　　如果在一个服务中出现这个同义类型定义，则为这个同义类型起一个新名字。

第 27 行　　构造一个限定表 *Qual* 用于导出这个同义类型的 AS_1 标识符（在使用此同义类型的地方）。

第 28—29 行　　返回 AS_1 同义类型定义（as_1dcl）和包含此同义类型描述符（*d*）的对字典 *Dict* 的贡献。

tion containing the syntype descriptor (*d*).

transform-valueset(*tqualset*, *valset*)(*dict*) ≙ **(3.4.7.21)**

```
1  (if valset = ⟨⟩ then
2     (let boolq = get-predef-sort("BOOLEAN")(dict) in
3      let orq = boolq ⌢ ⟨(OPERATOR, (OR, ⟨boolq, boolq⟩, boolq))⟩ in
4      (tqualset, mk-Range-condition1(make-as1-identifier(orq)(dict), {})))
5    else
6     (let mk-Condition0(cr, expr) = hd valset in
7      let (tset, as1range) = transform-valuerange(tqualset, cr, expr)(dict) in
8      let (trest, mk-Range-condition1(orid, condset)) = transform-valueset(tset, tl valset)(dict) in
9      (trest, mk-Range-condition1(orid, {as1range} ∪ condset))))
```

type: *Sortqual*-**set** *Condition*$_0$* → *Dict* → *Sortqual*-**set** *Range-condition*$_1$

目的　　把一个值集合变换到 AS_1

参数

tqualset　　由多个可能的（合法的）类别组成的集合

valset　　是值的范围的表。在同义类型定义中出现此值集合时，*tqualset* 仅包含此同义类型的父辈。在一个判定回答中出现这个值集合时，此 *tqualset* 包含询问表达式的可能类别的某个子集合（也与其它回答有关）。

结果　　在处理了此值集合之后得到可能的（合法）类别的集合；即结果集合是一个 *tqualset* 的子集合。还返回 AS_1 的值集合（*Range-Expression*$_1$*）。

算法

第 1—4 行　　当处理完值集合中的所有元素时，则返回可能类别的集合和一个包含布尔 **OR** 运算符的 AS_1 标识符的空范围条件 *Range-condition*$_1$。

第 6 行　　取出这个集合（表）中的第一个元素，它包括一个任选运算符或一个任选的表达式（*cr*）后随一个表达式（*expr*）。

第 7—9 行　　变换一个值范围，并且在值集合的其余部分变换期间使用所得到的（可能受限制的）类别集合（*tset*）。

transform-valuerange(*tqualset*, *cr*, *expr*)(*dict*) ≜　　(3.4.7.22)

```
 1  (let bq = get-predef-sort("BOOLEAN")(dict) in
 2   let cr' = if cr = nil then EQ else cr in
 3   let isrel = cr' ∈ {NE, EQ, GT, LT, LE, GE} in
 4   let cr'' = if isrel then cr' else LE in
 5   let tqualset' =
 6       if isrel then
 7         tqualset
 8        else
 9         (let (,tqset,) = transform-expr(cr, CONSTANT, {bq})(dict) in
10          tqset ∩ tqualset) in
11   let opqset =
12       {opqual ∈ all-visible-operators(⟨⟩, cr'', dict(SCOPEUNIT))(dict) |
13        is-OperatorD(dict(opqual)) ∧
14        (let (, (,argl, res)) = opqual[len opqual] in
15         len argl = 2 ∧
16         res = bq ∧ argl[1] = argl[2] ∧ elems argl ⊂ tqualset')} in
17   if opqset = {} then
18     exit("§5.4.1.9.1: 对于范围条件的类别没有定义有序运算符")
19    else
20     (let tset = tqualset' ∩ {sq ∈ dom dict | is-SortD(dict(sq)) ∧
21                                              (∃q ∈ opqset)((OPERATOR, (cr'', ⟨sq, sq⟩, bq)) = q[len q])} in
22      if card tset ≥ 1 then
23        (tset, nil)
24       else
25        (let (as1tree, , ) = transform-expr(expr, CONSTANT, tset)(dict) in
26         let sort ∈ tset in
27         let opqual = sort ⌢ ⟨(OPERATOR, (cr'', ⟨sort, sort⟩, bq))⟩ in
28         if isrel then
29           (tset, mk-Open-range1(make-as1-identifier(opqual)(dict), as1tree))
30          else
31           (let (as1tree', , ) = transform-expr(cr, CONSTANT, tset)(dict) in
32            let r1 = mk-Open-range1(make-as1-identifier(opqual)(dict), as1tree'),
33                r2 = mk-Open-range1(make-as1-identifier(opqual)(dict), as1tree) in
34            let andq = bq ⌢ ⟨(OPERATOR, (AND, ⟨sort, sort⟩, {bq}))⟩ in
35            (tset, mk-Closed-range1(make-as1-identifier(andq)(dict), r1, r2))))))
```

type:　*Sortqual*-**set** [*Expr*$_0$ | *Relop*$_0$] *Expr*$_0$ → *Dict* → *Sortqual*-**set** [*Condition-item*$_1$]

目的　　把一个 AS_0 条件 *Condition*$_0$ 变换为一个 AS_1 条件 *Condition-item*$_1$

参数

tqualset	在值范围变换之前的合法类别的集合
cr	关系运算符或出现在值范围中的第一个表达式
expr	值范围的（第二个）表达式

结果

在值范围的变换之后的合法类别的集合以及构造的 AS_1 条件 $Condition_1$。这个类别集合被用于变换判定动作，在该变换中本函数被使用了两次。第一次是为了导出类别，第二次是为了变换值范围。

算法

第 1 行	构造布尔类别的限定表 *Qual*。
第 2 行	如果值范围内既没有指定一个关系运算符又没有指定一个表达式，则应用相等运算符。
第 4 行	如果 *cr* 是一个表达式，则小于—等于运算符（**LE**）被当作关系运算符使用。
第 5—10 行	如果 *cr* 是一个表达式，则合法类别的集合（*tqualset*）被那个表达式的合法类别所限制。
第 11—16 行	构造合法的关系运算符的限定表 *Qual* 集合，使得变元表的长度等于 2，并且结果类别是布尔型的。这两个变元的类别相同，是使用值范围的那个上下文中合法类别之一（如果这个集合的元素不止一个，则这个运算符名（*cr*）超载）。
第 17—18 行	如果不存在这样的运算符，则出现错误。
第 20—21 行	构造一个新的合法类别的集合，办法是限制合法类别（*tqualset'*）的原集合，使它仅仅包含那些可以用在 *opqset* 中的运算符上的类别。
第 22—23 行	如果在限制的类别集合中存在的元素不止一个，则值范围（仍然）是二义性的并且不返回值范围（返回 **nil**）。
第 25 行	再次求这个表达式的值。这次，*tset* 只包含一个类别，因此返回一个 AS_1 表达式（另外它应是良形式的）。
第 26 行	用 *sort* 来表示 *tset* 中的类别。
第 27 行	构造所用的运算符的限定表 *Qual*。
第 28—29 行	如果 *cr* 表示一个关系运算符，则返回一个集合，它包含此类别和 $Condition\text{-}item_1$（这是一个 $Open\text{-}range_1$）
第 31 行	如果 *cr* 是一个表达式，则再次变换这个表达式。
第 32—33 行	构造两个开范围 $Open\text{-}range_1$，它们与 AND 运算符一起组成了闭范围 $Closed\text{-}range_1$。
第 34 行	构造 AND 运算符的限定表 *Qual*，并且
第 35 行	返回类别的结果集合（包含一个元素）和此闭范围 $Closed\text{-}range_1$，它包含 AND 运算符的 AS_1 标识符和两个开范围 $Open\text{-}range_1$。

transform-synonymdef(**mk-***Synonymdef*$_0$(*nm*, *tid*, *expr*))(*dict*) $\triangleq$ **(3.4.7.23)**

```
 1  (let squal = dict(SCOPEUNIT) ⌢ ⟨(VALUE, nm)⟩ in
 2   let sortset = if tid = nil then
 3                    all-visible-sorts(dict)
 4                  else
 5                    {get-parent(get-visible-qual(tid, TYPE)(dict))(dict)} in
 6   let (,tpset,) = transform-expr(expr, CONSTANT, sortset)(dict + [squal ↦ mk-ErrorD()]) in
 7   if card tpset = 1 then
 8     (let tp ∈ Qual be s.t. tp ∈ tpset in
 9      ([squal ↦ mk-SynD(tp, expr)]))
10    else
11     exit "§5.4.1.13: 同义词类别不能被唯一地确定"))
```

type: *Synonymdef*$_0$ → *Dict* → *Dict*

目的　　把一个同义词定义变换到 AS_1

参数　　一个 AS_0 同义词定义包含

nm　　同义词名

tid　　任选的类别标识符

expr　　基表达式

结果　　参见变换声明表函数 *transform-decllist*

算法

第 1 行　　构造表示此同义词的限定表 *Qual*。

第 2—5 行　　必须在类别的集合中找到同义词的类别；如果类别标识符 (*tid*) 不存在的话，类别的集合对于同义词定义来说是可见的，否则这个同义词的类别就是 *tid*。

第 6 行　　变换包含在同义词定义中的表达式，并返回匹配两个同义词类别的子集合 (*tpset*)，不使用所得到的 AS_1 表达式。由于同义词标识符一定不能用于 *expr* 中，所以在表达式的求值期间 *squal* 表示一个错误 *ErrorD*。

第 7 行　　只允许一个类别去匹配此同义词的类别和这个表达式的类别。

第 8—9 行　　构造包含此同义词描述符的字典 *Dict* 的贡献。

3.4.7.1 部分类型定义

$$transform\text{-}sortdef(nm, prop, parent)(dict) \triangleq \tag{3.4.7.1.1}$$

```
 1  (let nm' = if is-ServiceD(dict(dict(SCOPEUNIT))) then create-unique-name() else nm in
 2   let newqual = dict(SCOPEUNIT) ⌢ ⟨(TYPE, nm')⟩ in
 3   let descr = dict(SCOPEUNIT) ⌢ ⟨(TYPE, nm)⟩ in
 4   let dict' = dict + [SCOPEUNIT ↦ descr] in
 5   let prop' = transform-nameclass(prop) in
 6   let prop'' = add-ordering(prop', descr)(dict) in
 7   let mk-Properties0(lit, oplist, axioms, mapping, term) = prop'' in
 8   let (eqop, eqax) = add-equality(descr)(dict) in
 9   let (expr1, , ) = transform-expr(term, CONSTANT, {descr})(dict) in
10   if card elems lit ≠ len lit then
11     exit("§5.2.2: 在一个部分类型定义中字面值定义两次")
12    else
13     (let litd = [descr ⌢ ⟨(LITERAL, nm)⟩ ↦ mk-LiteralD(descr) | nm ∈ elems lit],
14          (as1 op, opd) = transform-typing(oplist ⌢ eqop)(dict'),
15          as1 ax = transform-axioms(axioms ⌢ eqax, AXIOMS)(dict'),
16          as1 mapping = transform-axioms(mapping, MAPPING)(dict') in
17      let sortid = make-as1-identifier(newqual)(dict) in
18      let as1 litset = {mk-Literal-signature1(nm, sortid) | nm ∈ card lit} in
19      let concaxioms1 = make-as1-concaxioms(dom litd)(dict) in
20      let typedef = dict(DATATYPEDEF) in
21      let mk-Data-type-definition1(typename, union, sorts, sigs, eqs ∪ concaxioms1) = typedef in
22      let datatypedef' = mk-Data-type-definition1(typename, union, sorts ∪ {name-to-name1(nm')},
23                                                  sigs ∪ as1 op ∪ as1 litset,
24                                                  eqs ∪ as1 ax ∪ as1 mapping ∪ concaxioms1) in
25      let sortdescr = mk-SortD(as1 ax ∪ as1 mapping, parent, expr1, newqual) in
26      (litd + opd + [descr ↦ sortdescr,
27                     DATATYPEDEF ↦ datatypedef'])))
```

type : $Name_0\ Properties_0\ [Sortqual] \to Dict \to Dict$

目的

变换一个类别定义的性质 $Properties_0$，并给字典 *Dict* 中的数据类型定义 *DATATYPEDEF* 项目中所包含的数据类型定义 $Data\text{-}type\text{-}definition_1$ 增加所得到的 AS_1 性质，也把字面值描述符、运算符描述符和类别描述符增加到字典 *Dict* 中。

参数

nm 被定义的类别的 AS_0 名字

prop 要变换的 AS_0 性质

parent 在性质来源于继承类别情况下的父辈类别的限定表 *Qual*，即 *parent* 表示这个类别所继承的类别的标识符。它被放入类别描述符 *SortD* 的描述符中（见 *SortD* 的定义）。

结果

用 AS_1 性质和各种描述符更新过的字典 *Dict*

算法

第 1—2 行 构造一个限定表 *Qual*，用于构造 AS_1 名字。如果在一个服务中定义了此类别，在 *Qual* 中就要使用一个新的唯一的名字。

第 3—4 行 更新作用域单位 **SCOPEUNIT** 以表示这些性质是在类别 *nm* 所在的范围求值的。并且更新字典 *Dict* 以包括这个新的作用域单位的信息。

第 5 行 修改这些性质使得在字面值定义中所有的名字类被扩展成为一个字面值的序列。

第 6 行 修改这些性质使得运算符标记中的任何次序 **ORDERING** 都由这个运算符标记和反映顺序性质的等式替换。

第 7 行 分解这个性质 $Properties_0$。

第 8 行 构造 AS_0 运算符和反映相等性质的等式。

第 9 行 把类别定义中的缺省表达式 DEFAULT 变换到 AS_1

第 10 行 字面值标记中不同的字面值名的数目必须与字面值标记表的长度相等。

第 13 行 构造所有在字面值标记中的字面值的描述符。

第 14 行 构造 AS_1 运算符标记 $Operator\text{-}signature_1$（$as_1op$）和运算符标记中的运算符的描述符以及相等运算符的描述符。

第 15 行 从 AS_0 等式（公理）构造 AS_1 等式，这些 AS_0 等式是在性质中指定的等式联合这些相等等式。

第 17 行 构造对应于 AS_0 性质的映射部分 MAP 的等式。

第 18 行 构造 AS_1 字面值标记。

第 19 行 构造从任何定义的字符串字面值隐含的 AS_1 等式（隐含的拼接等式）。

第 21 行 抽取并分解当前的数据类型定义 $Data\text{-}type\text{-}definition_1$。

第 22—23 行 构造一个新的数据类型定义 *Data-type-definition*，它是对旧的定义的更新，使之包括构造的 AS_1 性质。

第 25 行 构造此类别的描述符。

第 26 行 返回字典 *Dict*，包括字面值的描述符、运算符的描述符、类别的描述符和修改过的数据类型定义 $Data\text{-}type\text{-}definition_1$。

目的

修改某些 AS_0 性质使得任何所含的名字类被扩展成为字面值名字的一个序列

参数

AS_0 性质包含

litlist 字面值标记表

transform-nameclass(**mk-***Properties*$_0$(*litlist*, *o*, *a*, *m*, *t*)) $\triangleq$ (3.4.7.1.2)

```
 1  if (∃lit ∈ elems litlist)(is-Nmclass0(lit)) then
 2    (let i ∈ ind litlist be s.t. is-Nmclass0(litlist[i]) in
 3     let mk-Nmclass0(regexpr) = litlist[i] in
 4     let stringset = {str ∈ Char+ | is-in-regular-expr(str, regexpr)} in
 5     let nameset = form-names-and-strings(stringset) in
 6     let namelist be s.t. card elems namelist = card nameset ∧
 7         len namelist = card nameset ∧
 8         (∀i1, i2 ∈ ind namelist)
 9          (namelist[i1] < namelist[i2] ⊃ i1 < i2) in
10     let litlist' = ⟨litlist[n] | 1 ≤ n < i⟩ ⌢ namelist ⌢ ⟨litlist[n] | i < n ≤ len litlist⟩ in
11     transform-nameclass(mk-Properties0(litlist', o, a, m, t)))
12   else
13    mk-Properties0(litlist, o, a, m, t)
```

type : *Properties*$_0$ → *Properties*$_0$

o	运算符标记表
a	多个等式
m	多个映射等式
t	表示缺省值 DEFAULT 的项

结果 修改过的 AS$_0$ 性质

算法

第 1 行	如果字面值标记中（仍然）存在一个名字类，则
第 2—3 行	令 *i* 表示字面值标记表中元素（是一个名字类）的标引号，并分解这个名字类（第 3 行）。
第 4 行	构造（Meta-IV）字符串的集合，这些字符串满足由 *regexpr* 表示的正规表达式。
第 5 行	从这个字符串的集合构造一个名字集合 *nameset*，包括 AS$_0$ 名字和字符串。
第 6—9 行	把这个集合变换为一个按照字母顺序排列的表。
第 10 行	由构造好的表来替换字面值标记中的这个名字类，并且
第 11 行	替换字面值标记表中尚有的其它名字类。
第 13 行	如果没剩下名字类，则返回这些性质。

form-names-and-strings(*stringset*) $\triangleq$ (3.4.7.1.3)

```
 1  if stringset = {} then
 2    {}
 3   else
 4    (let string ∈ stringset in
 5     if len string ≥ 2 ∧ string[1] = "'" ∧ string[len string] = "'" then
 6       {mk-String0(⟨string[i] | 2 ≤ i ≤ len string − 1⟩)} ∪
 7       form-names-and-strings(stringset \ {string})
 8      else
 9       (let string' = check-name-syntax(string) in
10        {mk-Name0(string', nil)} ∪
11        form-names-and-strings(stringset \ {string})))
```

type : *Char**-**set** → (*String*$_0$ | *Name*$_0$)-**set**

目的　　从一个 Meta-IV 字符串的集合形成一个由 AS_0 名字和字符串组成的集合

算法

第 1 行　　当全部处理完时，什么也不返回（本函数是递归的）。
第 4 行　　取出这个集合的一个元素并用 *string* 表示它。
第 5 行　　如果第一个字符和最后一个字符是单引号，则 *string* 就表示一个 AS_0 字符串。
第 6—7 行　　与其余的名字和串一起返回这个 AS_0 字符串。
第 9—11 行　　如果它不是一个字符串，它就表示一个 AS_0 名字。
第 9 行　　根据下划线字符的语义删除或替换空格字符，把名字转换成大写字体，并检查这个字的拼法是否与 Z.100 中定义的词法规则抵触。
第 10—11 行　　与其余的名字和串一起返回这个 AS_0 名。

is-in-regular-expr(*string*, *regexpr*) $\triangleq$ (3.4.7.1.4)

```
 1  ( cases regexpr:
 2    (mk-Orregexp0(reg, partreg)
 3       → is-in-regular-expr(string, reg) ∧
 4         is-in-regular-expr(string, partreg),
 5     mk-Andregexp0(reg, partreg)
 6       → (∃(str1, str2))(str1 ⌢ str2 = string ∧
 7                         is-in-regular-expr(str1, reg) ∧
 8                         is-in-regular-expr(str2, partreg)),
 9     mk-Rngregexp0(str1, str2, exptp)
10       → is-in-regular-range(string, str1, str2, exptp),
11     mk-Singregexp0(str1, exptp)
12       → is-in-regular-single(string, str1, exptp),
13     mk-Parenregexp0(regexp, exptp)
14       → is-in-regular-par(string, regexp, exptp)))
```

type: $Char^*$ $Regularexp_0 \rightarrow Bool$

目的　　检测在一个正规表达式中是否含有一个 Meta-IV 字符串

参数

string　　Meta-IV 字符串
regexpr　　正规表达式

结果　　如果成功的话，则结果为真。

第 2—4 行　　如果此正规表达式由两个正规表达式用 OR 运算符连接组成，则只要这个串在其中的一个表达式中定义的话，结果就为真。
第 5—8 行　　如果正规表达式由两个拼字的正规表达式组成，则如果存在两个串使得它们在拼接时形成 *string*，并且使得第一子串在第一正规表达式中被定义，第二子串在第二正规表达式中被定义，则结果为真。
第 9 行　　参见函数 *is-in-regular-range*
第 11 行　　参见函数 *is-in-regular-single*
第 13 行　　参见函数 *is-in-regular-par*

is-in-regular-range(*string*, mk-*String*$_0$(*str*1), mk-*String*$_0$(*str*2), *exptp*) ≜ (3.4.7.1.5)

```
 1  (if len str1 ≠ 1 ∨ len str2 ≠ 1 then
 2     exit("§5.4.1.14: 在正规间隔中的字符串的长度不等于1")
 3    else
 4     (if len string = 0 ∧ exptp = MULT then
 5        true
 6       else
 7        (len string ≤ 1 ∧
 8         (∀ch ∈ elems string)
 9          ((hd str1 ≤ ch ∧
10            ch ≤ hd str2 ∧
11            ( cases exptp:
12              (mk-Name0(numstr, )
13                 → if elems numstr ⊂ {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"} then
14                      (let i = form-integer(numstr) in
15                       len string = i)
16                     else
17                      exit("§5.4.1.14: 在名字类中的重复名字必须表示一个自然数"),
18               T → true)))))))
```

type: *Char** *String*$_0$ *String*$_0$ [*Regexpexp*$_0$] → *Bool*

目的

检测一个串是否定义在一个正规表达式中，这个正规表达式就是一个字符的范围(character range)，这个串可能重复若干次。

参数

string	要被检测的串
str1，*str2*	表示下界和上界的字符序列
exptp	任选的重复因子

结果

如果成功的话，则结果为真。

算法

第 1—2 行	上界与下界的长度必须等于 1。
第 4 行	只有在指定了重复因子，且允许空串存在的情况下空串才能在指定的范围中。
第 7—10 行	否则串的长度必须大于或等于 1，并且对该串中的所有字符（*ch*）来说，必须满足字符的代表值大于等于下界（第 8 行），并且小于等于上界。
第 11—17 行	如果重复因子是一个名字，则必须满足它表示一个自然数的字面值名（它必须由数字组成）并且它表示的值等于这个串的长度。

form-integer(*str*) ≜ (3.4.7.1.6)

```
 1  (let ch = str[len str] in
 2   let n = cases ch:
 3           ("0" → 0,
 4            "1" → 1,
 5            "2" → 2,
 6            "3" → 3,
 7            "4" → 4,
 8            "5" → 5,
 9            "6" → 6,
10            "7" → 7,
11            "8" → 8,
```

```
12              "9" → 9) in
13    if tl str = ⟨⟩ then n else form-integer(⟨str[i] | 1 ≤ i ≤ len str⟩) * 10 + n)
```

type: $Char^{+} \rightarrow Intg$

目的　　把字符的序列（也叫做一个拼字）转换成一个 Meta-IV 整数 *Intg* 值

算法

第 1 行　　抽取字符表中最后一个字符。

第 2—12 行　　把这个字符转换成一个 Meta-IV 整数 *Intg* 值。

第 13 行　　如果它是字符表中仅有的一个字符，则返回其值。否则从字符表的其余部分获得的值乘以 10，并且把从这个字符获得的值加到此结果上。

is-in-regular-single(*string*, **mk-***String*$_0$(*str*), *exptp*) $\triangleq$ **(3.4.7.1.7)**

```
 1  ( cases exptp:
 2    (MULT
 3       → (∃n)(conc ⟨str | 0 ≤ i ≤ n⟩ = string),
 4     PLUS
 5       → (∃n)(conc ⟨str | 1 ≤ i ≤ n⟩ = string),
 6     mk-Name0(numstr, )
 7       → if elems numstr ⊂ {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"} then
 8            (let n = form-integer(numstr) in
 9             conc ⟨str | 1 ≤ i ≤ len n⟩ = string)
10           else
11            exit("§5.4.1.14: 在名字类中的重复名字必须表示一个自然数"),
12     nil
13       → str = string))
```

type: $Char^{*}$ $String_0$ $[Regexpexp_0] \rightarrow Bool$

目的　　检测某个串是否能通过多次重复另一个串来构成

参数

string　　要构成的串

str　　要重复的串

exptp　　重复因子

结果　　如果成功的话，则结果为真。

算法

第 2 行　　如果重复因子是个星号，则必须存在一个值 *n* 使得 *str* 的 *n* 次重复（拼接）可以构成 *string*。

第 4 行　　如果重复因子是一个加号，则必须存在一个非零值 *n*，使得 *str* 的 *n* 次重复（拼接）构成 *string*。

第 6 行　　如果重复因子是一个名字，它必须表示一个自然数。如果把 *str* 重复那个次数能构成 *string* 的话，结果就为真。

第 12 行　　如果省略了重复因子，这两个串 *str* 与 *string* 必须相等。

is-in-regular-par(*string*, *regexp*, *exptp*) ≜ **(3.4.7.1.8)**

```
 1  cases exptp:
 2   (MULT
 3      → (∃(str, n))(conc ⟨str | 0 ≤ i ≤ n⟩ = string ∧
 4                  is-in-regular-expr(str, regexp)),
 5    PLUS
 6      → (∃(str, n))(conc ⟨str | 1 ≤ i ≤ n⟩ = string ∧
 7                  is-in-regular-expr(str, regexp)),
 8    mk-Name0(numstr, )
 9      → if elems numstr ⊂ {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"} then
10          (let n = form-integer(numstr) in
11           (∃str)(conc ⟨str | 1 ≤ i ≤ n⟩ = string ∧
12                  is-in-regular-expr(str, regexp)))
13        else
14         exit("§5.4.1.14: 在名字类中的重复名字必须表示一个自然数"),
15    nil
16      → is-in-regular-expr(string, regexp))
```

type: $Char^*$ $Regularexp_0$ [$Regexpexp_0$] → *Bool*

目的　　检测某个串是否能通过若干次重复一个由正规表达式表示的子串来构成

参数

string	要检测的串
regexp	要被重复的正规表达式
exptp	任选的重复因子

结果　　如果成功的话，则结果为真。

算法

第 2—4 行　　如果重复因子是一个星号，则必须存在一个由正规表达式 *regexp* 定义的子串 *str* 和一个数字 *n*，使得这个子串的 *n* 次重复构成这个串。

第 5—7 行　　如果重复因子是一个加号，则必须存在一个由正规表达式 *regexp* 定义的子串 *str* 和一个非零数字 *n*，使得子串的 *n* 次重复构成这个串。

第 8—14 行　　如果重复因子是一个名字，它必须表示一个自然数。如果存在一个由正规表达式 *regexp* 定义的子串使得按这个次数重复的子串形成 *string*，则结果为真。

第 15 行　　如果省略了重复因子，如果此正规表达式定义了这个串，则结果为真。

transform-axioms(*axioms*, *context*)(*dict*) ≜ **(3.4.7.1.9)**

```
1  (if axioms = ⟨⟩ then {} else (transform-axiom(hd axioms, context)(dict) ∪
2                                transform-axioms(tl axioms, context)(dict)))
```

type: $Axiom_0^*$ *Context* → *Dict* → $Equations_1$

目的　　把一个 AS_0 等式（公理）表变换为一组 AS_1 等式

参数

axioms	要被变换的公理

context 这个上下文是变换公理的上下文。这个上下文或者表示正常公理（**AXIOMS**），或者表示映射公理（**MAPPING**）。

结果 生成的 AS_1 等式

算法

第 1 行 当全部处理完时，返回空集。

第 1—2 行 返回等式的集合，该集合由对应于第一个等式的集合（第 1 行）与对应于其余等式的集合（第 2 行）合并组成。

transform-axiom(*axiom*, *context*)(*dict*) ≙ (3.4.7.1.10)

```
 1  ( cases axiom:
 2    (mk-Equation0(,)
 3       → {transform-equation(axiom, context)(dict)},
 4     mk-Condequation0(eql, eq)
 5       → (let eql' = ⟨insert-equals-true(eql[i])(dict) | i ≤ i ≤ len eql⟩,
 6              eq' = insert-equals-true(eq)(dict) in
 7          {transform-condequation(mk-Condequation0(eql', eq'), context)(dict)}),
 8     mk-Quantequation0(,,)
 9       → {transform-quantequation(axiom, context)(dict)},
10     mk-Mappingaxiom0(,,)
11       → transform-mapping(axiom)(dict),
12     T → if is-Stringterm0(axiom) ∧ s-Qualifier0(axiom) = ⟨⟩ then
13           {mk-Informal-text1(axiom)}
14         else
15          (let axiom' = insert-equals-true(axiom)(dict) in
16           {transform-equation(axiom', context)(dict)}))
```

type: $Axiom_0$ *Context* → *Dict* → $Equations_1$

目的 把一个 AS_0 公理变换为一个 AS_1 等式的集合。除非这个公理是一个映射公理，该集合仅由一个元素组成。

参数

axiom 要被变换的公理

context 这个上下文是变换发生的上下文（见变换公理函数 *transform-axioms*）。

结果 生成的 AS_1 等式

算法

第 2 行 如果公理是一个简单的等式，则变换该等式。

第 4—7 行 如果公理是一个条件等式，则用一个简单等式替换限制条件（*eql*）中的布尔项和被限制的等式（*eq*）中的布尔项，并且变换结果条件等式。

第 8—10 行 如果公理是一个量化的等式（第 8 行）或一个映射等式（第 10 行），则变换它。

第 12—13 行 否则如果这个项是一个非限定的字符串，则公理是一个表示非正式正文的项。

第 15—16 行 如果公理不表示非形式的正文，则它是一个布尔公理，并且在变换它之前要插入“等于真”。

insert-equals-true(*axiom*)(*dict*) ≙ (3.4.7.1.11)

```
1  if is-Equation0(axiom) then
2    axiom
3   else
4    (let truenm = mk-Name0("TRUE", nil) in
5     let trueid = mk-Id0(⟨hd dict(SCOPEUNIT)⟩, truenm) in
6     mk-Equation0(axiom, trueid))
```

type: $Unquantequation_0 \rightarrow Dict \rightarrow Equation_0$

目的 如果公理是一个布尔项，则构造一个 AS_0 等式，即定义这个项等于真。

参数

axiom 要被检测的公理

结果 可能修改过的公理

算法

第 1 行 如果公理是一个等式，则它是合格的，否则

第 4 行 它是一个项，则构造其布尔类别的名字。

第 5 行 构造布尔类别的标识符。其限定词是 *Qual* 作用域单位 **SCOPEUNIT** 中的第一个元素，**SCOPEUNIT** 表示当前作用域单位。第一个元素表示系统层。

第 6 行 从此布尔项和真值字面值标识符构造一个等式。

transform-equation(mk-*Equation*$_0$(*term*1, *term*2), *context*)(*dict*) ≙ (3.4.7.1.12)

```
1  (let (as1term1, as1term2, dict') = transform-lhs-rhs(term1, term2, context)(dict) in
2   build-quant-equation(mk-Unquantified-equation1(as1term1, as1term2))(dict'))
```

type: $Equation_0\ Context \rightarrow Dict \rightarrow Equations_1$

目的 把一个简单的非量化的 AS_0 等式变换为一个 AS_1 等式

参数

*term*1，*term*2 要被变换的等式中的两个项

context 参见变换公理函数 *transform-axioms*

结果 构造好的 AS_1 等式

算法

第 1 行 变换这两个项。在 *dict'*（见字典 *Dict* 的定义）中的新的 *ValueidD* 描述符包含了这两个项内用到的所有的值标识符的信息。

第 2 行 从这两个 AS_1 项构造一个非量化的等式，并通过某种量化来封闭这个等式（如果 *dict'* 中有任何 *ValueidD* 描述符的话）。

transform-lhs-rhs(*term1*, *term2*, *context*)(*dict*) ≙ (3.4.7.1.13)

```
 1  (let totset = all-visible-sorts(dict) in
 2   let (,tset1,) =
 3       if is-Errorterm0(term1)
 4         then (...,totset,...)
 5         else transform-term(term1, context, totset)(dict),
 6       (,tset2,) =
 7       if is-Errorterm0(term2)
 8         then (...,totset,...)
 9         else transform-term(term2, context, totset)(dict) in
10   let tqual = if card (tset1 ∩ tset2) = 1 then
11                  (let tq ∈ (tset1 ∩ tset2) in
12                   tq)
13                 else
14                  exit("§4.2.3: 等式中的两项必须是相同类别的")                    in
15   let (as1term1,, d) =
16       if is-Errorterm0(term1)
17         then (mk-Error-term1(), ..., dict)
18         else transform-term(term1, context, {tqual})(dict) in
19   let (as1term2,, d') =
20       if is-Errorterm0(term2)
21         then (mk-Error-term1(), ..., d)
22         else transform-term(term2, context, {tqual})(d) in
23   (as1term1, as1term2, d'))
```

type: $Term_0\ Term_0\ Context \rightarrow Dict \rightarrow Term_1\ Term_1\ Dict$

目的

把一个等式中的两个 AS_0 项变换为两个 AS_1 项

参数

term1，*term2*	要被变换的两个项
context	参见变换公理函数 *transform-axioms*

结果

两个构造好的 AS_1 项和一个字典 *Dict*（它包括在这两个项中出现的值标识符的描述符 *ValueidD*）

算法

第 1 行	抽取包含所有可见类别的限定表 *Qual* 的集合。
第 2—9 行	为了确定这两个项的合法类别的集合，变换这两个项。*tset1* 表示 *term1* 的合法类别集合，*tset2* 表示 *term2* 的合法类别集合。如果两个项中有一个是错误项，则所有可见类别对那个项来讲都是合法的。
第 10—14 行	确定等式的类别。必须恰好有一个这样的类别，即这两个集合必须有一个公共类别。
第 15—23 行	如果两个项中有一个是错误项，则返回那个项的 AS_1 错误项，否则按所给的合法类别（*tqual*）作为变元来变换它（第 18 行和第 22 行）。在第二个项被变换时（第 22 行）要使用从第一个项的变换函数 *transform-term* 返回的字典 *Dict*（第 18 行），这样使得 *ValueidD* 描述符不被产生两次。

build-quant-equation(*equation*$_1$)(*dict*) ≜ **(3.4.7.1.14)**

```
1  if (∃q ∈ dom dict)(is-ValueidD(dict(q)) ∧ ¬s-Explicit(dict(q))) then
2    (let q ∈ dom dict be s.t. is-ValueidD(dict(q)) ∧ s-Mapvalue(dict(q)) = nil ∧ ¬s-Explicit(dict(q))) in
3     let mk-ValueidD(,tqs,) = dict(q) in
4     let tq ∈ tqs in
5     let (,nm) = q[len q] in
6     let equation' = build-quant-equation(equation1)(dict \ {q}) in
7     let as1 tid = make-as1-identifier(tq)(dict) in
8     mk-Quantified-equations1({name-to-name1(nm)}, as1 tid, {equation'}))
9    else
10    equation1
```

type: *Equation*$_1$ → *Dict* → *Equation*$_1$

目的 通过量化出现在 *dict* 中的每个值名字封闭一个 AS$_1$ 等式

参数

equation$_1$ 是要被量化的 AS$_1$ 等式。因为此函数是递归的，*equation*$_1$ 可以是一个已经被量化的等式。

算法

第 1 行 如果在字典 *Dict* 中有一个值标识符描述符 *ValueidD*，并且它不是由显式的量化引入的，则

第 2 行 用 *q* 表示这个值标识符的限定表 *Qual*。

第 3 行 分解这个值标识符描述符。类别 *tqs* 的集合在这个阶段仅包含一个类别。

第 4 行 用 *tq* 表示值标识符的类别。

第 5 行 抽取值标识符的名字。

第 6 行 由相应于其余的值标识符的量化来封闭那个旧等式，通过封闭那个旧等式来构造一个新等式。

第 7 行 从值标识符限定表 *Qual* 构造 AS$_1$ 标识符。

第 8 行 返回用一个 AS$_1$ 名字量化了的等式，它包含这个新的等式。

transform-quantequation(mk-*Quantequation*$_0$(*vl*, *tid*, *axl*), *context*)(*dict*) ≜ **(3.4.7.1.15)**

```
1  (let tqual = get-visible-qual(tid, TYPE)(dict) in
2   let qualset = {dict(SCOPEUNIT) ⌢ ⟨(VALUE, vl[i])⟩ | 1 ≤ i ≤ len vl} in
3   let dict' = dict + [q ↦ mk-ValueidD(nil, {tqual}, true) |
4                       q ∈ qualset] in
5   let as1 ax = transform-axioms(axl, context)(dict') in
6   let as1 tid = make-as1-identifier(tqual)(dict) in
7   let as1 vl = {name-to-name1(vl[i]) | 1 ≤ i ≤ len vl} in
8   mk-Quantified-equations1(as1 vl, as1 tid, as1 ax))
```

type: *Quantequation*$_0$ *Context* → *Dict* → *Quantified-equations*$_1$

目的 把一个量化的等式变换到 AS$_1$

参数

vl, *tid*, *axl* 在量化的等式中所含的值名表、它们的类别和公理表

context 参见变换公理函数 *transform-axioms*

结果 AS_1 量化的等式

算法

第 1 行　构造类别（*tid*）的限定表 *Qual*。

第 2 行　构造相应于值名（*vl*）的值标识符的限定表 *Qual*。

第 3—4 行　更新字典 *Dict* 以包括值标识符描述符。**nil** 表示它不是一个字面值的值标识符（映射），{*tqual*} 表示这些值标识符所允许的类别是 *tqual*（在隐式量化中）；该集合可以包含几个元素，*true* 表示值标识符是由显式的量化引入的。

第 5 行　把量化的等式中的公理变换到 AS_1。

第 6 行　为此类别的限定表 *Qual* 构造一个 AS_1 标识符。

第 7 行　构造一个 AS_1 名字集合包含这些值名。

第 8 行　返回一个构造好的 AS_1 量化的等式。

transform-condequation(**mk-***Condequation*$_0$(*eql*, *eq*), *context*)(*dict*) ≙ **(3.4.7.1.16)**

```
1  (let (eql1, dict') = transform-restrictions(eql, context)(dict) in
2   let mk-Equation0(term1, term2) = insert-equals-true(eq)(dict) in
3   let (as1term1, as1term2, dict'') = transform-lhs-rhs(term1, term2, context)(dict') in
4   let eq1 = mk-Unquantified-equation1(as1term1, as1term2) in
5   build-quant-equation(mk-Conditional-equation1(eql1, eq1))(dict''))
```

type: *Condequation*$_0$ *Context* → *Dict* → *Equation*$_1$

目的　把一个条件等式变换到 AS_1

参数

eql, *eq*　条件等式中的限制条件和被限制的等式

结果　构造好的 AS_1 条件等式

算法

第 1 行　变换这些限制条件。

第 2 行　分解这个 AS_0 等式。

第 3 行　变换 AS_0 被限制的等式中的两个项。注意，在被限制的等式的变换中要用 *dict'* 使得这两部分中的隐式值标识符相同。

第 4 行　构造一个 AS_1 等式。

第 5 行　用量化封闭所构造的等式（如果 *dict''* 包含任何值标识符描述符 *ValueidD*）。

transform-restrictions(*eql*, *context*)(*dict*) ≙ **(3.4.7.1.17)**

```
1  (if eql = ⟨⟩ then
2     ({}, dict)
3    else
4     (let mk-Equation0(term1, term2) = insert-equals-true(hd eql)(dict) in
5      if is-Errorterm0(term1) ∨ is-Errorterm0(term2) then
6        exit("§5.4.1.7: 错误项是限制条件的一部分")
7       else
8        (let (as1term1, as1term2, dict') = transform-lhs-rhs(term1, term2, context)(dict) in
9         let (eqlrest, dictrest) = transform-restrictions(tl eql, context)(dict') in
10        (eqlrest ∪ {mk-Unquantified-equation1(as1term1, as1term2)}, dictrest))))
```

type: *Unquantequation*$_0$* *Context* → *Dict* → *Equation*$_1$-**set** *Dict*

目的　　把一个限制条件表变换成 AS_1 限制条件的集合，办法是通过递归地遍历此表并在值标识符的字典 *Dict* 中收集信息。这个函数与变换等式函数 *transform-equation* 类似，但这里是被运用于等式表上。

$$transform\text{-}mapping(\mathbf{mk}\text{-}Mappingaziom_0(vl, tid, axl))(dict) \triangleq \quad (3.4.7.1.18)$$

```
1  (let tqual = get-visible-qual(tid, TYPE)(dict) in
2   let qualset = {dict(SCOPEUNIT) ↷ ⟨(VALUE, vl[i])⟩ | 1 ≤ i ≤ len vl} in
3   let litset = {qual ∈ dom dict | is-LiteralD(dict(qual)) ∧ s-Sortqual(dict(qual)) = tqual} in
4   transform-mappingazioms(qualset, tqual, litset, axl)(dict))
```

type：　$Mappingaziom_0 \to Dict \to Equations_1$

目的　　把一个映射公理变换为一个 AS_1 等式的集合

参数

vl	映射公理中的值名表
tid	这些值名的类别
axl	在映射公理中所包含的一些公理

结果　　变换过的 AS_1 等式

算法

第 1 行	抽取这个类别的限定表 *Qual*。
第 2 行	构造这些值名的限定表 *Qual* 的集合。
第 3 行	抽取为类别 *tqual* 定义的字面值的限定表 *Qual* 的集合。
第 4 行	进行变换

$$transform\text{-}mappingazioms(qualset, tqual, litset, axl)(dict) \triangleq \quad (3.4.7.1.19)$$

```
1  if qualset = {} then
2    {}
3   else
4    (let qual ∈ qualset in
5     union {transform-axioms(axl, MAPPING)(dict + [qual ↦ mk-ValueidD(litqual, {tqual}, true)]) |
6            litqual ∈ litset} ∪
7     transform-mappingazioms(qualset \ {qual}, tqual, litset, axl)(dict))
```

type：　*Qual*-**set** *Sortqual Qual*-**set** $Axiom_0^* \to Dict \to Equations_1$

目的　　利用所给的值标识符的集合、它们的类别和为那个类别定义的字面值，通过用字面值来替换那些公理中所有的值标识符的办法，扩展一个等式的集合

参数

qualset	表示值标识符的限定表 *Qual* 的集合
tqual	表示它们类别标识符的限定表 *Qual*
litset	表示为 *tqual* 定义的字面值的 *Qual* 的集合
axl	要被扩展的公理

结果　　AS_1 等式的集合，其中对应于 *qualset* 的值标识符已被替换

算法

第 1 行	当全部处理完时，返回等式的空集。
第 4 行	取出集合中的下一个值标识符。

第 5—6 行　　构造由所有这些等式集合的并形成的等式集合。在这些等式集合中，每个值标识符被 *litset* 中一个特定字面值所替换。

第 7 行　　返回这个构造好的等式集合，联合替换值标识符的其余部分所产生的等式一起返回。

$add\text{-}equality(tqual)(dict) \triangleq$ (3.4.7.1.20)

```
 1  (let bqual = get-predef-sort("BOOLEAN")(dict) in
 2   let bid = as0-id(bqual) in
 3   let tid = as0-id(tqual) in
 4   let opl = ⟨mk-Opspec0(EQ, ⟨tid, tid⟩, bid), mk-Opspec0(NE, ⟨tid, tid⟩, bid)⟩ in
 5   let anm = create-unique-name(),
 6       bnm = create-unique-name(),
 7       cnm = create-unique-name(),
 8       trueid = mk-Id0(bqual, mk-Name0("TRUE", nil)) in
 9   let aid = mk-Id0(⟨⟩, anm),
10       bid = mk-Id0(⟨⟩, bnm),
11       cid = mk-Id0(⟨⟩, cnm) in
12   let eq1 = mk-Equation0(mk-Infixterm0(aid, EQ, aid), trueid),
13       eq2 = mk-Equation0(mk-Infixterm0(aid, EQ, bid), mk-Infixterm0(bid, EQ, aid)),
14       eq3 = mk-Equation0(mk-Infixterm0(mk-Infixterm0(mk-Infixterm0(aid, EQ, bid), AND,
15                                                      mk-Infixterm0(bid, EQ, cid)),
16                                              IMPLY, mk-Infixterm0(aid, EQ, cid)), trueid),
17       eq4 = mk-Equation0(mk-Infixterm0(aid, NE, bid),
18                          mk-Monadterm0(NOT, mk-Infixterm0(aid, EQ, bid))),
19       eq5 = mk-Condequation0(⟨mk-Equation0(mk-Infixterm0(aid, EQ, bid), trueid)⟩,
20                              mk-Equation0(aid, bid)) in
21   (opl, ⟨mk-Quantequation0(⟨anm, bnm, cnm⟩, tid, ⟨eq1, eq2, eq3, eq4, eq5⟩)⟩))))
```

type: $Sortqual \rightarrow Dict \rightarrow Opspec_0^{+}\ Axiom_0^{+}$

目的　　对一给定类别，构造 AS_0 中“相等”的性质，参见在 Z.100 的 § 5.4.1.4 中的定义

参数

tqual　　类别的限定表 *Qual*

结果　　一个 AS_0 运算符定义表和一个 AS_0 公理表

算法

第 1—3 行　　构造布尔类别和 *tqual* 的 AS_0（受限的）标识符。

第 4 行　　构造下列定义：

```
"="  : tqual, tqual -> boolean
"/=" : tqual, tqual -> boolean
```

第 5—7 行　　起一些不同的名字，以便用于量化中。

第 8 行　　构造布尔字面值 TRUE 的受限定的 AS_0 标识符。

第 9—11 行　　为值名字构造 AS_0 标识符。

第 12—20 行　　构造五条公理（*eq*1—*eq*5）：

```
aid = aid                        == true;
aid = bid                        == bid = aid;
(aid = bid) and (bid = cid) =>
                  aid = cid      == true;
aid /= bid                       == not(aid = bid);
aid = bid == true ==>  aid       == bid;
```

第 21 行　　返回构造过的运算符定义和公理。

$$add\text{-}ordering(\mathbf{mk}\text{-}Properties_0(lit, oplist, axioms, map, init), tqual)(dict) \triangleq \qquad (3.4.7.1.21)$$

```
 1  (if (∃op ∈ elems oplist)(is-Ordering0(op)) then
 2     (let oplist' = ⟨oplist[i] | 1 ≤ i ≤ len oplist ∧ is-Opspec0(oplist[i])⟩ in
 3      if len oplist = len oplist' − 1 then
 4        (let ordaxioms = as0-ordering-axioms(tqual)(dict),
 5             order = if len lit ≤ 1 then ⟨⟩ else as0-order(hd lit, tl lit, tqual),
 6             ordtyping = as0-ordering-typing(tqual)(dict) in
 7         mk-Properties0(lit, ordtyping ⌢ oplist', ordaxioms ⌢ axioms ⌢ order, map, init))
 8       else
 9        exit("§2.2.2: 多重定义有序运算符"))
10    else
11     mk-Properties0(lit, oplist, axioms, map, init))
```

type: $Properties_0\ Sortqual \to Dict \to Properties_0$

目的　　把某些性质中有序的命令扩展为运算符定义和公理，参见 Z.100 的 § 5.4.1.8 中的定义

参数

lit	字面值定义
oplist	运算符定义
axioms	公理
map	映射公理
init	缺省值
tqual	表示定义这些性质的类别

结果　　修改后的性质

算法

第 1 行	如果在运算符定义中存在一个有序的命令，则
第 2 行	构造一运算符定义表，其中消除了有序的命令。
第 3 行	在运算符表中必须只存在一条这样的命令。
第 4 行	构造有序运算符的 AS_0 性质。
第 5 行	如果此类别定义了任何字面值，则构造反映字面值次序的公理。
第 6 行	为有序运算符构造定义。
第 7 行	返回修改后的性质。

as_0-*ordering-axioms*(*squal*)(*dict*) $\triangleq$ **(3.4.7.1.22)**

```
 1  (let bqual = get-predef-sort("BOOLEAN")(dict) in
 2   let falseid = mk-Id0(bqual, mk-Name0("FALSE", nil)) in
 3   let sid = as0-id(squal) in
 4   let anm = create-unique-name(),
 5       bnm = create-unique-name(),
 6       cnm = create-unique-name() in
 7   let aid = mk-Id0(⟨⟩, anm),
 8       bid = mk-Id0(⟨⟩, bnm),
 9       cid = mk-Id0(⟨⟩, cnm) in
10   let axioml =
11       ⟨mk-Equation0(mk-Infixterm0(aid, LT, aid), falseid),
12        mk-Equation0(mk-Infixterm0(aid, LT, bid), mk-Infixterm0(bid, GT, aid)),
13        mk-Equation0(mk-Infixterm0(aid, LE, bid), mk-Infixterm0(mk-Infixterm0(aid, LT, bid),
14                                                                OR,
15                                                                mk-Infixterm0(aid, EQ, bid))),
16        mk-Equation0(mk-Infixterm0(aid, GE, bid), mk-Infixterm0(mk-Infixterm0(aid, LE, bid),
17                                                                OR,
18                                                                mk-Infixterm0(aid, EQ, bid))),
19        mk-Infixterm0(mk-Infixterm0(aid, LT, bid),
20                      IMPLY,
21                      mk-Monadexpr0(NOT, mk-Infixterm0(aid, LT, bid))),
22        mk-Infixterm0(mk-Infixterm0(mk-Infixterm0(aid, LT, bid),
23                                    AND, mk-Infixterm0(bid, LT, cid)),
24                      IMPLY,
25                      mk-Infixterm0(aid, LT, cid))⟩ in
26   ⟨mk-Quantequation0(⟨anm, bnm, cnm⟩, sid, axioml)⟩)
```

type: *Sortqual* → *Dict* → $Axiom_0$

目的

构造反映有序运算符性质的 AS_0 公理，即

```
FOR ALL aid,bid,cid IN sid
(aid < aid  == False;
 aid < bid  == bid > aid;
 aid <= bid == (aid < bid) OR (aid = bid);
 aid >= bid == (aid > bid) OR (aid = bid);
 aid < bid  => not(bid < aid);
 aid < bid  and bid < cid => aid < cid);
```

算法

第 11 行　　构造第一个等式。

第 12 行　　构造第二个等式。

第 13 行　　构造第三个等式。

第 16 行　　构造第四个等式。

第 19 行　　构造第五个等式（即一布尔公理）。

第 22 行　　构造第六个等式（即一布尔公理）。

第 26 行　　构造并返回包含此六个等式的量化的等式。

as_0-*ordering-typing*(*tqual*)(*dict*) ≙ **(3.4.7.1.23)**

```
1  (let tid = as0-id(tqual),
2       bid = as0-id(get-predef-sort("BOOLEAN")(dict)) in
3   ⟨mk-Opspec0(LT, ⟨tid, tid⟩, bid),
4    mk-Opspec0(GT, ⟨tid, tid⟩, bid),
5    mk-Opspec0(LE, ⟨tid, tid⟩, bid),
6    mk-Opspec0(GE, ⟨tid, tid⟩, bid)⟩)
```

type: $Name_0 \rightarrow Dict \rightarrow Opspec_0{}^+$

目的 构造有序运算符的 AS_0 定义，即：

```
"<"  : tid, tid -> bid;
">"  : tid, tid -> bid;
"<=" : tid, tid -> bid;
">=" : tid, tid -> bid;
```

as_0-*order*(*lit*, *litlist*, *squal*) ≙ **(3.4.7.1.24)**

```
1  (if litlist = ⟨⟩ then
2     ⟨⟩
3   else
4    (if is-Nmclass0(lit) then
5       as0-order(hd litlist, tl litlist, squal)
6     else
7      if is-Nmclass0(hd litlist) then
8        as0-order(lit, tl litlist, squal)
9      else
10      (let id1 = if is-Name0(lit) then
11                   mk-Id0(squal, lit)
12                 else
13                   mk-Stringterm0(squal, lit),
14           id2 = if is-Name0(hd litlist) then
15                   mk-Id0(squal, hd litlist)
16                 else
17                   mk-Stringterm0(squal, hd litlist) in
18       ⟨mk-Infixterm0(id1, LT, id2)⟩ ⌢
19       as0-order(hd litlist, tl litlist, squal))))
```

type: $Literal_0\ Literal_0{}^*\ Sortqual \rightarrow Axiom_0{}^*$

目的 构造反映某类别字面值次序的公理

参数

lit	字面值表中的第一个字面值
litlist	字面值表的其余部分（此函数是递归的）
squal	定义这些字面值的类别的限定表 *Qual*

结果 一个公理表，其形式为：

id1 < id2

算法

第 1 行 当全都处理完时，没有东西返回。

第 4—5 行 如果当前字面值是一个名字类，则跳过它。

第 7—8 行　　如果字面值表中下一个字面值是名字类，则跳过它。

第 10—14 行　　构造当前字面值和表中下一字面值的 AS_0 标识符。请注意，全部限定词都出现，以便避免二义性。

第 18—19 行　　返回此字面值的布尔公理并与余下的字面值的布尔公理拼接起来。

transform-typing(*oplist*)(*dict*) $\triangleq$ (3.4.7.1.25

```
1  (if oplist = ⟨⟩ then
2     ({}, [])
3   else
4    (let mk-Opspec0(nm, opsortl, ressort) = hd oplist in
5     let (oprest, drest) = transform-typing(tl oplist)(dict) in
6     let tqual = get-visible-qual(ressort, TYPE)(dict) in
7     let pqual = get-parent(tqual)(dict) in
8     let as1id = make-as1-identifier(tqual)(dict) in
9     let arglist = ⟨get-visible-qual(opsortl[i], TYPE)(dict) | 1 ≤ i ≤ len opsortl⟩ in
10    let parglist = ⟨get-parent(arglist[i])(dict) | 1 ≤ i ≤ len arglist⟩ in
11    let opparml = ⟨make-as1-identifier(arglist[i])(dict) | 1 ≤ i ≤ len arglist⟩ in
12    let opqual = dict(SCOPEUNIT) ⌢ ⟨(OPERATOR, (nm, parglist, pqual))⟩ in
13    let newnm = create-unique-name() in
14    let newqual = dict(SCOPEUNIT) ⌢ ⟨(OPERATOR, (newnm, parglist, pqual))⟩ in
15    let descr = mk-OperatorD(arglist, tqual, newqual, true) in
16    let as1tree = mk-Operator-signature1(name-to-name1(newnm), opparml, as1id) in
17    if opqual ∈ dom drest then
18      exit("§2.2.2: 运算符定义的多重出现")
19     else
20      ({as1tree} ∪ oprest, drest + [opqual ↦ descr])))
```

type: $Opspec_0^* \to Dict \to Operator\text{-}signature_1\text{-}set\ Dict$

目的　　把运算符定义表变换到 AS_1，并且更新字典 *Dict* 以包括这些运算符的描述符

参数

oplist　　AS_0 运算符定义表

结果　　一组 AS_1 运算符定义和更新过的 *dict*

算法

第 1 行　　当全部处理完时，返回空的实物（本函数是递归的）。

第 4 行　　分解在表中的第一个运算符定义。

第 5 行　　变换其余的运算符定义。

第 6—7 行　　抽取结果类别的限定表 *Qual* 和取出在同义类型情况下的父辈类别（*pqual*）。

第 8 行　　构造结果类别的 AS_1 标识符。

第 9—10 行　　抽取变元类别的限定表 *Qual* 和取出在任何同义类型情况下的父辈类别（*parglist*）。

第 11 行　　构造变元类别的 AS_1 标识符表。

第 12 行　　构造运算符的限定表 *Qual*。请注意，运算符限定元素 *Operatorqualelem*（参见该域定义）包含父辈类别（没有同义类型）使得运算符的唯一性可以通过检查运算符的限定表 *Qual* 得以验证。

第 13 行　　为运算符构造唯一的名字。（为所有运算符产生唯一的名字，使得在 AS_1 中不发生超载现象。）

第 14 行　　构造此运算符的 *Qual* 用于把此运算符的应用性出现变换到 AS_1。

第 15 行　　为运算符构造 *Dict* 描述符。这一次，变元和结果可以是同义类型的。

第 16 行　　构造运算符的 AS_1 标记（signature）。

第 17 行　　如果此运算符的限定表 *Qual* 出现在其余运算符对字典 *Dict* 的贡献中，则具有相同名字、相同的变元类别和相同结果的两个运算符被定义在同一类别定义中。

第 20 行　　返回此运算符的标记，并和其余运算符的标记合并，并且返回此运算符对字典 *Dict* 的贡献和其余运算符对字典 *Dict* 的贡献。

3.4.7.2　结构定义

$$\mathit{transform\text{-}struc}(nm, \mathbf{mk}\text{-}\mathit{Struc}_0(fl), \mathbf{mk}\text{-}\mathit{Properties}_0(l, o, a, am, init))(dict) \triangleq \qquad (3.4.7.2.1)$$

```
1  (let sid = mk-Id0(dict(SCOPEUNIT), nm) in
2   let (o', a', ) = transform-fields(sid, fl, {}, ⟨⟩) in
3   let prop' = mk-Properties0(l, o ⌢ o', a ⌢ a', am, init) in
4   transform-sortdef(nm, prop', nil)(dict))
```

type：　$Name_0\ Struc_0\ Properties_0 \rightarrow Dict \rightarrow Dict$

目的　　把一个结构 STRUCT 的类别定义变换到 AS_1

参数

nm	被定义的类别名
fl	字段表
l, *o*, *a*, *am*, *init*	在 ADDING 构体中规定的字面值、运算符、公理、映射公理和缺省值

结果　　更新过的 *Dict* 以包括字面值描述符和运算符描述符，还包括在其数据类型定义 **DATATYPEDEF** 项目中产生的 AS_1 实物。

算法

第 1 行　　构造被 定义类别的 AS_0 标识符。

第 2 行　　生成 AS_0 运算符定义和隐含在结构定义中的公理。

第 3 行　　构造包含隐含性质和附加性质的 AS_0 性质。

第 4 行　　用通常的方法变换那些性质。

transform-fields(*tid*, *flist*, *fset*, *sidlist*) ≙ (3.4.7.2.2)

```
1  (if flist = ⟨⟩ then
2     (⟨mk-Opspec0(mk-Name0("MAKE", EXCLAMATION), sidlist, tid)⟩, ⟨⟩, len sidlist)
3   else
4    (let mk-Fieldspec0(fnml, fid) = hd flist in
5     let nm = hd fnml in
6     if nm ∈ fset then
7       exit("§5.4.1.10: 两个结构字段具有相同的名字")
8      else
9       (let flist' = if tl fnml = ⟨⟩ then
10                      tl flist
11                    else
12                      ⟨mk-Fieldspec0(tl fnml, fid)⟩ ⌢ tl flist in
13       let (as0oprest, as0axrest, argnum) =
14           transform-fields(tid, flist', fset ∪ {nm}, sidlist ⌢ fid) in
15       let (as0op, as0ax) = transform-field(tid, fid, nm, argnum, len sidlist + 1) in
16       (as0op ⌢ as0oprest, as0ax ⌢ as0axrest, argnum))))
```

type: $Sortid_0\ Fieldspec_0^*\ Fieldname_0\text{-set}\ Sortid_0^* \to Op_0^*\ Axiom_0^*\ N_1$

目的 把来自一 AS_0 结构定义的字段表变换为运算符定义表和公理表

参数

tid	定义了该结构的类别的 AS_0 标识符
flist	字段表
fset	已经使用的字段名字集合（此函数是递归的）
sidlist	字段的类别的当前表，这个表用于构造 MAKE！运算符的标记

结果 AS_0 运算符定义、由字段表隐含的公理和字段的完整数，该数仅用于此函数内部

算法

第 1—2 行	当全部处理完时，则类别表 *sidcist* 是完全的并且可以构造 MAKE! 运算符的标记。*sidlist* 的长度也被返回，使得包括 **MAKE!** 运算符的公理（构造在其它递归层上的）可被适当地建立起来。
第 4 行	分解第一字段的规格。
第 5 行	设 *nm* 表示第一个字段规格的字段名字表中的第一个名字。
第 6 行	如果已经使用了此名字，则是一个错误。
第 9 行	构造新的字段表，其中排除当前的名字。
第 13 行	变换其余的字段。
第 15 行	为当前字段构造运算符定义和公理。
第 16 行	返回当前字段的性质，并且与其余字段性质拼接起来。

transform-field(*tid*, *fid*, *fnm*, *totnum*, *index*) ≙ (3.4.7.2.3)

```
1  (let (modop, modax) = as0-modify-properties(tid, fid, fnm, totnum, index),
2       (extop, extax) = as0-extract-properties(tid, fid, fnm, totnum, index) in
3   (⟨modop ⌢ extop⟩, ⟨modax ⌢ extax⟩))
```

type: $Sortid_0\ Sortid_0\ Fieldname_0\ N_1\ N_1 \to Op_0^*\ Axiom_0^*$

目的 构造附属于一字段名的隐含的运算符定义和公理

参数

tid	定义了该结构的类别的 AS_0 标识符。
fid	字段类别的 AS_0 标识符
fnm	字段名
totum	类别 *tid* 的字段总数
index	在附属于 *tid* 的 MAKE! 运算符的变元表中字段的位置

结果 附加给字段 *fnm* 的运算符定义和公理

算法

第 1 行	为附属于字段的修改运算符构造运算符定义和公理。
第 2 行	为附属于字段的抽取运算符构造运算符定义和公理。
第 3 行	返回这两个运算符定义和两个公理。

as_0-*modify-properties*(*tid*, *fid*, **mk**-$Name_0$(*fst*,), *totlistlength*, *index*) $\triangleq$ **(3.4.7.2.4)**

```
1  (let mnm = mk-Name0(fst ⌢ "MODIFY", EXCLAMATION) in
2   let mk-Id0(qualifier, tnm) = tid in
3   let tqual = qualifier ⌢ ⟨(TYPE, tnm)⟩ in
4   let opspec = mk-Opspec0(mnm, ⟨tid, fid⟩, tid) in
5   let fvarid = mk-Id0(⟨⟩, create-unique-name()) in
6   let makearglist = ⟨mk-Id0(⟨⟩, create-unique-name()) | 1 ≤ i ≤ totlistlength⟩ in
7   let makearglist' = ⟨makearglist[i] | 1 ≤ i < index⟩ ⌢
8       ⟨fvarid⟩ ⌢
9       ⟨makearglist[i] | index < i ≤ totlistlength⟩ in
10  let makenm = mk-Name0("MAKE", EXCLAMATION) in
11  let makeop = mk-Operatorterm0(mk-Id0(tqual, makenm), makearglist),
12      makeop' = mk-Operatorterm0(mk-Id0(tqual, makenm), makearglist') in
13  let lterm = mk-Operatorterm0(mk-Id0(tqual, mnm), ⟨makeop, fvarid⟩) in
14  let ax = mk-Equation0(lterm, makeop') in
15  (opspec, ax))
```

type : $Sortid_0\ Sortid_0\ Fieldname_0\ N_1\ N_1 \rightarrow Op_0\ Axiom_0$

目的 为附属于一个结构字段的修改运算符构造 AS_0 性质

参数

tid	定义了该结构的类别标识符
fid	字段类别的标识符
fst	字段名的拼字
totlistlength	为类别 *tid* 定义的字段的数目
index	在附属于 *tid* 的 MAKE! 运算符的变元表中字段的位置

结果 隐含的运算符定义和公理，参见 Z.100 的 § 5.4.1.10 中的定义

算法

第 1 行　　通过拼接字段名字的拼字和串“MODIFY”来构造修改运算符的名字。

第 2—3 行　　构造表示类别 *tid* 的限定符（*tqual*）。MAKE! 运算符由 *tqual* 来限定，以便避免二义性。

第 4 行　　为字段修改运算符构造运算符定义。

第 5—6 行　　构造一个独自的值标识符（*fvarid*）并产生一个由一些不同的值标识符组成的表，以便用作为（左边）MAKE! 运算符（见 Z.100 的 § 5.4.1.10 中的例子）的变元。

第 6—9 行　　通过用 *fvarid* 来替换在位置 *index* 的值标识符来修改变元表。此变元表用在公理的右边。

第 10 行　　构造 MAKE! 运算符的名字 $Name_0$。

第 11 行　　为左边的 MAKE! 运算符构造 AS_0 运算符应用。

第 12 行　　为右边的项构造 AS_0 运算符应用。

第 13 行　　构造左边的项。

第 14—15 行　　返回运算符定义和构造的公理。

as_0-*extract-properties*(*tid*, *fid*, **mk**-$Name_0$(*fst*,), *totlistlength*, *index*) ≙ **(3.4.7.2.5)**

```
1  (let enm = mk-Name0(fst ⌢ "EXTRACT", EXCLAMATION) in
2   let mk-Id0(qualifier, tnm) = tid in
3   let tqual = qualifier ⌢ ⟨(TYPE, tnm)⟩ in
4   let opspec = mk-Opspec0(enm, ⟨tid⟩, fid) in
5   let makearglist = ⟨mk-Id0(⟨⟩, create-unique-name()) | 1 ≤ i ≤ totlistlength⟩ in
6   let makenm = mk-Name0("MAKE", EXCLAMATION) in
7   let makeop = mk-Operatorterm0(mk-Id0(tqual, makenm), makearglist) in
8   let lterm = mk-Operatorterm0(mk-Id0(tqual, enm), ⟨makeop⟩) in
9   let ax = mk-Equation0(lterm, makearglist[index]) in
10  (opspec, ax))
```

type: $Sortid_0\ Sortid_0\ Fieldname_0\ N_1\ N_1 \rightarrow Op_0\ Axiom_0$

目的　　为附属于一个结构的字段的抽取运算符构造 AS_0 的性质

参数

tid　　定义了该结构的类别标识符

fid　　字段类别的标识符

fst　　字段名的拼字

totlistlength　　为类别 *tid* 定义的字段数

index　　在附属于 *tid* 的 MAKE! 运算符的变元表中字段的位置

结果　　隐含的运算符定义和公理，参见 Z.100 的 § 5.4.1.10 中的定义

算法

第 1 行　　用通过拼接字段名字的拼字和串“EXTRACT”来构造抽取运算符的名字。

第 2—3 行　　构造表示类别 *tid* 的限定符（*tqual*）。

第 4 行　　为字段抽取运算符构造运算符定义。

第 5—6 行　　构造一独自的值标识符（*fvarid*），并产生一个由不同的值标识符组成的表，用作为（左边）MAKE! 运算符（见 Z.100 的 § 5.4.1.10 中的例子）的变元。

第 6—7 行　　构造左边 MAKE! 运算符的应用。

第 8 行　　构造左边的项。

第 10—11 行　　返回构造的运算符定义和公理。

3.4.8 定时器定义

transform-timerdef(**mk-***Timerdef*$_0$(*vlist*))(*dict*) ≙ (3.4.8.1)

```
1  if (∃squal ∈ dom dict)(is-ServiceD(dict(squal)) ∧ get-sur(squal) = dict(SCOPEUNIT)) then
2    exit("§4.10.2: 进程包括服务定义和定时器定义两类定义")
3   else
4    (let mk-Timerelem0(tnm, sortlist) = hd vlist in
5     let tqual = dict(SCOPEUNIT) ⌢ ⟨(SIGNAL, tnm)⟩ in
6     let unm = if is-ServiceD(dict(dict(SCOPEUNIT))) then create-unique-name() else tnm in
7     let nqual = dict(SCOPEUNIT) ⌢ ⟨(SIGNAL, unm)⟩ in
8     let squallist = ⟨get-visible-qual(sortlist[i], TYPE)(dict) | i ∈ ind sortlist⟩ in
9     let as1 idlist = ⟨make-as1-identifier(squallist[i])(dict) | i ∈ ind squallist⟩ in
10    let as1 tree = mk-Timer-definition1(name-to-name1(unm), as1 idlist),
11        delem = [tqual ↦ mk-TimerD(squallist, nqual) | 1 ≤ i ≤ len squallist] in
12    if tl vlist = ⟨⟩ then
13      (⟨as1 tree⟩, delem)
14     else
15      (let (as1 rest, drest) = transform-timerdef(mk-Timerdef0(tl vlist))(dict) in
16       ({as1 tree} ∪ as1 rest, delem + drest)))
```

type: *Timerdef*$_0^+$ → *Dict* → *Timer-definition*$_1$-**set** *Dict*

目的

把一定时器定义变换到 AS_1

参数

vlist 包含在（组合的）定时器定义中的（基本）定时器定义表

结果

AS_1 定时器定义的集合和对字典 *Dict* 的贡献，它包含这些定时器的描述符

算法

第 1—2 行 外围进程一定不含任何服务。

第 4 行 分解下一个基本定时器的定义（此函数是递归的）。

第 5 行 构造此定时器的限定表 *Qual*。

第 6—7 行 构造唯一的名字和唯一的 *Qual* 用于导出 AS_1 标识符（参见 *Newqual* 定义）。

第 8—9 行 构造附属于定时器的类别的 *Qual* 表，并构造对应的 AS_1 类别标识符表。

第 10 行 为当前的定时器构造 AS_1 定时器定义。

第 11 行 为当前的定时器构造对字典 *Dict* 的贡献。

第 12 行 如果该基本定时器定义是表中最后一个元素，则返回 AS_1 定义和对字典 *Dict* 的贡献。否则

第 15—16 行 把它们和对应于其余基本定时器的定义和对 *Dict* 的贡献汇合到一起。

3.4.9 变量定义

transform-vardef(**mk-***Vardef*$_0$(*revatt*, *expatt*, *vlist*))(*dict*) ≙ (3.4.9.1)

```
1  (let mk-Vardefelem0(vl, tp, expr) = hd vlist in
2   let tqual = get-visible-qual(tp, TYPE)(dict) in
3   let pq = get-parent(tqual)(dict) in
4   let expr' = if expr = nil then
5                 cases dict(tqual):
6                  (mk-SortD(, , expr1, )    → expr1,
```

```
 7                  mk-SyntypeD(, , expr1, ) → expr1)
 8              else
 9               (let (expr1, , ) = transform-expr(expr, CONSTANT, {pq})(dict) in
10                expr1) in
11  let vnum = len vl in
12  let scope = dict(SCOPEUNIT) in
13  let nmlist = ⟨if is-ServiceD(dict(scope)) then create-unique-name() else vl[i] | 1 ≤ i ≤ vnum⟩ in
14  let quallist = ⟨scope ⌢ ⟨(VALUE, vl[i])⟩ | 1 ≤ i ≤ vnum⟩,
15      newquallist = ⟨scope ⌢ ⟨(VALUE, nmlist[i])⟩ | 1 ≤ i ≤ vnum⟩ in
16  let (decll, dexport) = if expatt = nil then
17                             ({}, [])
18                         else
19                          build-exported-vardef(vl, tqual, expr)(dict) in
20  let (as1 set, delem) =
21      (let as1 tp = make-as1-identifier(tqual)(dict) in
22       let as1 dl = {mk-Variable-definition1(name-to-name1(nmlist[i]), as1 tp, revatt) | i ∈ ind vnum},
23           d = [quallist[i] ↦ mk-VarD(tqual, revatt, expatt, expr', newquallist[i]) | 1 ≤ i ≤ vnum] in
24       (as1 dl, d)) in
25  if ¬is-ProcessD(dict(scope)) ∧ (revatt ≠ nil ∨ expatt ≠ nil) then
26    exit("§2.4.5: 在过程(或服务)中的变量是不能被出口的(EXPORTED)或被透露的(REVEALED)")
27   else
28    if card elems vl ≠ len vl then
29      exit("§2.2.2: 在相同的作用域单位中的两个定义使用相同的名字")
30     else
31      if tl vlist = ⟨⟩ then
32        (as1 set ∪ decll, delem + dexport)
33       else
34        (let (as1 rest, drest) =
35             transform-vardef(mk-Vardef0(revatt, expatt, tl vlist))(dict) in
36         if dom delem ∩ dom drest ≠ {} then
37           exit("§2.2.2: 在相同的作用域单位中的两个定义使用相同的名字")
38          else
39           (as1 set ∪ decll ∪ as1 rest, delem + drest + dexport)))
```

type: $Vardef_0 \to Dict \to Variable\text{-}definition_1\text{-}\mathbf{set}\ Dict$

目的

把一变量定义变换到 AS_1

变量定义包含

参数

revatt 任选的 REVEAL 属性

expatt 任选的 EXPORT 属性

vlist （基本）变量定义表

结果

AS_1 变量定义的集合和对字典 *Dict* 的贡献，它包含定义的变量的描述符

算法

第 1 行 分解第一个基本变量定义（本函数是递归的）。

第 2 行 抽取它们类别的限定表 *Qual*。

第 3 行 抽取父辈的限定表 *Qual*（在同义类型的情况下）。

第 4—10 行 如果在变量定义中没有规定初始表达式，则为其类别抽取 AS_1 缺省表达式(也可能是 **nil**)。否则变换所规定的表达式。

第 11 行 令 *vnum* 表示在基本变量定义中引入的变量名字的数目。

第 12 行 令 *scope* 表示定义这些变量的作用域单位的限定表 *Qual*。

第 13 行　　如果变量是在一服务中定义的，则为它构造新的唯一名字。
第 14 行　　为这些变量构造变量的 *Qual* 用作字典 *Dict* 的项目。
第 15 行　　构造要被放入变量描述符的 *Newqual* 成分中的 *Qual*（如果在服务中的话，这些 *Qual* 不同于在 14 行构造的 *Qual*）。
第 16—19 行　　如果这些变量有 EXPORT 属性，则为那些隐式变量构造 AS_1 定义（*decll*）。
第 20 行　　构造对于变量的 AS_1 变量定义和对字典 *Dict* 的贡献，方法是：
构造它们类别的 AS_1 标识符。
第 21 行　　为每个变量构造 AS_1 变量定义，包含名字（*nmlist* [*i*]）、它的类别（as_1tp）和可能的透露属性（*revatt*）。
第 23 行　　为每个变量，构造一个字典 *Dict* 的描述符，它包含它的类别的 *Qual*（*tqual*）、可能的一个透露属性（*revatt*）、可能的一个出口属性（*expatt*）、可能的一个初始 AS_1 表达式（*expr'*）和（可能是新的）用于此变量的应用性出现的 *Qual*。
第 25—26 行　　除非这些变量是在进程层定义的，否则它们不能是 **EXPORTED** 或 **REVEALED** 的。
第 28 行　　*vl* 中的名字必须是唯一的。
第 31 行　　如果在组合变量定义中没有基本变量定义，则返回所构造的实物。否则
第 34—39 行　　把它们与其余基本变量定义的 AS_1 定义和对字典 *Dict* 的贡献汇合到一起，条件为在变量定义中的名字是唯一的。

build-exported-vardef(*varlist*, *tqual*, *expr*)(*dict*) ≙ **(3.4.9.2)**

```
1  if varlist = ⟨⟩ then
2    ({}, [])
3   else
4    (let nm = hd varlist in
5     let (vdrest, drest) = build-exported-vardef(tl varlist, tqual, expr)(dict) in
6     let id0 = as0-id(tqual) in
7     let mk-SignalnamesD(impnm, , ) = exportmap((tqual, nm)) in
8     let vardef0 = mk-Vardef0(nil, nil, ⟨mk-Vardefelem0(⟨impnm⟩, id0, expr)⟩) in
9     let (vardef1, dict') = transform-vardef(vardef0)(dict) in
10    (vdrest ∪ vardef1, drest + dict'))
```

type: $Name_0^*$ *Qual* [$Expr_0$] → *Dict* → *Variable-definition*$_1$-**set** *Dict*

目的　　为由出口变量隐含的实物构造 AS_1 变量定义

参数

varlist	被出口的变量表
tqual	变量类别的限定表 *Qual*
expr	表示这些变量初值的表达式

结果　　AS_1 定义

算法

第 1 行　　当全部处理完时，没有东西返回（此函数是递归的）。
第 4 行　　令 *nm* 表示变量表中第一个（下一个）名字。
第 5 行　　为变量表的其余部分构造 AS_1 定义。
第 6 行　　（重新）构造这些变量类别的 AS_0 标识符。

第 7 行　　分解出口描述符（*signalnamesD*），以便访问隐式变量的名字。

第 8 行　　构造那个隐式名字的 AS_0 变量定义。

第 9 行　　把隐式变量定义变换到 AS_1。

第 10 行　　返回 AS_1 定义并使之和其余出口变量的定义汇合。把（包括在 *dict'* 中的）隐式变量的 *varD* 描述符包括到结果的 *Dict* 中。

3.4.10　视见定义

transform-viewdef(**mk-***Viewdef*$_0$(*vars*))(*dict*) ≙　　(3.4.10.1)

```
1  (let mk-Viewdefelem0(varlist, tp) = hd vars in
2   if card elems varlist ≠ len varlist then
3     exit("§2.2.2: 在相同作用域单位中的两个定义使用相同的名字")
4    else
5     (let tqual = get-visible-qual(tp, TYPE)(dict) in
6      let tqual' = get-parent(tqual)(dict) in
7      let (as1 set, d) = transform-view(varlist, tqual')(dict) in
8      if tl vars = ⟨⟩ then
9        (as1 set, d)
10       else
11       (let (as1 rest, drest) = transform-viewdef(mk-Viewdef0(tl vars))(dict) in
12        if dom d ∩ dom drest ≠ {} then
13          exit("§2.2.2: 在相同作用域单位中的两个定义使用相同的名字")
14         else
15          (as1 set ∪ as1 rest, d + drest))))
```

type:　*Viewdef*$_0$ → *Dict* → *View-definition*$_1$-**set** *Dict*

目的　　把一视见定义变换到 AS_1

参数

vars　　包含在（组合的）视见定义中的（基本）定义表

结果　　AS_1 视见定义的集合和包含视见变量的描述符的对字典 *Dict* 的贡献

算法

第 1 行　　分解第一个（下一个）基本视见定义（本函数是递归的）。

第 2 行　　检查变量表中名字的唯一性。

第 5 行　　抽取它们类别的 *Qual*。

第 6 行　　抽取同义类型情况下父辈类别的 *Qual*。

第 7 行　　变换第一个基本视见定义。

第 8 行　　如果没有基本的视见定义，则返回这些视见变量的 AS_1 定义和对字典 *Dict* 的贡献，否则

第 11—15 行　　就把它们和其余基本视见定义的 AS_1 定义和贡献汇合起来，条件为视见定义中的名字应是唯一的。

transform-view(*varlist*, *tqual*)(*dict*) ≙ (3.4.10.2)

```
 1  (if varlist = ⟨⟩ then
 2     ({}, [])
 3   else
 4    (let mk-Id0(q, nm) = hd varlist in
 5     let as1 tid = make-as1-identifier(tqual)(dict),
 6         (as1 rest, drest) = transform-view(tl varlist, tqual)(dict) in
 7     let qual = (let bqual = get-sur(dict(SCOPEUNIT)) in
 8                 if (∃!vqual ∈ dom dict)
 9                    ((get-sur(get-sur(vqual)) = bqual) ∧
10                     (∃q')(q' ↷ q ↷ ⟨(VALUE, nm)⟩ = vqual) ∧
11                     is-VarD(dict(vqual)) ∧
12                     (let mk-VarD(tq, attr, , ) = dict(vqual) in
13                      tq = tqual ∧ attr = REVEALED)) then
14                   (let vqual be s.t. (get-sur(get-sur(vqual)) = bqual ∧
15                                       is-VarD(dict(vqual)) ∧
16                                       (∃q')(q' ↷ q ↷ ⟨(VALUE, nm)⟩ = vqual) ∧
17                                       (let mk-VarD(tq, attr, , ) = dict(vqual) in
18                                        tq = tqual ∧ attr = REVEALED)) in
19                    vqual)
20                  else
21                   exit("§2.6.1.2: 没有和那个类别唯一对应的被透露变量")                    in
22     let as1 id = make-as1-identifier(qual)(dict) in
23     let as1 tree = mk-View-definition1(as1 id, as1 tid) in
24     ({as1 tree} ∪ as1 rest, drest + [dict(SCOPEUNIT) ↷ ⟨(VIEW, qual)⟩ ↦ mk-ViewD(qual)])))
```

type: Id_0^* *Qual* → *Dict* → *View-definition*$_1$-**set** *Dict*

目的 把一连串的视见变量变换为一些 AS_1 视见定义。

参数

varlist	一些视见变量组成的表
tqual	这些变量类别的限定表 *Qual*

结果 一些 AS_1 定义

算法

第 1 行	当全部处理完时，没有东西返回（本函数是递归的）。
第 4 行	分解第一个变量标识符。
第 5 行	把变量类别变换为一个 AS_1 标识符。
第 6 行	变换其余的视见变量。
第 7—21 行	用 *qual* 表示对应于被透露变量的限定表 *Qual*。
第 7 行	构造外围功能块的限定表 *Qual*。
第 8 行	必须存在一个唯一的变量标识符（即变量的 *Qual*）定义在同一外围功能块的一个进程中，
第 10 行	它包含在视见定义中规定的限定词和名字，
第 11 行	它是一个变量，
第 13 行	它是同类别的和被透露的。
第 14—19 行	如果存在这样一个标识符，则令 *vqual* 表示它的限定表 *Qual*。

第 22—23 行　　构造 AS_1 视见定义并

第 24 行　　把它和含有其余视见变量（$as_1 rest$）的定义汇合在一起。也把视见描述符（*viewD*）和其余的视见定义的描述符汇合在一起。

3.4.11　进口定义

$$transform\text{-}importdef(\mathbf{mk}\text{-}Importdef_0(importelemlist))(dict) \triangleq \tag{3.4.11.1}$$

```
1  (let mk-Importelem0(varl, tid) = hd importelemlist in
2   let tqual = get-visible-qual(tid, TYPE)(dict) in
3   let pqual = get-parent(tqual)(dict) in
4   let scope = dict(SCOPEUNIT) in
5   let delem = [scope ↷ ⟨(IMPORT, (varl[i], pqual))⟩ ↦ mk-ImportD(tqual) | 1 ≤ i ≤ len varl] in
6   if tl importelemlist = ⟨⟩ then
7     ({}, delem)
8    else
9     (let drest = transform-importdef(tl importelemlist)(dict) in
10     if dom delem ∩ dom drest ≠ {} then
11       exit("§2.2.2: 在相同的作用域单位中的两个定义使用相同的名字")
12      else
13       ({}, delem + drest)))
```

type:　$Importdef_0 \to Dict \to Decl_1\text{-set } Dict$

目的　　为进口的实体构造对字典 *Dict* 的贡献

参数　　进口定义含有

importelemlist　　（基本的）进口定义表

结果　　进口变量对字典 *Dict* 的贡献和一个空集（参见变换信号表定义函数 *transform-signallistdef*）

算法

第 1 行　　分解第一个基本进口定义（本函数是递归的）。

第 2 行　　抽取所含变量（*varl*）的类别的限定表 *Qual*。

第 3 行　　抽取在同义类型情况下的父辈类别。

第 4—5 行　　为所含变量构造对字典 *Dict* 的贡献。注意，*pqual* 用于 *Qual* 中是为了获得“进口名字与类别”偶对的唯一性。而 *tqual* 用于描述符中是因为与进口表达式相关联的隐式变量可以是同义类型的。

第 6—9 行　　并且把它们和其余基本视见定义（如果有的话）汇合到一起。

第 10 行　　“进口名字与类别”偶对必须是唯一的。

3.4.12 信号路由定义

transform-signalroutedef(**mk-***Sigroutedef*$_0$(*nm*, *path*, *opath*))(*dict*) ≙ **(3.4.12.1)**

```
 1  (let mk-Sigroutepath0(endpoint1, endpoint2, siglist) = path in
 2   let sigroutequal = dict(SCOPEUNIT) ↷ ⟨(SIGNALROUTE, nm)⟩ in
 3   let sigset = transform-signallist(siglist)(dict) in
 4   let p1 = get-visible-qual(endpoint1, PROCESS)(dict),
 5       p2 = get-visible-qual(endpoint2, PROCESS)(dict) in
 6   let osigset =
 7       if opath = nil then
 8         {}
 9        else
10         (let mk-Sigroutepath0(orig', dest', osiglist) = opath in
11          let p1' = get-visible-qual(orig', PROCESS)(dict),
12              p2' = get-visible-qual(dest', PROCESS)(dict) in
13          if p1 = p2' ∧ p2 = p1' then
14            transform-signallist(osiglist)(dict)
15           else
16            exit("§2.5.2: 第二条路径的方向必须与第一条路径的方向相反")) in
17   if p1 ≠ p2 ∧ is-local(p1)(dict) ∧ is-local(p2)(dict) then
18     (let as1p1 = if p1 = ENV then ENVIRONMENT else make-as1-identifier(p1)(dict),
19          as1p2 = if p2 = ENV then ENVIRONMENT else make-as1-identifier(p2)(dict),
20          as1sigidset = make-as1idset(sigset)(dict),
21          as1sigidset' = make-as1idset(osigset)(dict) in
22      let as1path1 = mk-Signal-route-path1(as1p1, as1p2, as1sigidset),
23          as1path2 = if as1sigidset' = {} then
24                       nil
25                     else
26                       mk-Signal-route-path1(as1p2, as1p1, as1sigidset'),
27          descr = mk-SignalrouteD(p1, p2, sigset, osigset) in
28      let as1tree = mk-Signal-route-definition1(name-to-name1(nm), as1path1, as1path2) in
29      ({as1tree}, [sigroutequal ↦ descr]))
30     else
31      if p1 = p2 then
32        exit("§2.5.2: 信号路由的各个端点应是不同的")
33       else
34        exit("§2.5.2: 信号路由的各个端点不是局部定义的"))
```

type: *Sigroutedef*$_0$ → *Dict* → *Signal-route-definition*$_1$-**set** *Dict*

目的

把一信号路由定义变换到 AS_1

参数

AS_0 信号路由定义包含

nm 信号路由的名字

path 第一条路径

opath 第二条任选的路径

结果

包含一个 AS_1 信号路由定义的集合和此信号路由描述符对字典 *Dict* 的贡献

算法

第 1 行 分解第一条路径。

第 2 行 构造表示信号路由的限定表 *Qual*。

第 3 行 把第一条路径中的信号表变换为信号限定表 *Qual* 的一个集合。

第 4—5 行 抽取第一条路径的两个端点的 *Qual*（函数 *get-visible-qual* 可以处理端点为 ENV 的情况）。

第 6—16 行　　令 *osigset* 表示相反方向（第二条路径）上信号的 *Qual* 的集合。
第 6 行　　如果没有第二条路径，则 *osigset* 为空集。否则
第 10 行　　分解第二条路径。
第 11—12 行　　抽取第二条路径的两个端点的 *Qual*。
第 13 行　　第一条路径的第一个端点必须等于第二条路径的第二个端点，而第一条路径的第二个端点必须等于第二条路径的第一个端点。
第 14 行　　令 *osigset* 表示来自第二条路径的信号的 *Qual* 集合。
第 17 行　　此信号路由的两个端点必须不同，并且这两个端点必须是局部定义的。
第 18—19 行　　构造这两个端点的 AS_1 标识符（或 ENVIRONMENT）。
第 20—21 行　　构造在两个方向上传递的信号的两个 AS_1 信号集。
第 22—23 行　　构造这两个方向的两条 AS_1 路径。
第 27 行　　构造信号路由描述符。
第 28 行　　构造 AS_1 信号路由定义
第 29 行　　返回 AS_1 定义和包含构造的描述符对字典 *Dict* 的贡献。
第 31—34 行　　为清楚起见，把错误条件分裂为两个 **exit**（退出）。

3.4.13　信号表定义

$$\mathit{transform\text{-}signallistdef}(\mathbf{mk}\text{-}\mathit{Signallistdef}_0(nm, \mathit{signals}))(\mathit{dict}) \triangleq \qquad \textbf{(3.4.13.1)}$$

```
1  (let siglistqual = dict(SCOPEUNIT) ⌢ ⟨(SIGNALLIST, nm)⟩ in
2   let () =
3       transform-signallist(signals)(dict + [siglistqual ↦ mk-ErrorD()]) in
4   ({}, [siglistqual ↦ mk-SignallistD(signals)]))
```

type:　$\mathit{Signallistdef}_0 \to \mathit{Dict} \to \mathit{Decl}_1\text{-}\mathbf{set}\ \mathit{Dict}$

目的　　为信号表定义构造对字典 *Dict* 的贡献

参数　　AS_0 信号表定义包含
nm　　信号表的名字
signals　　相关的信号表

结果　　除了对 *Dict* 的贡献外，还返回 AS_1 定义的空集（使得所有的“定义变换”函数可进行同样的处理）

算法
第 1 行　　构造信号表的 *Qual*。
第 3 行　　把信号表变换为一信号的 *Qual* 集合。但结果是不被使用的，因为此函数仅仅用于检查递归定义。
第 4 行　　返回对 *Dict* 的贡献，它包括信号表描述符。

transform-signallist(*sigl*)(*dict*) ≜ (3.4.13.2)

```
1  (if sigl = ⟨⟩ then
2     {}
3    else
4     (let sigrest = if tl sigl = ⟨⟩ then {} else transform-signallist(tl sigl)(dict) in
5      let sigsetelem =
6           cases hd sigl:
7           (mk-Id0(,)
8              → {signal-qual(hd sigl)(dict)},
9            mk-Signallistid0(id)
10             → (let qual = get-visible-qual(id, SIGNALLIST)(dict) in
11                let mk-SignallistD(siglist) = dict(qual) in
12                transform-signallist(siglist)(dict + [qual ↦ mk-ErrorD()]))) in
13     if sigsetelem ∩ sigrest ≠ {} then
14       exit("§2.5.5: 信号表中的信号标识符不能区分")
15      else
16       sigsetelem ∪ sigrest))
```

type: *Signallist*$_0$ → *Dict* → *Signalqual-set*

目的 把信号表变换为一个 *Qual* 的集合

参数

sigl 信号表

算法

第 4 行 变换除表中第一个元素外的所有元素（本函数是递归的）。

第 5—12 行 令 *sigsetelem* 表示表中第一个元素的 *Qual*。

第 7 行 如果表中第一个元素是一个标识符，则它表示一个信号，而 *sigsetelem* 表示含有那个信号的 *Qual* 的集合。

第 9 行 如果表中第一个元素是一个信号表，则令 *qual* 表示该信号表的 *Qual*。

第 11 行 令 *siglist* 表示附属于此信号表标识符的 AS$_0$ 信号表。

第 12 行 变换那个信号表，但要通过把此信号表标识符表示为 *Dict* 中的一个 *ErrorD* 来使得信号表标识符无效。

第 13 行 如果从第一个元素得到的任何信号的 *Qual* 也出现在另一个元素中，则为一种错误情况。否则

第 16 行 返回来自第一个元素的集合和来自信号表的其余部分的集合的并集。

3.4.14 连接语句

transform-block-connect(*decllist*, *sigrouteset*, *connectmap*)(*dict*) ≙ (3.4.14.1)

```
 1  (if decllist = ⟨⟩ then
 2     (let bqual = dict(SCOPEUNIT) in
 3      let outerchanset = {mk-ChannelD(endp1, endp2, , , newc) ∈ rng dict |
 4                          bqual ∈ {endp1, endp2} ∧ newc = nil} in
 5      if outerchanset ≠ dom connectmap then
 6        exit("§2.5.3: 没有用于把信道连接到功能块的连接")
 7       else
 8        if sigrouteset = union rng connectmap then
 9          ({}, connectmap)
10         else
11          exit("§2.5.3: 信号路由应出现在一个且仅出现在一个连接中"))
12    else
13     ( cases hd decllist:
14       (mk-Sigroutedef0(nm, mk-Sigroutepath0(end1, end2, ), )
15          → (let qset = if end1 = ENV ∨ end2 = ENV
16                        then {dict(SCOPEUNIT) ⌢ ⟨(SIGNALROUTE, nm)⟩}
17                        else {} in
18             let newset = sigrouteset ∪ qset in
19             transform-block-connect(tl decllist, newset, connectmap)(dict)),
20        mk-Connect0(, )
21          → (let (channel, routeset, connect) = block-connect(hd decllist)(dict) in
22             if channel ∈ dom connectmap then
23               exit("§2.5.3: 信道到信号路由的连接出现两次")
24              else
25               if union rng connectmap ∩ routeset ≠ {} then
26                 exit("§2.5.3: 在多于一个连接中提及同一个信号路由")
27                else
28                 (let nconnectmap = connectmap + [channel ↦ routeset] in
29                  let (restas1, restmap) =
30                      transform-block-connect(tl decllist, sigrouteset, nconnectmap)(dict) in
31                  ({connect} ∪ restas1, restmap))),
32        T → transform-block-connect(tl decllist, sigrouteset, connectmap)(dict))))
```

type: $Decl_0^*$ *Qual*-**set** *BlockconnectionD* → *Dict* → *Channel-to-route-connection*$_1$-**set** *BlockconnectionD*

目的　把一个功能块的一些连接定义变换到 AS_1 的一组信道与路由的连接

参数

decllist　一个功能块的 AS_1 定义表

sigrouteset　表示已（递归地）处理过的信号路由的 *Qual* 的集合。在初始时，该集合为空。

connectmap　是一个映射表，从 *Qual*（表示已出现在连接定义中的信道）映射到这些信道所连接到的信号路由的 *Qual*。初始时，此映射表为空。

结果　构造的 AS_1 连接定义

算法

第 2 行　令 *bqual* 表示此功能块。

第 3 行　令 *outerchanset* 表示以此功能作为它们的一个端点的信道集合。不包括已经被两个新信道代替的信道（即 *newc*=nil）。

第 5—6 行　以此功能块作为其一个端点的信道的集合必须等于在此功能块的连接中提及的信道的集合。

第 8—11 行　　如果从功能块的信号路由定义导出的信号路由的 *Qual* 的集合与构造的集合（汇合 *connectmap* 中所包含的集合而成）相同，就返回。也就是说，在一个连接中所包含的所有信号路由都必须在此功能块中定义，且以环境作为一个端点，并且反过来也成立。

第 14—19 行　　如果在表中的第一个（下一个）定义是一信号路由定义，那么如果其一个端点是 ENV，则把信号的 *Qual*（包含在 *qset* 中）包括到 *sigrouteset* 中。

第 20 行　　如果下一个定义是一个功能块的连接，则

第 21 行　　从功能块连接求出信道的 *Qual*、信号路由的 *Qual* 和 AS_1 连接定义。

第 22 行　　如果此信道已用于功能块的连接中，则为一种错误。

第 25 行　　每个信号路由必须只在一个信道到信号路由的连接中被描述。

第 28 行　　更新连接映射表以包括当前连接的信息。

第 32 行　　如果下一个定义是其它任何事情，则继续处理其余的定义。

***block-connect*(mk-*Connect*$_0$(*chid*, *routeidlist*))(*dict*) ≙**　　**(3.4.14.2)**

```
1  (let cqual = get-visible-qual(chid, CHANNEL)(dict) in
2   let mk-ChannelD(, endpoint2, sigset1, sigset2, newc) = dict(cqual) in
3   let bqual = dict(SCOPEUNIT) in
4   let connectset = {get-visible-qual(routeidlist[i], SIGNALROUTE)(dict) | i ∈ ind routeidlist} in
5   let (insigset, outsigset) = if endpoint2 = bqual then (sigset1, sigset2) else (sigset2, sigset1) in
6   let cas1id = make-as1-identifier(cqual)(dict),
7       sas1set = {make-as1-identifier(cq)(dict) | cq ∈ connectset} in
8   let cqual' = if newc = nil then cqual else convert-channel-qual(newc, bqual) in
9   if card connectset = len routeidlist ∧
10    is-wf-connectsignalroutes(connectset, insigset, outsigset, {}, {})(dict) then
11    (cqual', connectset, mk-Channel-to-route-connection1(cas1id, sas1set))
12    else
13     exit("§2.5.3: 信道到信号路由的连接不是良形式的")
```

type:　*Connect*$_0$ → *Dict* → *Channelqual Qual*-set *Channel-to-route-connection*$_1$

目的　　从一个 AS_0 信道到路由的连接中求出该信道的 *Qual*、信号路由的 *Qual* 的集合和一个 AS_1 连接定义

参数

chid　　在连接中的 AS_0 信道标识符

routeidlist　　信号路由标识符的 AS_0 表

结果　　在包围作用域单位中的信道的 *Qual*、包含连接到那个信道的那些信号路由的 *Qual* 集合和 AS_1 连接

算法

第 1 行　　抽取信道的 *Qual*。

第 2 行　　分解信道的描述符。

第 3 行　　令 *bqual* 表示包含这些连接定义的功能块的 *Qual*。

第 4 行　　构造表示信号路由标识符的 *Qual* 集合，这些信号路由标识符包括在信道到路由的连接中。

第 5 行　　令 *insigset* 表示进入信号集合，*outsigset* 表示走出信号集合。

第 6—7 行　　构造此信道的 AS_1 标识符和包含那些信号路由的 AS_1 标识符集合。
第 8 行　　如果此信道有一个子结构，则利用那个适合替换信道的 *Qual*。
第 9 行　　一个信号路由标识符在连接定义中一定不能出现两次，而且
第 10 行　　在信号路由中的信号必须在信道中提及。
第 11 行　　返回信道的 *Qual*、信号路由的 *Qual* 的集合和 AS_1 的信道到路由的连接。

is-wf-connectsignalroutes(*connectset*, *insigset*, *outsigset*, *res1*, *res2*)(*dict*) ≙ **(3.4.14.3)**

```
1  if connectset = {} then
2    insigset = res1 ∧ outsigset = res2
3   else
4   (let squal ∈ connectset in
5    let mk-SignalrouteD(endpoint1, endpoint2, sset1, sset2) = dict(squal) in
6    if ENV ∉ {endpoint1, endpoint2} then
7      false
8     else
9      (let (inset, outset) = if endpoint1 = ENV then (sset1, sset2) else (sset2, sset1) in
10      is-wf-connectsignalroutes(connectset \ {squal}, insigset, outsigset,
11                                inset ∪ res1, outset ∪ res2)(dict)))
```

type: *Qual*-**set** *Signalqual*-**set** *Signalqual*-**set** *Signalqual*-**set** → *Dict* → *Bool*

目的　　检查在一个信道到路由的连接中的那些信号路由的接口是否适当

参数

connectset　　（余下的）信号路由的 *Qual* 的集合（本函数是递归的）。
insigset　　经由在信道到路由的连接中规定的信道进入功能块的信号的 *Qual* 的集合
outsigset　　经由在信道到路由的连接中规定的信道离开功能块的信号的 *Qual* 的集合
res1，*res2*　　在信号路由中规定递归地产生的输入信号集合和输出信号集合，最后这些集合必须等于为信道指定的相应的集合。

结果　　收集的进入信号集合（*res1*）和走出信号集合（*res2*）必须分别等于信道中规定的进入信号和离开信号。

算法

第 1 行　　当全部处理完时，如果所有来自信道的信号已在一信号路由中出现，则返回真。
第 4 行　　令 *squal* 表示信号路由集合中的一个 *Qual*。
第 5 行　　分解信号路由的 *Dict* 描述符。
第 6 行　　ENV 必须是该信号路由的端点之一。
第 9 行　　令 *inset* 表示取自进入功能块的信号路由描述符的信号集合，*outset* 表示走出功能块的信号集合。
第 10 行　　在此连接中其余的信号路由的接口必须是适当的。

transform-substructure-connect(*decllist*, *channelset*, *connectmap*)(*dict*) ≜ **(3.4.14.4)**

```
1  (if decllist = ⟨⟩ then
2     (let bqual = get-sur(dict(SCOPEUNIT)) in
3      let outerchanset = {mk-ChannelD(endp1, endp2, , , newc) ∈ rng dict |
4                          bqual ∈ {endp1, endp2} ∧ newc = nil} in
5      if card outerchanset > card dom connectmap then
6        exit("§3.2.2: 没有连接用于连接到功能块子结构的信道")
7       else
8        if channelset = union rng connectmap then
9          {}
10         else
11          exit("§3.2.2: 子信道应出现在且仅出现在一个连接中"))
12    else
13     ( cases hd decllist:
14       (mk-Chandef0(nm, mk-Chanpath0(orig, dest, ), , , )
15          → (let qset = if orig = ENV ∨ dest = ENV
16                        then {dict(SCOPEUNIT) ⌢ ⟨(CHANNEL, nm)⟩}
17                        else {} in
18             let newset = channelset ∪ qset in
19             transform-substructure-connect(tl decllist, newset, connectmap)(dict)),
20        mk-Connect0(, )
21          → (let (chan, chanset, connect) = transform-block-substructure-connect(hd decllist)(dict) in
22             if chan ∈ dom connectmap then
23               exit("§3.2.2: 信道连接出现两次")
24              else
25               if union rng connectmap ∩ chanset ≠ {} then
26                 exit("§3.2.2: 子信道在多于一个连接中被描述")
27                else
28                 (let nconnectmap = connectmap + [chan ↦ chanset] in
29                  {connect} ∪
30                  transform-substructure-connect(tl decllist, channelset, nconnectmap)(dict))),
31        T → transform-substructure-connect(tl decllist, channelset, connectmap)(dict))))
```

type: $Decl_0^*$ *Channelqual*-**set** (*Channelqual* $\overset{m}{\rightarrow}$ *Channelqual*-**set**) → *Dict* → *Channel-connection*$_1$-**set**

目的 构造附属于功能块子结构的信道连接

参数

dellist 功能块子结构的 AS_0 定义表与 AS_0 的连接汇合，此 AS_0 的连接把隐式出口—进口信道连接起来

Channelset, *Connectmap* 在递归地遍历定义表 *decllist* 期间，收集来自进入或离开环境的信道定义的信道名，并记录在信道集合 *channelset* 中，收集连接信息并记录在连接映射表 *connectmap* 中。当（在函数 *transform-blockdef* 中）最初应用函数 *transform-substructure-connect* 时，这些参数为空。

结果 收集到的 AS_1 连接

算法

第 2 行 令 *bqual* 表示包围功能块子结构的功能块。

第 3 行 令 *outerchanset* 表示以此功能块为一个端点的信道的集合，并且此功能块没有子结构。

第 5—6 行 如果以功能块为一个端点的信道多于信道连接中的信道，则缺少了某些信道连接。

第 8—11 行　　如果所有信道（其一个端点作为环境）都已出现在连接中，则返回。

第 14 行　　如果当前的定义是一个信道定义，则

第 15—18 行　　如果此信道不是来自或通向环境，则构造由空集组成的集合。否则构造由信道的 *Qual* 组成的集合。

第 19 行　　遍历定义表的其余部分，所带有的 *channelset* 中已加入了所构造的集合。

第 21 行　　如果当前的定义是一个信道连接，则把该连接变换到 AS_1，获得信道的 *Qual*（*chan*）、信道 *Qual* 的集合（*chanset*）和 AS_1 连接 *connect*。

第 22 行　　此信道一定不出现在一个以上的信道连接中。

第 25 行　　一个子信道必须只出现在一个信道连接中。

第 28—29 行　　遍历定义表的其余部分。所带有的 *connectmap* 中已加入了以上的连接信息。

transform-block-substructure-connect(**mk-***Connect*$_0$(*chid*, *chidlist*))(*dict*) ≜ **(3.4.14.5)**

```
 1  (let cqual = get-visible-qual(chid, CHANNEL)(dict) in
 2   let mk-ChannelD(endpoint1, endpoint2, sigset1, sigset2, newc) = dict(cqual) in
 3   let bqual = get-sur(dict(SCOPEUNIT)) in
 4   if bqual ∈ {endpoint1, endpoint2} then
 5     (let connectset = {get-visible-qual(chidlist[i], CHANNEL)(dict) | i ∈ ind chidlist} in
 6      let (insigset, outsigset) = if endpoint2 = bqual then
 7                                    (sigset1, sigset2)
 8                                  else
 9                                    (sigset2, sigset1) in
10      let cas1id = make-as1-identifier(cqual)(dict),
11          cas1list = {make-as1-identifier(cq)(dict) | cq ∈ connectset} in
12      let cqual' = if newc = nil then cqual else convert-channel-qual(newc, bqual) in
13      if card connectset = len chidlist ∧
14        is-wf-connectchannels(connectset, insigset, outsigset, {}, {})(dict) then
15        (cqual', connectset, mk-Channel-connection1(cas1id, cas1list))
16        else
17         exit("§3.2.2: 信道连接不是良形式的"))
18     else
19      exit("§3.2.2: 信道连接不是良形式的"))
```

type: *Connect*$_0$ → *Dict* → *Channelqual Channelqual-set Channel-connection*$_1$

目的　　把信道连接变换到 AS_1

参数　　信道连接包括

chid　　在包围作用域单位中的信道标识符

chidlist　　与 *chid* 连接的信道表

结果　　在此包围作用域单位中的信道的 *Qual*、一个 *Qual* 集合（包含连接到那个信道的那些信道）和 AS_1 信道连接。

算法

第 1 行　　构造 *chid* 的 *Qual*。
第 2 行　　分解其信道描述符。
第 3 行　　令 *bqual* 表示包围功能块的 *Qual*。
第 4 行　　该功能块必须是在连接定义中提到的信道的一个端点。
第 5 行　　构造一个集合，包含与 *chid* 连接的那些信道的 *Qual*。
第 6 行　　令 *insigset* 表示经由此信道进入子结构的信号的 *Qual* 集合，并令 *outsigset* 表示经由此信道离开子结构的信号的 *Qual* 集合。
第 10—11 行　　构造外部信道的 AS_1 标识符和它连接的信道的 AS_1 标识符集合。
第 12 行　　如果此信道有子结构，则使用适当替换信道的 *Qual*。
第 13 行　　在信道连接中一个信道不能两次被提及。并且
第 14 行　　信道中的信号和外部信道的接口必须是适当的。
第 15 行　　如果每件事都符合要求，则返回外部信道的 *Qual*、它连接的信道的 *Qual* 集合和一个 AS_1 信道连接。

is-wf-connectchannels(*connectset*, *insigset*, *outsigset*, *res1*, *res2*)(*dict*) ≙ (3.4.14.6)

```
1  (if connectset = {} then
2     is-wf-refinement(insigset, outsigset, res1, res2)(dict)
3   else
4    (let cqual ∈ connectset in
5     let mk-ChannelD(endpoint1, endpoint2, sset1, sset2, ) = dict(cqual) in
6     if ENV ∉ {endpoint1, endpoint2} then
7       false
8      else
9       (let (inset, outset) = if endpoint1 = ENV then (sset1, sset2) else (sset2, sset1) in
10       is-wf-connectchannels(connectset \ {cqual}, insigset, outsigset, res1 ∪ inset, res2 ∪ outset)(dict))))
```

type: *Channelqual*-**set** *Signalqual*-**set** *Signalqual*-**set** *Signalqual*-**set** *Signalqual*-**set** → *Dict* → *Bool*

目的　　检查在一个信道连接中各信道的接口是否适当

参数

connectset　　（余下的）信道 *Qual* 的集合（本函数是递归的）。
insigset　　经由子信道进入功能块子结构的信号 *Qual* 的集合。
outsigset　　经由子信道离开功能块子结构的信号 *Qual* 的集合。
res1，*res2*　　在子信道中规定递归地建立的输入信号集合和输出信号的集合。

结果　　如果成功了，则结果为真。

算法

第 1 行　　当全部处理完时，检查构造的信号集合（*res1*，*res2*）是否为外部信道规定信号的良形式的具体化。
第 4 行　　令 *squal* 表示在子信道集合中的一个限定表 *Qual*。
第 5 行　　分解子信道的 *Dict* 描述符。
第 6 行　　ENV 必须是子信道的一个端点。
第 9 行　　令 *inset* 表示取自子信道描述符并进入功能块子结构中的信号，*outset* 表示离开功能块子结构的信号。
第 9 行　　返回真，如果在连接中的其余子信道的接口是适当的。

is-wf-refinement(*parentset1*, *parentset2*, *subset1*, *subset2*)(*dict*) ≜ (3.4.14.7)

```
1   (if (∃q ∈ subset1 ∪ subset2)(get-sur(q) ∈ parentset1) then
2      (let q ∈ subset1 ∪ subset2 be s.t. get-sur(q) ∈ parentset1 in
3       let parent = get-sur(q) in
4       let mk-SignalD(, sub1, sub2) = dict(parent) in
5       if q ∈ subset1 ∧ sub1 ⊆ subset1 ∧ sub2 ⊆ subset2 then
6         is-wf-refinement(parentset1 \ {parent}, parentset2, subset1 \ sub1, subset2 \ sub2)(dict)
7        else
8         false)
9     else
10     (if (∃q ∈ subset2 ∪ subset1)(get-sur(q) ∈ parentset2) then
11        (let q ∈ subset1 ∪ subset2 be s.t. get-sur(q) ∈ parentset2 in
12         let parent = get-sur(q) in
13         let mk-SignalD(, sub1, sub2) = dict(parent) in
14         if q ∈ subset2 ∧ sub1 ⊆ subset2 ∧ sub2 ⊆ subset1 then
15           is-wf-refinement(parentset1, parentset2 \ {parent}, subset1 \ sub2, subset2 \ sub1)(dict)
16          else
17           false)
18       else
19        parentset1 = subset1 ∧ parentset2 = subset2))
```

type: *Signalqual*-**set** *Signalqual*-**set** *Signalqual*-**set** *Signalqual*-**set** → *Dict* → *Bool*

目的

检查在子结构边界上的信号的具体化是否适当

参数

parentset1	由外部信道传递的进入信号的 Qual 的集合
parentset2	由外部信道传递的走出信号的 Qual 的集合
subset1	由子信道传递的进入信号的 Qual 的集合
subset2	由子信道传递的走出信号的 Qual 的集合

结果

如果成功了，则结果为真。

算法

第 1 行	如果存在一个由子信道传递的信号，该信号有一个由外部信道传递的父辈进入信号，则进入信号已被细分（具体化）。
第 2 行	抽取任意子信号。
第 3 行	抽取此子信号的父辈。
第 4 行	分解信号描述符以便访问它的子信号 *sub*1 和 *sub*2。
第 5 行	既然有一个子信号在子信道中规定了，所有的子信号就都必须被规定，也就是说在两个方向（*sub*1，*sub*2）上的子信号必须是为子信道指定的信号的子集。
第 6 行	在检查是否有进一步的细分之前，先从参数中删除此父辈信号和它的子信号。
第 10—17 行	处理同 1—6 行，但这里所处理的是一个走出的父辈信号。
第 9 行	如果不再细分，则由外部信道传递的那些进入信号必须等于由子信道传递的那些进入信号，而由外部信号传递的那些走出信号必须等于由子信道传递的那些走出信号。

service-connectmap(*decllist*, *sigrouteset*, *connectmap*)(*dict*) ≜ (3.4.14.8)

```
 1  (if decllist = ⟨⟩ then
 2     (let pqual = dict(SCOPEUNIT) in
 3      let outersigrouteset = {mk-SignalrouteD(endp1, endp2, , ) ∈ rng dict | pqual ∈ {endp1, endp2}} in
 4      if sigrouteset = {} ∧ connectmap = [] then
 5        []
 6       else
 7        if card outersigrouteset > card dom connectmap
 8          then exit("§4.10.2: 在进程中丢失了信号路由连接")
 9          else if sigrouteset ≠ union rng connectmap
10                 then exit("§4.10.2: 所有服务信号路由必须在一个连接中描述")
11                 else connectmap)
12   else
13    ( cases hd decllist:
14      (mk-Sigroutedef0(nm, mk-Sigroutepath0(end1, end2, ), )
15         → (let qset = if end1 = ENV ∨ end2 = ENV
16                       then {dict(SCOPEUNIT) ⌢ ⟨(SIGNALROUTE, nm)⟩}
17                       else {} in
18            let newset = sigrouteset ∪ qset in
19            service-connectmap(tl decllist, newset, connectmap)(dict)),
20       mk-Connect0(, )
21         → (let (route, routeset) = service-connect(hd decllist)(dict) in
22            let nconnectmap = connectmap + [route ↦ routeset] in
23            if route ∉ dom connectmap ∧ routeset ∩ union rng connectmap = {} then
24              service-connectmap(tl decllist, sigrouteset, nconnectmap)(dict)
25             else
26              exit("§4.10.2: 非法的服务信号路由连接")),
27       T → service-connectmap(tl decllist, sigrouteset, connectmap)(dict))))
```

type: *Decompositiondecl*$_0^*$ *Qual*-**set** *ProcessconnectionD* → *Dict* → *ProcessconnectionD*

目的

检查一个服务分解的服务信号路由的连接是否适当

参数

decllist
一个分解的定义表

sigrouteset
是以 ENV 为一个端点的那些服务路由的限定表 *Qual*。这些服务信号路由是在递归地遍历定义表期间收集的。当（在函数 *transform-decomposition-body* 中）初始应用本函数时，此集合为空。

connectmap
是一个映射表，它规定了信号路由和它所连接的服务信号路由之间的关系。这个映射表是在递归地遍历定义表期间构造的。

结果

是上面所提到的进程连接描述符 *Process-connectionD*，它用于检查服务中的 VIA 构件

算法

第 1—9 行
当所有的连接和服务信号路由已被考虑过以后，必须满足：服务信号路由的集合必须等于服务信号路由的连接中所提及的服务信号路由（第 9 行）。

第 2 行
令 *pqual* 表示进程

第 3 行
令 *outersigrouteset* 表示以此进程作为一个端点的那些信号路由。

第 4 行　　如果没有为进程指定服务信号路由（和连接），则返回空的映射表。

第 7 行　　如果以进程作为一个端点的信号路由多于服务信号路由连接中的信号路由，则某些服务信号路由连接被遗漏了（第 7 行）。

第 13 行　　考虑下一个定义：

第 14—18 行　　如果下一个定义是一服务信号路由定义并且它的一个端点是 **ENV**，则令 *newset* 表示包括那个服务信号路由的新的集合。

第 19 行　　继续处理下一个定义。

第 20 行　　如果下一个定义是一服务信号路由连接，则

第 21 行　　检查此连接的良形式性并且导出信号路由（*route*）的 *Qual* 和它所连接的那些服务信号路由（*routeset*）的 *Qual*。

第 22 行　　更新连接映射表以包括有关此连接的信息。

第 23 行　　信号路由或服务信号路由只能在一个（不能在两个或更多）连接中提及。本连接中的路由或信号路由一定不能连接在另一个连接中。

第 24 行　　继续处理其余定义。

第 27 行　　继续处理其余定义。

service-connect(**mk-***Connect*$_0$(*sigrouteid*, *routeidlist*))(*dict*) ≙ **(3.4.14.9)**

```
1  (let squal = get-visible-qual(sigrouteid, SIGNALROUTE)(dict) in
2   let mk-SignalrouteD(, endpoint2, sigset1, sigset2) = dict(squal) in
3   let pqual = dict(SCOPEUNIT) in
4   let connectset = {get-visible-qual(routeidlist[i], SIGNALROUTE)(dict) | i ∈ ind routeidlist} in
5   let (insigset, outsigset) = if endpoint2 = pqual then (sigset1, sigset2) else (sigset2, sigset1) in
6   if card connectset ≠ len routeidlist then
7     exit("§4.10.1: 服务信号路由标识符在服务信号路由连接中出现两次")
8    else
9     if is-wf-connectsignalroutes(connectset, insigset, outsigset, {}, {})(dict) then
10      (squal, connectset)
11     else
12      exit("§4.10.1: 服务信号路由中的信号必须和用于所连接的信号路由中的信号相同"))
```

type: *Connect*$_0$ → *Dict* → *Qual Qual*-**set**

目的　　检查一个服务信号路由连接的良形式性

参数　　AS$_0$ 服务信号路由连接包含

sigrouteid　　信号路由

routeidlist　　若干服务信号路由组成的表

结果　　此信号路由的限定表 *Qual* 和这些服务信号路由的 *Qual* 的集合

算法

第 1 行　　构造信号路由的限定表 *Qual*。

第 2 行　　分解信号路由描述符。

第 3 行　　令 *pqual* 表示外围进程的限定表 *Qual*。

第 4 行　　构造这些服务信号路由的 *Qual*。

第 5 行　　令 *insigset* 表示进入进程的信号路由中的信号，令 *outsigset* 表示离开信号路由的信号。

第 6—7 行　　在此连接中的那些服务信号路由必须是不相交的。并且

第 9 行　　信号路由中的那些信号必须与服务信号路由中的那些信号相匹配。

3.5 表达式的变换

本节包含各种函数，用于把表达式和项变换为 AS_1 表达式和项。它们也注意“通过上下文求解”的问题，即，有时调用表达式变换函数是为了辨别表达式的类别。因此，这些函数也把所允许类别的集合用作为一个变元，而它们返回的允许的类别集合总是这个变元集合的非空子集。例如在等式的变换中，当变换等式左边时，最初给出的所有可见类别的集合作为变元。然后把所得到的类别集合用于变换等式右边时。该变换的最终类别集合则必须仅只包含一个类别。然后再一次变换左边与右边。这一次，把那个特殊的类别用作变元。

此外，有关表达式（或项）所在的上下文信息被作为变元给出。使用四个不同类型的上下文：**CONSTANT** 表示变元表达式（或项）必须是一常量表达式，**EXPRESSION** 表示此变元可以是一个非常量表达式，**AXIOMS** 表示此变元必须是用于公理中的一个项，而 **MAPPING** 表示此变元必须是用于公理的映射部分中的一个项。

在变换期间，收集有关进口表达式和在表达式（或项）中出现的值标识符的信息。因此也要返回一个字典 *Dict*，此 *Dict* 中可能包含值标识符描述符（*ValueidD*）和在 *Quotdict* 的项目 **IMPLIED** 和 **IMPORTLIST** 中的进口表达式的信息（参见 *Quotdict* 的域定义）。

入口函数是 *transform-term* 和 *transform-expr* 这两个（完全一样的）函数。

$$transform\text{-}term(term, context, sortset)(dict) \triangleq \tag{3.5.1}$$

1　*transform-expr*(*term*, *context*, *sortset*)(*dict*)

type：　$Term_0$ *Context Sortqual-set* → *Dict* → $Term_1$ *Sortqual-set Dict*

目的　　把 AS_0 项变换为 AS_1 项，此函数与函数 *transform-expr* 完全一样，引入它是为了提高可读性。

transform-expr(*expr*, *context*, *sortset*)(*dict*) ≜ (3.5.2)

```
1  (let (as1tree, sortset', dict') =
2      ( cases expr:
3        (mk-Scopeexpr0(scope, ex)
4           → transform-expr(ex, context, sortset)(dict + [SCOPEUNIT ↦ scope]),
5         mk-Id0(,)
6           → if context ∈ {AXIOMS, MAPPING} then
7                transform-axiom-id(expr, sortset)(dict)
8              else
9                transform-id(expr, context, sortset)(dict),
10        mk-Stringterm0(,)
11          → transform-stringexpr(expr, sortset)(dict),
12        mk-Condexpr0(e1, e2, e3),
13        mk-Condterm0(e1, e2, e3)
14          → transform-condexpr(mk-Condexpr0(e1, e2, e3), context, sortset)(dict),
15        mk-Operatorappo(,)
16          → transform-operatorexpr(expr, context, sortset)(dict),
17        mk-Operatorterm0(,)
18          → transform-operatorterm(expr, context, sortset)(dict),
19        mk-Monadexpr0(op, ex),
20        mk-Monadterm0(op, ex)
21          → transform-monadexpr(mk-Monadexpr0(op, ex), context, sortset)(dict),
22        mk-Infixexpr0(e1, op, e2),
23        mk-Infixterm0(e1, op, e2)
24          → transform-infixexpr(mk-Infixexpr0(e1, op, e2), context, sortset)(dict),
25        mk-Errorterm0()
26          → exit("§5.4.1.7: 错误项被用于一个组合项中"),
27        mk-Spellingterm0()
28          → transform-spelling(expr, context, sortset)(dict),
29        T → transform-build-in-expression(expr, context, sortset)(dict))) in
30   let as1tree' = if context ∉ {AXIOMS, MAPPING} ∧ is-Ground-term1(as1tree) then
31                    mk-Ground-expression1(as1tree)
32                  else
33                    as1tree in
34   (as1tree', sortset', dict'))
```

type: (*Expr*$_0$ | *Term*$_0$ | *Scopeexpr*$_0$) *Context Sortqual*-**set** → *Dict* →
[*Expression*$_1$ | *Term*$_1$] *Sortqual*-**set** *Dict*

目的

把一个 AS$_0$ 表达式变换到 AS$_1$ 表达式

参数

expr	AS$_0$ 表达式或项
context	使用 *expr* 时所在的（语法的）上下文（参见上面）
sortset	对表达式（或项）的合法结果类别的集合

结果

- 所得到的 AS$_1$ 表达式或项
- 所得到的表达式的（可能的）合法结果类别的集合，该集合总是 *sortset* 的一个子集。如果该集合的基数大于 1，则由于二义性，不用所得到的表达式或项（=**nil**）。（这未必是一个错误，因为 *transform-expr* 也注意“通过上下文求解”，参见上面）。
- 用项中可能的隐式值标识符的信息和用隐式进口变量更新 *dict*（参见上面）。

算法

第 3 行　如果此表达式是专门的综合 $Scopeexpr_0$（参见 AS_0 定义），则变换由 *scope* 表示的服务的上下文中所包含的表达式。

第 5—9 行　如果表达式是一个标识符，则作如下处理：如果表达式出现在公理中则把它作为一个字面值或一个值标识符进行变换，否则就把它作为一个字面值、一个同义词或一个变量来变换。

第 10 行　变换一个字符串。

第 12—13 行　变换一个条件表达式或一个条件项。

第 15 行　变换应用于表达式上下文中的一个运算符的应用。

第 17 行　变换应用于项的上下文中的一个运算符的应用。

第 19—20 行　变换一个一元表达式或项。

第 22—23 行　变换一个二元表达式或项。

第 25 行　错误项通常不出现在项（或表达式）中。在应用 *transform-term* 之前就已经排除了允许错误项的特殊情况。

第 27 行　变换拼字项。

第 29 行　变换其它种类（命令）的表达式。

第 30—33 行　如果所变换的项（或表达式）是常量，则它是一个基表达式 $Ground\text{-}expression_1$。

第 34 行　返回 AS_1 项（或表达式）、被允许类别的受限集合以及字典 *Dict*。字典 *Dict* 可能含有值标识符描述符和有关所包含的入口表达式的信息。

3.5.1 标识符

transform-axiom-id(*id*, *sortset*)(*dict*) ≜ (3.5.1.1)

```
 1  if card sortset = 1 then
 2    transform-axiom-id-of-this-sort(id, sortset)(dict)
 3   else
 4    (let mk-Id0(q, nm) = id in
 5     let litset = all-visible-literals(id, sortset)(dict) in
 6     let litsortset = {get-sur(q) | q ∈ litset} in
 7     let qual = dict(SCOPEUNIT) ⌢ ⟨(VALUE, nm)⟩ in
 8     let sorts = if q ≠ ⟨⟩ then
 9                   {}
10                 else
11                  if qual ∈ dom dict
12                    then (let mk-ValueidD(, so, ) = dict(qual) in
13                          so)
14                    else sortset in
15     let valset = sortset ∩ sorts in
16     let totset = litsortset ∪ valset in
17     if card totset = 0 then
18       exit "§2.2.2: 没有类别可用于值标识符")
19      else
20       if card totset = 1 then
21         transform-axiom-id-of-this-sort(id, totset)(dict)
22        else
23         if q = ⟨⟩ then
24           (let mk-ValueidD(val, so, e) = if qual ∈ dom dict then
25                                            dict(qual)
26                                          else
27                                            mk-ValueidD(nil, sortset, false) in
28            if so ∩ sortset = {} then
29              (nil, totset, dict)
30             else
31              (let dict' = dict + [qual ↦ mk-ValueidD(val, so ∩ sortset, e)] in
32               (nil, totset, dict')))
33          else
34           (nil, totset, dict))
```

type: Id_0 *Sortqual*-**set** → *Dict* → [$Term_1$] *Sortqual*-**set** *Dict*

目的　　把一个字面值项或值标识符项变换到 AS_1

参数　　参见变换表达式函数 *transform-expr* 和本节的引言

结果　　参见变换表达式函数 *transform-expr* 和本书的引言

算法

第 1 行　　如果已确定了此标识符的类别，则按照这种类别变换此标识符。

第 4 行　　分解此标识符。

第 5 行　　构造所有可见字面值的 *Qual* 集合，这些字面值具有相同的 AS_0 标识符并且是在该上下文中允许的类别。

第 6 行　　构造包含这些字面值类别的 *Qual* 的集合。

第 7 行　　构造一个值标识符的限定表 *Qual*。

第 8—14 行　　如果此标识符是限定的，则它不能是一值标识符（可能类别的集合是空的），否则如果在字典 *Dict* 中存在一个值标识符的描述符（由显式量化隐含或其它地方的隐式量化所隐含），则在描述符（*so*）中会发现可能类别的集合。否则此值标识符可以是在该上下文（*sortset*）所允许的任何类别。

第 15 行　　如果此标识符是一个值标识符，则令 *valset* 表示在该上下文中允许类别的集合。

第 16 行　　如果此标识符表示一个值标识符或一个字面值，则令 *totset* 表示该上下文中所允许类别的集合。

第 17 行　　至少必须存在一个可能的类别。

第 20—34 行　　如果恰好存在一个可能的类别，则按照这个类别变换此标识符，否则没有 AS_1 实物返回（**nil**），而返回所允许类别的完整集合。但是，如果此标识符可以表示一值标识符（如果它没有限定），则首先要创建或修改值标识符描述符。

第 24—27 行　　如果值标识符描述符已不存在，则建造一个描述符。

第 28—29 行　　如果描述符中的类别集合没有一个元素和该上下文中所允许的类别集合中的元素相同，则该标识符不能表示一值标识符，因此也不会有描述符加到 *Dict* 中。

第 31—32 行　　如果描述符有元素和所允许的类别集合中的元素相同，则限制描述符中的类别集合，使它仅包含那些公共的元素。返回修改过的 *Dict*。

transform-id(*id*, *context*, *sortset*)(*dict*) ≜ **(3.5.1.2)**

```
1  (let squal = get-visible-qual(id, VALUE)(dict) in
2   let sort = get-parent(s-Sortqual(dict(squal)))(dict) in
3   if is-local(squal)(dict) then
4     if (context = EXPRESSION ∨ is-SynD(dict(squal))) ∧ sort ∈ sortset then
5       if is-SynD(dict(squal)) then
6         transform-expr(s-Expr0(dict(squal)), CONSTANT, {sort})(dict)
7        else
8         (make-as1-identifier(squal)(dict), {sort}, dict)
9       else
10       exit("§2.2.2: 局部同义词或变量的类别不正确")
11    else
12     (let totalset = all-visible-literals(id, sortset)(dict) ∪
13          all-variables-and-synonyms(id, context, sortset)(dict) in
14      let squalset = {q | (∃d ∈ totalset)(q = get-parent(s-Sortqual(dict(d)))(dict))} in
15      if totalset = {} then
16        exit("§2.2.2: 同义词、变量或字面值与上下文的类别不匹配")
17       else
18        if card squalset = 1 then
19          if card totalset ≠ 1 then
20            exit("§2.2.2: 对于该上下文有几个 变量和同义词是可能的")
21           else
22            (let qual ∈ totalset in
23             let squal ∈ squalset in
24             if is-SynD(dict(qual)) then
25               (let deflevel = [SCOPEUNIT ↦ get-sur(qual)] in
26                transform-expr(s-Expr0(dict(qual)), CONSTANT, {squal})(dict + deflevel))
27              else
28               if is-LiteralD(dict(qual)) then
29                 (mk-Ground-term1(make-as1-identifier(qual)(dict)), {squal}, dict)
30                else
31                 (make-as1-identifier(qual)(dict), {squal}, dict))
32         else
33          (nil, squalset, dict)))
```

type: Id_0 *Context Sortqual*-**set** → *Dict* → [$Expression_1$] *Sortqual*-**set** *Dict*

目的　　把一个字面值、一个变量或一个同义词变换到 AS_1

参数　　参见变换表达式函数 *transform-expr* 和本节的引言

结果　　参见变换表达式函数 *transform-expr* 和本节的引言

算法

第 1 行　　首先利用通常的可见性算法去找一个变量或同义词。

第 2 行　　如果它是局部定义的，则

第 4 行　　如果上下文是常量表达式，则它必须是一同义词，并且该变量或同义词的类别必须是正确的类别。

第 5 行　　如果此标识符表示同义词，则变换此同义词的表达式，否则返回变量的 AS_1 标识符和变量类别。

第 12—26 行　　如果此同义词或变量不是局部定义的，则从上下文求得定义的地方。

第 12—14 行　　对可见字面值、变量和同义词可能的 *Qual* 的总集合用 *totalset* 表示，而它们的类别用 *squalset* 表示。

第 15—16 行　　至少必须存在一个合适的变量、同义词或字面值。

第 18—19 行　　如果恰好存在一个上下文的可能的类别，则一定也恰好存在一个可能的标识符，而且如果这样的话，则

第 22 行　　令 *qual* 表示此标识符的 *Qual*。

第 23 行　　令 *squal* 表示此类别的 *Qual*。

第 24—26 行　　如果此标识符表示一个同义词，则在定义同义词的上下文中（即 *deflevel*）变换同义词的表达式。

第 28—31 行　　返回此变量或同义词的 AS_1 的标识符，而如果该标识符表示一个字面值，则把此标识符放在一个基项 $Ground\text{-}term_1$ 中。

第 33 行　　如果有多于一个的可能类别，则表达式（仍然）不是唯一的，并返回 **nil**。

$$all\text{-}variables\text{-}and\text{-}synonyms(\mathbf{mk}\text{-}Id_0(qu, nm), context, sortset)(dict) \triangleq \qquad (3.5.1.3)$$

```
1  (let level = dict(SCOPEUNIT) in
2   {q ∈ dom dict | (is-SynD(dict(q)) ∨ (is-VarD(dict(q)) ∧ context ≠ CONSTANT)) ∧
3                   (∃q')(get-parent(q' ⌢ qu)(dict) ⌢ ⟨(VALUE, nm)⟩ = q ∧
4                        (∃q'')(q ⌢ q'' = level) ∧
5                        get-parent(s-Sortqual(dict(q)))(dict) ∈ sortset)})
```

type：　Id_0 *Context Sortqual*-**set** → *Dict* → *Qual*-**set**

目的　　抽取可被用于给定条件的所有的同义词和变量的 *Qual*

参数

Id_0　　一个标识符，它限定（限制）所得到的 *Qual* 集合

context　　如果上下文等于 **CONSTANT**，则仅抽取同义词。

sortset　　类别的集合，它也限定所得到的 *Qual* 集合

算法

第 1 行　　令 *level* 表示包围作用域单位。

第 2—5 行　　返回所有这样的 *Qual*，它们表示同义词或变量，包括限定词在内（第 3 行），是可见的（第 4 行），并且其类别也是正确的（第 5 行）。

transform-axiom-id-of-this-sort(*id*, *totset*)(*dict*) ≜ (3.5.1.4)

```
1  (let mk-Id0(qual, nm) = id in
2   let q = dict(SCOPEUNIT) ⌢ ⟨(VALUE, nm)⟩ in
3   if qual = ⟨⟩ ∧ q ∈ dom dict ∧
4     (let mk-ValueidD(, so, e) = dict(q) in
5      so = totset ∧ e) then
6     (let mk-ValueidD(val, , ) = dict(q) in
7      if val = nil then
8        (mk-Composite-term1(make-as1-identifier(q)(dict)), totset, dict)
9       else
10       transform-axiom-id-of-this-sort(as0-id(val), totset)(dict))
11    else
12     (let litset = all-visible-literals(id, totset)(dict) in
13      if litset = {} then
14        if q ≠ ⟨⟩ then
15          exit("§5.2.3: 值标识符一定不能限定")
16         else
17          (let vald = if q ∈ dom dict then
18                        (let mk-ValueidD(, so, ) = dict(q) in
19                         if so ∩ totset = {} then
20                           exit("§5.2.3: 不一致地使用隐含量化的值名字")
21                         else
22                           mk-ValueidD(nil, so ∩ totset, false))
23                      else
24                       mk-ValueidD(nil, totset, false) in
25           let d = dict + [q ↦ vald] in
26           (mk-Composite-term1(make-as1-identifier(q)(dict)), totset, d))
27       else
28        (let litqual ∈ litset in
29         (mk-Ground-term1(make-as1-identifier(litqual)(dict)), totset, dict))))
```

type: Id_0 *Sortqual*-**set** → *Dict* → $Term_1$ *Sortqual*-**set** *Dict*

目的

把一规定类别的标识符变换到 AS_1

参数

id	要被变换的 AS_0 标识符
totset	仅包含规定类别的类别的 Qual 集合

结果

参见变换表达式函数 *transform-expr* 和本节的引言

算法

第 1 行	分解该标识符。
第 2 行	构造此标识符的 *Qual*，假定它是一个值标识符。
第 3 行	如果它没有被限定，则它表示一个显式量化的值标识符，且它有一个在 *Dict* 中的值标识符描述符 *valueidD*（第 3 行），且规定的类别等于值标识符的类别（第 5 行），而且表示这个标识符是被显式限定的标志（*e*）为真。
第 6—8 行	如果值标识符不是通过字面值量化（*val*=**nil**）引入的，则返回包含 AS_1 值标识符的组合项，否则
第 10 行	变换相关字面值的标识符。
第 12 行	如果此标识符不表示一个被显式量化的值标识符，则抽取所有可能的字面值的 Qual。
第 13—14 行	如果此标识符不表示一个字面值，则它表示一隐式量化的值标识符，因而它必须是

未被限定的。

第 17—24 行　抽取或构造此值标识符的描述符，并且用 *vald* 表示它。

第 18 行　如果其描述符在 *Dict* 中已经有了（由于别处用过了），则把它分解，并且

第 19 行　对这里的使用和别处的使用至少必须存在一个合法的和公共的类别，而描述符中的合法类别的集合则受 *totset* 的限制（第 22 行）。

第 24 行　否则构造一新描述符。

第 26 行　返回 AS_1 组合项，它包含值标识符、（那一个）合法类别和更新了的 *Dict*。

第 28—29 行　如果存在此类别的一个字面值的 *Qual*，则把它变换到 AS_1。

3.5.2 字符串

transform-stringexpr(mk-*Stringterm*$_0$(*q*, *strnm*), *sortset*)(*dict*) ≙ **(3.5.2.1)**

```
1  (let lqualset = all-visible-literals(mk-Stringterm0(q, strnm), sortset)(dict) in
2   let litsortset = {get-sur(lit) | lit ∈ lqualset} in
3   if lqualset = {} then
4     exit("§2.2.2: 字符串的类别不适合")
5    else
6     if card lqualset = 1 then
7       (let lqual ∈ lqualset in
8        let as1lit = make-as1-identifier(lqual)(dict) in
9        (mk-Ground-term1(as1lit), sortset, dict))
10      else
11       (nil, litsortset, dict))
```

type: *Stringterm*$_0$ *Sortqual-set* → *Dict* → [*Ground-term*$_1$] *Sortqual-set* *Dict*

目的　把一个字符串变换到 AS_1

参数　参见变换表达式函数 *transform-expr* 和本节的引言

结果　参见变换表达式函数 *transform-expr* 和本节的引言

算法

第 1 行　抽取所有字面值的 *Qual*，这些字面值既是 *sortset* 中的一个类别，又可以用 *stringterm*$_0$ 来表示。

第 2 行　抽取它们的类别的 *Qual*。

第 3—4 行　至少存在一种可能和此类别匹配的字面值。

第 6—9 行　如果恰好存在一个可能的字面值（和结果的类别），则抽取该字面值的 *Qual*（第 7 行），构造 AS_1 字面值（第 8 行）并返回包含该字面值的基项、包含它的类别的类别集合和未改变的 *Dict*。

第 11 行　如果（仍然）存在不止一个可能的字面值（和类别），则不返回 AS_1 项，而返回可能的类别集合和未改变的 *Dict*。

3.5.3 运算符

transform-operatorexpr(mk-*Operatorapp*$_0$(*expr*, *exprlist*), *context*, *tpset*)(*dict*) ≙ **(3.5.3.1)**

```
1  (let (qual, nm) = cases expr:
2                    (mk-Id0(q, n)       → (q, n),
3                     mk-Qualop0(q, n) → (q, n),
```

```
 4                    T                  → (⟨⟩,nil)) in
 5   let level = dict(SCOPEUNIT) in
 6   let opqualset = if nm = nil then {} else all-visible-operators(qual, nm, level)(dict) in
 7   let legalqualset = extract-legal-operators(exprlist, context, opqualset, tpset)(dict) in
 8   let op = build-extract-operator(expr, exprlist) in
 9   let (as1 tree', tpset', dict') =
10       if op = nil then
11         (nil, {}, [])
12        else
13         (trap exit with (nil, {}, []) in
14          transform-operatorexpr(op, context, tpset)(dict)) in
15   let fop = build-field-operator(expr, exprlist) in
16   let (as1 tree'', tpset'', dict'') =
17       if fop = nil then
18         (nil, {}, [])
19        else
20         (trap exit with (nil, {}, []) in
21          transform-selectexpr(fop, context, tpset)(dict)) in
22   (card (legalqualset ∪ tpset' ∪ tpset'') = 0
23       → exit("§2.2.2: 该运算符项不能用于这个上下文中"),
24    card legalqualset = 0 ∧ card tpset' = 0
25       → (as1 tree'', tpset'', dict''),
26    card legalqualset = 0 ∧ card tpset'' = 0
27       → (as1 tree', tpset', dict'),
28    card (tpset' ∪ tpset'') = 0 ∧ card legalqualset = 1
29       → (let qu ∈ legalqualset in
30          transform-qual-operator(qu, exprlist, context, tpset)(dict)),
31    T → (nil, {get-parent(s-Result(dict(q)))(dict) |
32               q ∈ legalqualset} ∪ tpset' ∪ tpset'', dict)))
```

type: *Operatorapp*$_0$ *Context Sortqual-set* → *Dict* → *Expression*$_1$ *Sortqual-set Dict*

目的

把一个 AS_0 表达式变换到 AS_1，此 AS_0 表达式可以是一种运算符应用、一个抽取的速记符或一个字段选择速记符

参数

参见变换表达式函数 *transform-expr* 以及本节的引言

结果

参见变换表达式函数 *transform-expr* 以及本节的引言

算法

第 1—4 行　抽取此运算符的限定符和名字。如果运算符是一般表达式（第 4 行），则没有名字，因为它是在那种情况下的抽取运算符的一个速记符。

第 5 行　令 *level* 表示包围的作用域单位。

第 6 行　根据限定符和名字抽取所有可能的运算符的 *Qual*。

第 7 行　限制 *Qual* 集合以便在考虑变元表的时候使集合仅包含可以被使用的那些运算符。

第 8—14 行　试探抽取运算符的速记符。*op* 包含抽取应用的可能的结果类别。

第 15—21 行　试探字段选择速记符。*fop* 包含字段选择的可能的结果类别。

第 22 行　如果既没有通常运算符的应用，也没有抽取应用，也没有字段选择，可以用于该上下文，那么这是一种错误。

第 24 行　如果只有该字段选择可以用于该上下文中，则选择这个速记符作为结果。

第 26 行　如果只有抽取的应用可用于此上下文，则选择这个速记符作为结果。

第 28 行　　如果只有通常的运算符应用可用于该上下文中，则抽取该运算符的限定表 *Qual*，并给定此 *Qual* 变换该运算符应用。

第 31 行　　否则此运算符（仍然）是二义的，因而不返回 AS_1 表达式，而返回由该上下文中所有可能的运算符的结果类别所组成的类别集合，与抽取速记符的可能类别集合（*tpset'*）以及字段选择速记符的类别集合（*tpset''*）的并集。

all-visible-operators(*qual*, *nm*, *level*)(*dict*) ≙ **(3.5.3.2)**

```
1  {q ∈ dom dict | (∃q')(get-sur(get-sur(q)) ⌢ q' = level) ∧
2                  (∃q'')(get-parent(q'' ⌢ qual)(dict) ⌢ ⟨(OPERATOR, (nm, , ))⟩ = q)}
```

type: *Qual* (*Name*$_0$ | *Quotedop*$_0$) *Qual* → *Dict* → *Operatorqual-set*

目的　　抽取在作用域单位 *level* 中可见的、包括限定词 *qual* 并具有名字 *nm* 的所有运算符的 *Qual*

算法　　结果的集合，包括可见的（即，含有定义运算符的部分类型定义的作用域单位必须是 *level* 的一部分）所有那些 *Operatorqual* 和在 *Operatorqual* 中的限定词。

transform-operatorterm(**mk**-*Operatorterm*$_0$(*id*, *exprlist*), *context*, *tpset*)(*dict*) ≙ **(3.5.3.3)**

```
1  (let (qual, nm) = cases id:
2                      (mk-Id0(q, n)       → (q, n),
3                       mk-Qualop0(q, n) → (q, n)) in
4   let level = get-sur(dict(SCOPEUNIT)) in
5   let opqualset = all-visible-operators(qual, nm, get-sur(level))(dict) in
6   let legalqualset = extract-legal-operators(exprlist, context, opqualset, tpset)(dict) in
7   if card legalqualset = 0 then
8     exit("§2.2.2: 该运算符项不能用于这个上下文中")
9    else
10    if card legalqualset = 1 then
11      (let qu ∈ legalqualset in
12       transform-qual-operator(qu, exprlist, context, tpset)(dict))
13     else
14      (nil, {get-parent(s-Result(dict(q)))(dict) | q ∈ legalqualset}, dict))
```

type: *Operatorterm*$_0$ *Context Sortqual-set* → *Dict* → *Term*$_1$ *Sortqual-set Dict*

目的　　把一运算符项变换到 AS_1

参数　　参见变换表达式函数 *transform-expr* 及本节的引言

结果　　参见变换表达式函数 *transform-expr* 及本节的引言

算法

第 1—3 行　　抽取运算符的限定符和名字。

第 4 行　　此运算符必须可见的层次，就是包围使用该运算符的类别定义的作用域单位。

第 5 行　　抽取在外围作用域单位中可见的和用 *qual* 作 AS_0 限定词的并且有名字 *nm* 的所有那些运算符的 *Qual*。

第 6 行　　限制 *Qual* 的集合以便仅包含那些当把变元表考虑在内时可以使用的运算符。

第 7 行　　至少必须存在一个和上下文匹配的可能运算符。

第 10—12 行　　如果仅有一个运算符项与上下文匹配，则根据正确的 *Qual*（*qu*）把它变换到 AS_1。

第 14 行　　如果运算符项（仍然）是二义的，则不返回 AS_1 项，而返回这些可能运算符的结果类别。

build-extract-operator(*expr*, *exprlist*) ≙ (3.5.3.4)

```
1  if is-Qualop0(expr) then
2    nil
3   else
4    (let nm = mk-Name0("EXTRACT", EXCLAMATION) in
5     mk-Operatorapp0(mk-Id0(⟨⟩, nm), ⟨⟨expr⟩ ⌢ exprlist⟩))
```

type: $Expr_0\ Exprlist_0 \rightarrow [Operatorapp_0]$

目的　　从一个抽取运算符的速记表示来构造此抽取运算符的应用

例如

`a(b, c)`

（`a` 是形参 *expr*，表 `b`、`c` 是形参表 *exprlist*）被变换为

`EXTRACT ! (a, b, c)`

如果运算符标识符是一个中缀运算符，则返回 **nil**，因为在那种情况下该表示法总是表示一种运算符应用。

build-field-operator(*var*, *exprlist*) ≙ (3.5.3.5)

```
1  (let expr = exprlist[len exprlist] in
2   let rest = ⟨exprlist[i] | 1 ≤ i ≤ len exprlist − 1⟩ in
3   if ¬is-Id0(expr) ∧ s-Qualifier0(expr) = ⟨⟩ then
4     nil
5    else
6     (let exprvar = if len rest = 0 then var else build-field-operator(var, rest) in
7      if exprvar = nil then
8        nil
9       else
10       (let mk-Id0(, nm) = expr in
11        mk-Selectexpr0(exprvar, nm))))
```

type: $Expr_0\ Exprlist_0 \rightarrow [Selectexpr_0]$

目的　　从带有括号的速记表示构造一个字段抽取速记符。

例如

`a(b, c)`

（`a` 是形参 *var*，表 `b`、`c` 是形参表 *exprlist*）被变换为

`a! b! c`

此函数是递归的。也就是说，首先构造 `a! b`，然后构造 `a! b! c`。

***extract-legal-operators*(*exprlist*, *context*, *opqualset*, *sortset*)(*dict*)** $\triangleq$ **(3.5.3.6)**

```
1  (if opqualset = {} then
2     {}
3   else
4    (let q ∈ opqualset in
5     let (,ts,) =
6         (trap exit with (nil, {}, []) in
7          transform-qual-operator(q, exprlist, context, sortset)(dict)) in
8     let qs = if ts = {} then {} else {q} in
9     qs ∪ extract-legal-operators(exprlist, context, opqualset \ {q}, sortset)(dict)))
```

type: *Exprlist*$_0$ *Context Operatorqual-set Sortqual-set* → *Dict* → *Operatorqual-set*

目的 抽取与实参类别和结果类别匹配的并具有规定名字的所有运算符的 *Qual*

参数

exprlist	实在参数表
context	使用运算符的上下文（参见 *transform-expr*）
opqualset	所有具有规定名字的可能的 *Qual*
sortset	可能的结果类别

结果 可被使用的运算符的 *Qual*，这个集合是 *opqualset* 的一个子集

算法

第 1 行	当全部处理完时，返回空集（此函数是递归的）。
第 4 行	取出一运算符的 *Qual* 并用下述方法测试它：
第 5—7 行	给定此运算符的 *Qual*，试变换此运算符的应用。如果该变换成功（即如果函数 *transform-qual-operator* 没有被捕捉），则结果类别的集合（*ts*）是非空的。
第 8 行	如果此运算符的 *Qual* 是可用的，则用 *qs* 表示该 *Qual*。
第 9 行	把 *qs* 与 *opqualset* 其余部分中所允许的那些运算符的 *Qual* 汇合起来。

transform-qual-operator(*qual*, *exprlist*, *context*, *sortset*)(*dict*) ≙ **(3.5.3.7)**

```
1  (let (, (,parm, tqual)) = qual[len qual] in
2   if tqual ∉ sortset then
3     exit("§2.2.2: 运算符的结果类别对于该上下文是非法的")
4    else
5     if len parm ≠ len exprlist then
6       exit("§5.2: 对一个运算符来说参数表的长度不正确")
7      else
8       (let (as1 parmlist, d) = transform-actparms(parm, exprlist, context)(dict) in
9        let as1 id = make-as1-identifier(qual)(dict) in
10       let const = (∀p ∈ elems as1 parmlist)(is-Ground-term1(p)) in
11       let as1 tree =
12           if context ∈ {AXIOMS, MAPPING} then
13             if const then
14               mk-Ground-term1((as1 id, as1 parmlist))
15              else
16               mk-Composite-term1((as1 id, as1 parmlist))
17            else
18             if const then
19               mk-Ground-term1((as1 id, as1 parmlist))
20              else
21               mk-Operator-application1(as1 id, as1 parmlist) in
22       (as1 tree, {tqual}, d)))
```

type: *Operatorqual* (*Exprlist*$_0$ | *Term*$_0$*) *Context Sortqual*-**set** → *Dict* →
(*Term*$_1$ | *Expression*$_1$) *Sortqual*-**set** *Dict*

目的　　把一特定定义的运算符变换到 AS$_1$

参数

qual	唯一地标识此运算符的限定表 *Qual*
exprlist	变元表，如果此运算符出现在等式中，则此表为多个项组成的表。否则此表由多个表达式组成。
context, *sortset*	参见变换表达式函数 *transform-expr*

结果　　参见变换表达式函数 *transform-expr*，结果的类别集合仅包含一种类别。

算法

第 1 行	从运算符的限定表 *Qual* 抽取运算符限定元素 *Operatorqualelem*，并且通过 *tqual* 标识结果的类别，通过 *parm* 标识变元的类别。
第 2—6 行	结果的类别必须为该上下文所允许的类别之一，形式参数表的长度必须等于实参表的长度。
第 8 行	变换那些实在参数（它们可以是项或是表达式）。
第 9 行	构造运算符的 AS$_1$ 标识符。
第 10 行	如果所有的实参都是基项（常量），则 *const* 为真。
第 11—18 行	如果运算符出现在公理中，则构造一个 AS$_1$ 项（第 13 行）。否则构造一个 AS$_1$ 表达式。
第 12 行	返回所构造的 AS$_1$ 项（或表达式）、运算符的结果类别和字典 *Dict*，该 *Dict* 在实参的求值期间可能被更新（参见变换表达式函数 *transform-expr*）。

3.5.3.1 拼字运算符

transform-spelling(mk-*Spellingterm*$_0$(mk-*Id*$_0$(*q*, *nm*)), *context*, *sortset*)(*dict*) ≜ **(3.5.3.1.1)**

```
1  if context ≠ MAPPING ∨ q ≠ ⟨⟩ then
2    exit("§5.4.1.15: 非法使用SPELLING(拼字) 运算符")
3   else
4    (let stringsort = get-predef-sort("CHARSTRING")(dict) in
5     let qualset = {q ∈ dom dict | is-ValueidD(dict(q)) ∧ s-Mapvalue(dict(q)) ≠ nil ∧
6                                   (VALUE, nm) = q[len q]} in
7     if card qualset = 1 ∧ stringsort ∈ sortset then
8       (let qual ∈ qualset in
9        let mk-ValueidD(litq, , ) = dict(qual) in
10       let (, mk-Name0(str, )) = litq[len litq] in
11       transform-stringexpr(mk-Stringterm0(⟨⟩, str), {stringsort})(dict))
12      else
13       exit("§5.4.1.15: 非法使用SPELLING(拼字) 运算符"))
```

type: *Spellingterm*$_0$ *Context Sortqual*-**set** → *Dict* → *Term*$_1$ *Sortqual*-**set** *Dict*

目的

把拼字运算符 SPELLING 变换为它所表示的 AS_1 字符串字面值（在变换映射公理函数 *transform-mappingaxioms* 中，为限定的值名字的每个可能的字面值都产生一个公理）

参数

参见变换表达式函数 *transform-expr* 及本节的引言

结果

参见变换表达式函数 *transform-expr* 及本节的引言

算法

第 1 行　拼字运算符必须仅仅用于字面值的量化中，并且所含的值标识符不能被量化。

第 4 行　抽取字符串类别的 *Qual*。

第 5—7 行　所含的值名字（*nm*）必须只有一个 *Qual*（即一个唯一的描述符），而预定义的字符串类别在该上下文中必须是允许的。

第 8—9 行　抽取值名字的 *Qual*，并分解它的描述符。*Litq* 表示由值名字表示的字面值的 *Qual*。

第 10 行　抽取该字面值的拼字。

第 11 行　从此字面值的拼字构造一 AS_0 字符串项，并变换该字符串项。

3.5.3.2 出口、进口、视见、活跃

transform-build-in-expression(*expr*, *context*, *sortset*)(*dict*) ≙ (3.5.3.2.1)

```
 1  if context ≠ EXPRESSION ∧ ¬is-Selectexpr0(expr) ∧ ¬is-Tupleexpr0(expr) then
 2    exit("§5.5.4: 命令运算符不能被用于常数表达式中")
 3  else
 4    cases expr:
 5     (mk-Importexpr0(v, exp)
 6        → transform-importexpr(v, exp, sortset)(dict),
 7      mk-Viewexpr0(v, exp)
 8        → transform-viewexpr(v, exp, sortset)(dict),
 9      mk-Nowexpr0()
10        → transform-nowexpr(sortset)(dict),
11      mk-Activeexpr0(,)
12        → transform-activeexpr(expr, sortset)(dict),
13      mk-Selectexpr0(,)
14        → transform-selectexpr(expr, context, sortset)(dict),
15      mk-Tupleexpr0(,)
16        → transform-tupleexpr(expr, context, sortset)(dict),
17      T → transform-pid-build-in(expr, sortset)(dict))
```

type: $Expr_0$ *Context Sortqual*-**set** → *Dict* → [$Expression_1$] *Sortqual*-**set** *Dict*

目的 把一种特定的表达式变换到 AS_1

参数 参见变换表达式函数 *transform-expr* 及本节的引言

结果 参见变换表达式函数 *transform-expr* 及本节的引言

算法

第 1—2 行 只有上面那些表达式的 $Selectexpr_0$ 或 $Tupleexpr_0$ 两种情况才允许出现在常数表达式中。

第 5 行 变换一进口表达式 IMPORT。

第 7 行 变换一视见表达式 VIEW。

第 9 行 变换一现在表达式 NOW。

第 11 行 变换一定时器活跃表达式 ACTIVE。

第 13 行 变换一字段选择表达式。

第 15 行 变换一元组表达式。

第 17 行 否则，它是一预定义的 PID 表达式。

transform-importexpr(**mk-***Id*$_0$(*qu*, *nm*), *expr*, *sortset*)(*dict*) ≙ (3.5.3.2.2)

```
 1  (let level = process-or-service-level(dict(SCOPEUNIT)) in
 2   let qualset = {q ∈ dom dict | (∃q', sort)(q' ⌢ qu ⌢ ⟨(IMPORT, (nm, sort))⟩ = q ∧
 3                                        (get-sur(q) = level ∨ get-sur(q) = process-level(level)) ∧
 4                                        sort ∈ sortset)} in
 5   if qualset = {} then
 6     exit("§2.2.2: 没有进口变量可与该上下文匹配")
 7    else
 8     if card qualset = 1 then
 9       (let qual ∈ qualset in
10        let sort = s-Sortqual(dict(qual)) in
11        let importlist = dict(IMPORTLIST) in
12        let imp = dict(IMPLIED) in
13        let nm' = create-unique-name() in
14        let newqual = level ⌢ ⟨(VALUE, nm')⟩,
15            imp' = imp ∪ {(sort, nm')},
16            as1 id = make-as1-identifier(newqual)(dict) in
17        let importlist' = importlist ⌢ ⟨(nm', qual, expr)⟩ in
18        (as1 id, {get-parent(sort)(dict)}, dict + [IMPORTLIST ↦ importlist',
19                                                   IMPLIED    ↦ imp'] ))
20      else
21       (let sortset' = {sort | (∃q ∈ qualset)(dict(q) = mk-ImportD(sort))} in
22        (nil, sortset', dict)))
```

type: *Id*$_0$ [*Expr*$_0$] *Sortqual*-**set** → *Dict* → [*Expression*$_1$] *Sortqual*-**set** *Dict*

目的

把一进口表达式变换为它的 AS$_1$ 表示法

参数

Id	进口变量的标识符
expr	表示出口进程的任选 PID 表达式
sortset	参见函数 *transform-expr*

结果

参见函数 *transform-expr* 及本节的引言，所返回的表达式是由进口表达式所隐含的隐式变量标识符

算法

第 1 行	抽取定义进口变量的层次，即一个进程或一个服务。
第 2—4 行	抽取所有进口变量的进口 *Qual*，这些变量可以借助一限定词（*qu*）、一个名字（*nm*）（第 2 行）和一个来自类别集合 *sortset* 中的一个类别 *sort* 来规定，并且这些变量是在包围的服务或包围的进程中定义的。
第 5 行	至少必须存在一个可能的进口变量。
第 9 行	如果存在一个唯一的进口 *Qual*，则
第 9 行	用 *qual* 表示那个进口变量的 *Qual*。
第 10 行	用 *sort* 表示它的类别的 *Qual*。
第 11 行	用 *importlist* 表示由一些进口表达式组成的当前表。这些进口表达式是在外围表达式中某些地方用到的。
第 12 行	用 *imp* 表示用户进程体中的隐式变量的当前集合。
第 13—14 行	创建一个唯一的 AS$_0$ 变量名并构造其 *Qual*。
第 15 行	用隐式变量的类别（*sort*）与其新名字组成的偶对来更新它的当前集合。
第 16 行	构造隐式变量的 AS$_1$ 标识符。

第 17 行　　更新当前的进口表达式的表。

第 18 行　　返回 AS_1 标识符和字典 *Dict*，此 *Dict* 包含已更新的进口表和隐式变量集合。

第 21 行　　如果进口表达式（仍然）是二义的，则返回可能的类别集合。

transform-viewexpr(*id*, *expr*, *sortset*)(*dict*) ≙ (3.5.3.2.3)

```
1  (let mk-Id0(qu, nm) = id in
2   let pqual = process-or-service-level(dict(SCOPEUNIT)) in
3   let qset = {q ∈ dom dict | (let qual = pqual ⌢ ⟨(VIEW, q)⟩ in
4                               qual ∈ dom dict ∧
5                               get-parent(s-Qual(dict(qual)))(dict) ∈ sortset ∧
6                               (∃q')(q' ⌢ qu ⌢ ⟨(VALUE, nm)⟩ = q))} in
7   if card qset ≠ 1 then
8     exit("§2.2.2: 必须有且仅有一个被透露的变量与视见表达式匹配")
9     else
10    (let q ∈ qset in
11     let inst = get-predef-sort("PID")(dict) in
12     let as1id = make-as1-identifier(q)(dict),
13         (as1expr, , d) = transform-expr(expr, EXPRESSION, {inst})(dict) in
14     let as1tree = mk-View-expression1(as1id, as1expr) in
15     (as1tree, {s-Sortqual(dict(q))}, d)))
```

type: *Varid*$_0$ *Expr*$_0$ *Sortqual-set* → *Dict* → *View-expression*$_1$ *Sortqual-set* *Dict*

目的　　把一视见表达式变换到 AS_1

参数

id　　视见变量

expr　　表示透露进程的 Pid 表达式

sortset　　参见函数 *transform-expr*

结果　　参见函数 *transform-expr*

算法

第 1 行　　分解视见变量标识符。

第 2 行　　抽取定义视见变量的层次。

第 3—6 行　　令 *qset* 表示与视见表达式匹配的集合。该集合由在进程中被视见的（第 3—4 行），且是合适类别的（第 5 行）变量的限定表 *Qual* 组成，而且在这些限定表 *Qual* 中还包括在视见表达式中规定的限定词和名字（第 6 行）。

第 11 行　　抽取 PID 类别的 *Qual*。

第 12 行　　构造透露变量的 AS_1 标识符（它与视见变量的标识符相同）。

第 13 行　　变换 PID 表达式。

第 14 行　　构造 AS_1 视见表达式。

第 15 行　　返回 AS_1 视见表达式、变量的类别和（可能）更新过的 *Dict*。

transform-nowexpr(*sortset*)(*dict*) ≙ (3.5.3.2.4)

```
1  (let tqual = get-predef-sort("TIME")(dict) in
2   if tqual ∈ sortset then
3     (mk-Now-expression1(), {tqual}, dict)
4    else
5     exit("§5.5.4.1: NOW表达式必须被用于TIME值被允许的地方"))
```

type: *Sortqual-set* → *Dict* → *Now-expression*$_1$ *Sortqual-set* *Dict*

目的 把 NOW 表达式变换到 AS_1

参数 参见函数 *transform-expr*

结果 参见函数 *transform-expr*

算法

第 1 行 抽取表示 TIME 类别的限定表 *Qual*。

第 2 行 如果该类别允许在此地方出现，

第 3 行 则返回 AS_1 的现在表达式、TIME 类别和未改变的 *Dict*。

transform-activeexpr(mk-*Activeexpr*$_0$(*id*, *actparm*), *sortset*)(*dict*) ≙ (3.5.3.2.5)

```
1  (let qual = get-visible-qual(id, SIGNAL)(dict),
2       bqual = get-predef-sort("BOOLEAN")(dict) in
3   let as1 id = make-as1-identifier(qual)(dict) in
4   if ¬is-TimerD(dict(qual)) then
5     exit("§2.8: 在定时器活跃表达式中的标识符不表示一个定时器")
6    else
7     (let mk-TimerD(tplist, ) = dict(qual) in
8      if bqual ∈ sortset ∧ len actparm = len tplist then
9        (let (as1 actparm, d) = transform-actparms(tplist, actparm, EXPRESSION)(dict) in
10        let as1 tree = mk-Timer-active-expression1(as1 id, as1 actparm) in
11        (as1 tree, {bqual}, d))
12       else
13        exit("§2.8: 非法的定时器活跃表达式") )))
```

type: *Activeexpr*$_0$ *Sortqual*-**set** → *Dict* → *Timer-active-expression*$_1$ *Sortqual*-**set** *Dict*

目的 把一个定时器活跃表达式变换到 AS_1

参数

id 定时器的标识符

actparm 实在参数

sortset 参见函数 *transform-expr*

结果 参见函数 *transform-expr*

算法

第 1—2 行 抽取定时器的限定表 *Qual* 和布尔类别的限定表 *Qual*。

第 3 行 构造此定时器的 AS_1 标识符。

第 4 行 该 AS_0 标识符必须表示一个定时器（不是一个信号）。

第 7 行 令 *tplist* 表示参数类别的表。

第 8 行 布尔类别必须是在该上下文中允许的结果类别，且实在参数表的长度必须等于类别表的长度。

第 9 行 把这些实在参数变换到 AS_1。

第 10—11 行 构造 AS_1 定时器活跃表达式，并把它和布尔类别一起返回，还返回（可能）更新过的 *Dict*。

transform-selectexpr(mk-*Selectexpr*$_0$(*v*, *nm*), *context*, *sortset*)(*dict*) ≙ (3.5.3.2.6)

```
1  (let mk-Name0(str, ) = nm in
2   let opnm = mk-Name0(str ⌢ "EXTRACT", EXCLAMATION) in
3   let opid = mk-Id0(⟨⟩, opnm) in
4   transform-expr(mk-Operatorapp0(opid, ⟨v⟩), context, sortset)(dict))
```

type: *Selectexpr*$_0$ *Context Sortqual*-**set** → *Dict* → *Expression*$_1$ *Sortqual*-**set** *Dict*

目的 把字段选择表达式变换到 AS_1

参数 参见函数 *transform-expr*

结果 参见函数 *transform-expr*

算法

第 1 行 令 *str* 表示此字段的拼字。

第 2 行 构造 EXTRACT！运算符的名字，该运算符在指定的字段上操作。

第 3—4 行 构造运算符的标识符；把左边的表达式作为参数，构造一个运算符应用，并用通常的方法变换它。

transform-tupleexpr(**mk-***Tupleexpr*$_0$(*q*, *exprlist*), *context*, *sortset*)(*dict*) ≙ **(3.5.3.2.7)**

```
1  (let opid = mk-Id0(q, mk-Name0("MAKE", EXCLAMATION)) in
2   transform-expr(mk-Operatorapp0(opid, exprlist), context, sortset)(dict))
```

type: *Tupleexpr*$_0$ *Context Sortqual-set* → *Dict* → *Expression*$_1$ *Sortqual-set Dict*

目的 把一个元组表达式（结构的基础）变换到 AS_1

参数 参见函数 *transform-expr*

结果 参见函数 *transform-expr*

算法

第 1 行 构造 MAKE！运算符的标识符。

第 2 行 给定表达式表作为参数表，构造一个运算符应用，并用通常的方法变换它。

transform-pid-build-in(*expr*, *sortset*)(*dict*) ≙ **(3.5.3.2.8)**

```
1  (let pqual = get-predef-sort("PID")(dict) in
2   if pqual ∈ sortset then
3     ( cases expr:
4       (mk-Parentexpr0()
5          → (mk-Parent-expression1(), {pqual}, dict),
6        mk-Offspringexpr0()
7          → (mk-Offspring-expression1(), {pqual}, dict),
8        mk-Senderexpr0()
9          → (mk-Sender-expression1(), {pqual}, dict),
10       mk-Selfexpr0()
11         → (mk-Self-expression1(), {pqual}, dict)))
12    else
13     exit("§5.5.4.3: PID 表达式用于错误的上下文中"))
```

type: *Expr*$_0$ *Sortqual-set* → *Dict* → *Pid-expression*$_1$ *Sortqual-set Dict*

目的 把预定义 PID 表达式之一变换到 AS_1

参数 参见函数 *transform-expr*

结果 参见函数 *transform-expr*

算法

第 1 行 抽取该 PID 类别的 *Qual*。

第 2 行 该 PID 类别必须是在这个地方允许的一种类别。

第 4—10 行 返回 PID 类别的 *Qual*（*pqual*）、未改变的 *Dict* 和

第 4 行 一个父辈表达式或

第 6 行　　一个后代表达式或
第 8 行　　发送者表达式或
第 10 行　　自身表达式。

3.5.4　条件表达式

transform-condexpr(**mk-***Condexpr*$_0$(*bexpr*, *expr1*, *expr2*), *context*, *sortset*)(*dict*) ≙　　**(3.5.4.1)**

```
 1  (let boolqual = get-predef-sort("BOOLEAN")(dict) in
 2   let (as1, , d) = transform-expr(bexpr, context, {boolqual})(dict) in
 3   let (, s', d') = transform-expr(expr1, context, sortset)(d) in
 4   let (, s'', ) = transform-expr(expr2, context, sortset)(d') in
 5   if s' ∩ s'' = {} then
 6     exit("§2.2.2: 条件表达式与上下文不匹配")
 7    else
 8     if card (s' ∩ s'') = 1 then
 9       (let sort ∈ s' ∩ s'' in
10        let (as1 unique, , dict') = transform-expr(expr1, context, {sort})(dict) in
11        let (as1 unique', , dict'') = transform-expr(expr2, context, {sort})(dict') in
12        let as1 tree =
13            if context ∈ {AXIOMS, MAPPING} then
14              mk-Conditional-term1(as1, as1 unique, as1 unique')
15             else
16              mk-Conditional-expression1(as1, as1 unique, as1 unique') in
17        let as1 tree' =
18            if is-Ground-term1(as1) ∧ is-Ground-term1(as1 unique) ∧ is-Ground-term1(as1 unique')
19              then mk-Ground-term1(as1 tree)
20              else as1 tree in
21        (as1 tree', {sort}, dict''))
22      else
23       (nil, s' ∩ s'', dict))
```

type:　*Condexpr*$_0$ *Context Sortqual*-**set** → *Dict* → [*Term*$_1$ | *Expression*$_1$] *Sortqual*-**set** *Dict*

目的　　把一条件表达式或一条件项变换到 AS$_1$

参数　　参见函数 *transform-expr*

结果　　参见函数 *transform-expr*

算法

第 1 行　　抽取布尔类别的限定表 *Qual*。
第 2 行　　变换布尔条件。
第 3 行　　变换“then”表达式（或项）。
第 4 行　　变换“else”表达式（或项）。
第 5—6 行　　至少必须存在一个可以使用的结果类别。
第 8 行　　如果恰好有一个可以使用的类别，则
第 9 行　　令 *sort* 表示此类别。
第 10—11 行　　再一次变换这些表达式，这一次用此唯一的类别作为变元。
第 12—16 行　　构造 AS$_1$ 表达式或项。
第 17—20 行　　如果该条件、“then”部分和“else”部分都是基项，则结果的 AS$_1$ 表达式是一基项。
第 20 行　　返回 AS$_1$ 表达式（或项）、条件表达式（或项）的类别和（可能）更新过的 *Dict*。

3.5.5 中缀和前缀运算符

transform-monadexpr(**mk-***Monadexpr*$_0$(*op*, *expr*), *context*, *sortset*)(*dict*) $\triangleq$ (3.5.5.1)

```
1  (let qinfix = mk-Qualop0(⟨⟩, mk-Quotedop0(op)) in
2   let operator =
3       if context ∈ {MAPPING, AXIOMS}
4         then mk-Operatorterm0(qinfix, ⟨expr⟩)
5         else mk-Operatorapp0(qinfix, ⟨expr⟩) in
6   transform-expr(operator, context, sortset)(dict))
```

type: *Monadexpr*$_0$ *Context Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ [*Expression*$_1$ | *Term*$_1$] *Sortqual*-**set** *Dict*

目的 把一个前缀运算符应用变换到 AS$_1$

参数 参见函数 *transform-expr*

结果 参见函数 *transform-expr*

算法

第 1 行 从运算符符号构造受限定的前缀运算符(限定符为空)。

第 2—5 行 依据是用于公理还是用于表达式来构造运算符的项或运算符的应用。

第 6 行 用通常的方法变换构造的项(或表达式)。

transform-infixexpr(**mk-***Infixexpr*$_0$(*expr*1, *op*, *expr*2), *context*, *sortset*)(*dict*) $\triangleq$ (3.5.5.2)

```
1  (let qinfix = mk-Qualop0(⟨⟩, mk-Quotedop0(op)) in
2   let operator =
3       if context ∈ {MAPPING, AXIOMS}
4         then mk-Operatorterm0(qinfix, ⟨expr1, expr2⟩)
5         else mk-Operatorapp0(qinfix, ⟨expr1, expr2⟩) in
6   transform-expr(operator, context, sortset)(dict))
```

type: *Infixexpr*$_0$ *Context Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ [*Expression*$_1$ | *Term*$_1$] *Sortqual*-**set** *Dict*

目的 把中缀运算符应用变换到 AS$_1$

参数 参见函数 *transform-expr*

结果 参见函数 *transform-expr*

算法

第 1 行 从运算符符号构造受限定的中缀运算符(限定词是空的)。

第 2-5 行 依据是用于公理还是用于表达式来构造运算符的项或运算符的应用。

第 6 行 用通常的方法变换构造的项(或表达式)。

3.6 可见性处理

all-visible-sorts(*dict*) $\triangleq$ (3.6.1)

```
1  {qual ∈ dom dict | is-SortD(dict(qual)) ∧
2                     (∃q ∈ Qual)(get-sur(qual) ↷ q = dict(SCOPEUNIT))}
```

type: *Dict* $\rightarrow$ *Sortqual*-**set**

目的　　从 *Dict* 中抽取类别的 *Qual* 的集合，这些类别在由 **SCOPEUNIT** 指明的地方是可见的。

算法　　返回所有表示一个类别的 *Qual*（第 1 行），并且存在一个部分 *Qual*（*q*）使得定义该类别的作用域单位的 *Qual* 当与 *q* 汇合时可以构成当前作用域单位的 *Qual*。

all-visible-literals(*id*, *sortset*)(*dict*) ≙ **(3.6.2)**

```
1  (let (qual, nm) = cases id:
2                      (mk-Id0(q, n)          → (q, n),
3                       mk-Stringterm0(q, n) → (q, n)) in
4   {q ∈ dom dict | (∃squal ∈ sortset)(squal ⌢ ⟨(LITERAL, nm)⟩ = q) ∧
5                   (∃qu)(get-parent(qu ⌢ qual)(dict) ⌢ ⟨(LITERAL, nm)⟩ = q)})
```

type：　(*Id*$_0$ | *Stringterm*$_0$) *Sortqual*-**set** → *Dict* → *Qual*-**set**

目的　　抽取在一个给定位置可见的、可由某个 AS_0 标识符来表示的、并且是某些特定类别的所有字面值的 *Qual* 集合

参数

id　　可表示该字面值的 AS_0 标识符

sortset　　结果的字面值的 *Qual* 是包含在 *sortset* 中某个类别的全部

算法

第 1—3 行　　抽取此标识符（或字符串）的限定符和名字。

第 4 行　　返回所有的 *Qual*，它表示定义在一种可能类别中的字面值，并包括该限定符。

all-input-signals(*qual*)(*dict*) ≙ **(3.6.3)**

```
1  (let qual' = process-or-service-level(qual) in
2    cases dict(qual'):
3     (mk-ServiceD(,,, inp,) → inp,
4      mk-ProcessD(, inp,,) → inp))
```

type：　*Qual* → *Dict* → *Signalqual*-**set**

目的　　抽取输入信号（包括定时器在内）的 *Qual*

参数

qual　　是一个限定表 *Qual*，用来表示需要输入信号的层次（作用域单位）

算法

第 1 行　　在过程的情况下，用选出的外围进程或服务来修改层次。

第 2 行　　如果此作用域单位是一服务，则返回所有的输入信号。

第 3 行　　如果此作用域单位是一个进程，则返回所有的输入信号。

import-export-signals(*dict*) ≙ (3.6.4)

```
1  (let (emptyq, , ) = dict(GLOBALNAMES) in
2   let (exportinput, exportoutput) = extract-exports(dict) in
3   let importset = {(tq, nm) ∈ dom exportmap | (∃q ∈ dom dict)(process-level(q) = dict(SCOPEUNIT) ∧
4                                                              (IMPORT, (nm, tq)) = q[len q])} in
5   let importinput =
6       {xrq | (∃(tq, nm) ∈ importset)
7               ((let mk-SignalnamesD(, , xr') = exportmap((tq, nm)) in
8                 get-sur(tq) ⌢ ⟨(SIGNAL, xr')⟩ = xrq)} in
9   let importoutput =
10      {xqq | (∃(tq, nm) ∈ importset)
11              ((let mk-SignalnamesD(, xq, ) = exportmap((tq, nm)) in
12                get-sur(tq) ⌢ ⟨(SIGNAL, xq)⟩ = xqq)} in
13  (exportinput ∪ importinput ∪ {emptyq}, exportoutput ∪ importoutput))
```

type: *Dict* → *Signalqual*-**set** *Signalqual*-**set**

目的

抽取并返回与一进程的进口和出口相关的所有隐式信号的 *Qual*，并返回空队列(emptyq)信号

算法

第 1 行　抽取用于允许条件(或连续信号)中的空队列(emptyq)信号的 *Qual*。

第 2 行　构造当一个进程出口变量时用到的所有输入信号的 *Qual* 和所有输出信号的 *Qual*(即 xtQUERY 输入信号和 xtREPLY 输出信号)。

第 3—4 行　抽取进程中所有进口变量的 *NameclosureD*(参见 *exportmap* 的定义)。这个集合由所有的那些类别(*tq*)和名字(*nm*)的偶对组成，每个偶对表示存在一个被进口的变量定义在一个类别为 *tq* 的进程或所包含的服务 *nm* 中。

第 5—8 行　构造由一个进程使用的所有 xtREPLY 信号的 *Qual* 集合。此集合由所有那些 xtREPLY *Qual* 组成，这些 *Qual* 可从包含在 *importset* 的一个 *NameclosureD* 的一个 *SignalnamesD* 描述符中的 xtREPLY 名字产生。

第 9—12 行　对由一个进程使用的 xtQUERY 信号做同样的处理。

第 13 行　返回 xtQUERY 信号的 *Qual*、xtREPLY 信号的 *Qual* 和 emptyq 信号的 *Qual* 的并集。emptyq 信号不作为一输出信号返回，因为收集输出信号的目的是为了构造隐式信号路由，而信号路由不包含 emptyq 信号。

extract-exports(*dict*) ≜ (3.6.5)

```
1  (let vqualset = {qual ∈ dom dict | process-level(qual) = dict(SCOPEUNIT) ∧ is-VarD(dict(qual))} in
2   let closset =
3       {(tq, nm) ∈ dom dict | (∃q ∈ vqualset)
4                                ((let mk-VarD(t, , exp, , ) = dict(q) in
5                                  t = tq ∧ (VALUE, nm) = q[len q] ∧
6                                  exp = EXPORTED))} in
7   let inputsig =
8       {qual | (∃(tq, nm) ∈ closset)
9                ((let mk-SignalnamesD(, xq, ) = exportmap((tq, nm)) in
10                 get-sur(tq) ⌢ ⟨(SIGNAL, xq)⟩ = qual))},
11      outputsig =
12      {qual | (∃(tq, nm) ∈ closset)
13               ((let mk-SignalnamesD(, , xr) = exportmap((tq, nm)) in
14                 get-sur(tq) ⌢ ⟨(SIGNAL, xr)⟩ = qual))} in
15  (inputsig, outputsig))
```

type: *Dict* → *Signalqual*-**set** *Signalqual*-**set**

目的 抽取并返回由一出口进程使用的所有 xtQUERY 和 xtREPLY 信号的 *Qual*

算法

第 1 行 抽取进程变量的 *Qual* 集合。

第 2—6 行 从此变量集合构造由进程使用的名字闭包描述符 *NameclosureD* 的集合，即，从出口类别为 *tq* 的变量的出口映射表 *Exportmap* 构造由类别 *tq* 和变量名 *nm* 组成的偶对的集合。

第 7—10 行 从该名字闭包描述符 *NameclosureD* 的集合构造 xtQUERY 信号的 *Qual*（与处理函数 *import-export-signals* 中的 xtREPLY 信号方法相同）。

第 11—14 行 对 xtREPLY 信号做同样的处理。

第 15 行 返回输入和输出信号。

get-visible-qual(*entity*, *entityclass*)(*dict*) ≜ (3.6.6)

```
1  if entity = ENV then
2    ENV
3   else
4   (let (qual, nm) = cases entity:
5                       (mk-Id0(q, n)          → (q, n),
6                        mk-Stringterm0(q, n) → (q, n),
7                        mk-Qualop0(q, n)     → (q, n)) in
8    let level = dict(SCOPEUNIT) in
9    if (∃q ∈ dom dict)((∃q')(q' ⌢ qual ⌢ ⟨(entityclass, nm)⟩ = q) ∧
10                        (∃q'')(get-sur(q) ⌢ q'' = level)) then
11     (let q ∈ dom dict be s.t. (∃q')(q' ⌢ qual ⌢ ⟨(entityclass, nm)⟩ = q) ∧
12          (∃q'')(get-sur(q) ⌢ q'' = level ∧
13                 ¬(∃qu)(get-sur(q) ⌢ qu = level ∧ len qu < len q'')) in
14      if is-ErrorD(dict(q)) then
15        exit("§2.2.2: 同义词、生成程序或信号表是递归定义的")
16       else
17        q)
18     else
19      exit("§2.2.2: 标识符是不可见的"))
```

type: (*Id*0 | *Stringterm*0 | *Qualop*0 | ENV) *Quot* → *Dict* → (*Qual* | ENV)

目的　　如果这个标识符可见，为一个 AS_0 标识符构造限定表 *Qual*

参数

entity　　一个标识符(或字符串或中缀运算符)，要为它构造一个 *Qual*

entityclass　　是标识符的实体类(参见 Z.100 的 § 2.2.2)。实体类 **VALUE** 包括同义词、变量和值标识符。此函数不用于构造字面值和运算符的 *Qual*(由于专门的可见性规则)。

算法

第 1 行　　如果实体是 **ENV**，则返回 **ENV**(参见函数 *transform-channeldef*)。

第 4—7 行　　抽取标识符(字符串或中缀运算符)的限定词和名字。

第 9—10 行　　如果 *Dict* 中存在一个 *Qual*(该 *Qual* 包括此限定词和在其最右部分由实体类别与名字组成的偶对)(第 9 行)并且由 $qualifier_0$ 表示的作用域单位是确定的、可见的，则

第 11—13 行　　令 q 表示这样的一个 *Qual*(第 11—13 行)，并通过表明 q 必须表示相对于当前层(第 13 行)是"最近的"*Qual*，来使得选择成为唯一的。

第 14—15 行　　标识符一定不能递归地定义。

signal-qual(*sigid*)(*dict*) ≙ **(3.6.7)**

```
1  (let qual = (trap exit with nil in
2                {get-visible-qual(sigid, SIGNAL)(dict)}) in
3   if qual ≠ nil then
4     qual
5    else
6     (let mk-Id0(q, nm) = sigid in
7      let qualelem = q ⌢ ⟨(SIGNAL, nm)⟩ in
8      let level = dict(SCOPEUNIT) in
9      let qualset = {qu ∈ dom dict | (∃qua)(qua ⌢ qualelem = qu) ∧
10                                    (∃q', q'')(q' ⌢ q'' = level ∧ signaldeflevel(qu) = q')} in
11     if card qualset ≠ 1 then
12       exit("§2.2.2: 信号标识符表示不止一个(子)信号")
13      else
14       (let qual' ∈ qualset in
15        qual')))
```

type:　$Sigid_0 \to Dict \to Signalqual$

目的　　构造并返回一 AS_0 信号标识符的 *Qual*。函数 *get-visible-qual* 不能直接对信号使用的原因是由于子信号概念使得信号的可见性规则更为复杂。

参数

sigid　　AS_0 信号标识符

算法

第 1—3 行　　如果 *sigid* 不是子信号，则用函数 *get-visible-qual* 就可以找到限定表 *Qual*。

第 7 行　　构造部分 *Qual*，包括含有此信号名的 *Qualelem*。

第 8 行　　*level* 表示使用 *sigid* 的作用域单位。

第 9 行　　此信号标识符可能的 *Qual* 的完全集是表示这些子信号的 *Qual*。子信号的 *Qual* 集合是那些以此部分 *Qual* 作为结束部分(第 9 行)的 *Qual*，并是在一可见信号定义 q' 中包含的 *Qual*。

第 11—15 行　　如果 *Sigid* 有且仅有一个可能的 *Qual*，则返回该 *Qual*。

signaldeflevel(*qual*) ≜ (3.6.8)

```
1  if s-Kind(qual[len qual]) = SIGNAL then
2    signaldeflevel(get-sur(qual))
3   else
4    qual
```

type: *Signalqual* → *Signalqual*

目的　　从表示一个(子)信号定义的作用域单位的 *Qual* 构造最外层信号定义的作用域单位的 *Qual*

参数

算法

第 1—2 行　　如果最后一个限定表元素 *Qualelem* 包含实体类 **SIGNAL**，则该限定表 *Qual*（仍然）表示一个信号定义而且在再次应用此函数之前，删去该限定表元素 *Qualelem*。

get-visible-variable(*id*, *sortset*, *export*)(*dict*) ≜ (3.6.9)

```
1  (let qualset = all-variables-and-synonyms(id, EXPRESSION, sortset)(dict) in
2   let qualset' =
3       {qual ∈ qualset | cases dict(qual):
4                           (mk-VarD(,e,,,)
5                              → export = false ∨ e = EXPORT,
6                            mk-SynD(,)
7                              → false)} in
8   if card qualset' = 1 then
9     (let qual ∈ qualset' in
10     qual)
11    else
12     exit("§2.2.2: 有且仅有一个变量必须与该上下文匹配")
```

type: Id_0 *Sortqual-set Bool* → *Dict* → *Qual*

目的　　抽取一变量的限定表 *Qual*。函数 *get-visible-variable* 仅应用于不允许用同义词的地方（例如，赋值句、IN/OUT 参数等）。

参数

id	变量标识符
sortset	在该上下文中允许的类别集合
export	是一个标志，用来表示该变量是否用在一出口动作中

结果　　该上下文中唯一的 *Qual*

算法

第 1 行　　构造由（与 *Sortset* 中的类别之一匹配的）那些变量和同义词组成的限定表 *Qual* 的集合。

第 2—6 行　　限制此集合使之仅含有变量。并且如果在出口动作中使用了 *id*，则该集合中的变量必须出口。

第 10 行　　该集合必须包含一个唯一的 *Qual*。

3.7 过程和进程图的变换

在这一节，将介绍进程或过程体的变换。其结果是一个进程图 *process-graph*1、或是一个过程图 *procedure-graph*1 和一个字典 *Dict*，它带有在体中用到的隐式变量和输出信号的信息。如果进程包含服务，则也返回某些 AS_1 定义。变换一个进程体的入口函数是 *transform-process-body*，而变换过程体的入口函数是 *transform-body*（如果此进程不包含服务的分解，此函数也被变换进程体函数 *transform-process-body* 使用）。为了简便起见，跃迁仅遍历一次，这意味着当变换跃迁时，该跃迁中的速记符（即任选的结点、用划线表示的下一状态和进口/出口）在变换过程中被扩展。

体的变换由下述步骤组成：

1. 在判定的回答中和任选中插入一些终端符，使得每个 AS_0 跃迁 $Transition_0$ 含有一个终端语句 $Termstmt_0$。
2. 构造标号字典 *Labeldict*，即在体中使用的标号和它们关联的跃迁的集合。
3. 构造一个状态字典 *Statedict*，即把状态表变换为一个映射表。在变换状态的时候，这是一种更适当的表示法。在这个构造过程中，删除状态的多次出现和星号状态。
4. 扩展星号输入和星号保存。同时也把隐式跃迁加进去。因为由出口隐含的信号和空队列信号被所有服务共享，所以在这些服务被合并以前，不会把这些信号引起的跃迁加进去。
5. 如果进程包含服务，则把所有服务的状态字典 *Statedict* 合并（“状态爆发”）为一个 *Statedict*，其中每个项目都是一个新状态名。
6. 允许条件和连续信号的扩展。通过修改构造的 *Statedict* 来进行这种扩展。
7. 把状态字典 *Statedict* 和所包含的跃迁变换为一进程图 $Process\text{-}graph_1$ 或过程图 $Procedure\text{-}graph_1$。在这一变换期间，在 *Statedict* 中要增加进口表达式所隐含的新项目。

transform-process-body(*body*)(*dict*) $\triangleq$ **(3.7.1)**

```
1  if is-Body0(body) then
2    (let (as1 body, bdict) = transform-body(body)(dict) in
3     ({},as1 body, bdict, []))
4   else
5    (let mk-Decomposition0(decll) = body in
6     let (as1 dclset, sdict) = transform-decllist(decll)(dict) in
7     let connectmap = service-connectmap(decll, {}, [])(dict) in
8     if is-wf-servicesignals(dict) then
9       (let (as1 body, dict') = transform-decomposition-body(dict) in
10       (as1 dclset, as1 body, sdict + dict', connectmap))
11     else
12      exit("§4.10.2: 服务信号不是良形式的"))
```

type: $Processbody_0 \rightarrow Dict \rightarrow Decl_1$-**set** $Process\text{-}graph_1$ *Dict ProcessconnectionD*

目的 把一个进程体 $Processbody_0$ 变换为它的 AS_1 表示法

参数

body 进程体

结果　　包括有 AS_1 定义表（如果进程包含服务，则该表非空）、AS_1 进程体、一个带有在进程体中用到的隐式变量和输出信号等信息的 *Dict*，还包括进程的信号路由和服务信号路由的连接

算法

第 1—3 行　　如果进程不含服务，则变换包含在进程中的状态体（类似于过程情况）。否则

第 5 行　　令 *decll* 表示一组服务定义和服务信号路由定义。

第 6 行　　变换那些服务和信号路由。在此变换中，用服务中所包含的定义来替换服务。它们的状态体被表示为这些服务的描述符（包含在 *sdict* 里）中的 *Statedict*。

第 8 行　　这些服务的有效输入信号集必须是不相交的，在进程外部的信号路由中没有信号可以是高优先级信号。

第 9 行　　从包含在 *dict* 的服务描述符中的 *Statedict* 构造 AS_1 过程体。

第 10 行　　返回来自这些服务的 AS_1 定义，返回进程体，以及返回从定义和从服务中用到的隐式变量、输出信号等得到的对字典 *Dict* 的贡献。

transform-body(**mk-***Body*$_0$(*trans*, *statelist*))(*dict*) ≙ **(3.7.2)**

```
1  (let trans' = insert-trans-term(trans) in
2   let statelist' = ⟨insert-state-term(statelist[i]) | 1 ≤ i ≤ len statelist⟩ in
3   let labeldict = build-trans-labeldict(trans')([]) in
4   let labeldict' = build-state-labeldict(statelist')(labeldict) in
5   let statedict = build-statedict(statelist', dict(SCOPEUNIT))([]) in
6   let statedict' = remove-asterisk-input-and-save(statedict)(dict) in
7   let statedict'' = remove-cont-enable-from-statelist(dom statedict)(dict, statedict') in
8   let dict' = dict + [LABELDICT ↦ labeldict',
9                       STATEDICT ↦ statedict''] in
10  let (trans1, dict'') = transform-transition(nil, trans', ⟨⟩)(dict') in
11  let (statebody1, dict''') = transform-statelist({})(dict'') in
12  let transition1 = add-varinit(trans1, dict''') in
13  if is-ProcedureD(dict(dict(SCOPEUNIT))) then
14    (mk-Procedure-graph1(mk-Procedure-start-node1(transition1), statebody1), dict''')
15   else
16    (mk-Process-graph1(mk-Process-start-node1(transition1), statebody1), dict'''))
```

type:　*Body*$_0$ → *Dict* → (*Procedure-graph*$_1$ | *Process-graph*$_1$) *Dict*

目的　　变换进程或过程的状态体

参数　　一个 AS_0 体包含

trans　　初始跃迁

statelist　　状态表

结果　　一个 AS_1 过程图或进程图和一个字典 *Dict*，它带有隐式变量和输出信号的信息

算法

第 1 行　　在初始跃迁中，在判定的回答和任选中插入终端符。

第 2 行　　在状态表中，在判定的回答和任选中插入终端符。

第 3 行　　收集用于初始跃迁中有关标号（连接符）的信息。

第 4 行　　收集用于状态表中有关标号的信息。*labeldict'* 反映用于进程或过程体中的标号和跟随

它们的跃迁之间的联系（参见 *Labeldict* 的定义）。

第 5 行　把状态表变换为一个映射表，从状态名映射到状态描述符 *stateD*（参见 *Quotdict* 中的 *Statedict* 定义）。状态的多次出现和星号状态在该变换期间被合并。

第 6 行　扩展 *Statedict* 中的星号输入、星号保存和隐式跃迁。

第 7 行　扩展在构造的 *Statedict* 中的允许条件和连续信号。在这一扩展之后，*Statedict* 包含被删除状态的 *ContenablestateD* 描述符。新状态用一个 *StateD* 描述符表示。

第 8 行　把 *Labeldict* 和 *Statedict* 放入字典 *Dict* 中，使得变换函数仅需要一个字典参数（而不是三个）。

第 10 行　把初始跃迁变换到 AS_1。

第 11 行　把状态表变换为 AS_1 状态集合。

第 12 行　修改初始跃迁以便包括用于进程中的隐式变量的初始化。

第 13—16 行　如果状态体属于一个过程，则返回一个 AS_1 过程图。否则返回一个 AS_1 进程图。在所返回的字典 *Dict*（*dict‴*）中只用到 *Quotdict* 中的 **OUTSIGNAL** 和 **IMPLIED** 项目。

add-varinit(*trans*, *dict*) ≙ **(3.7.3)**

```
1  (let qualset = {qual ∈ dom dict | is-VarD(dict(qual)) ∧
2                                    get-sur(qual) = dict(SCOPEUNIT)} in
3   add-default-assign(trans, qualset)(dict))
```

type: $Transition_1$ *Dict* → $Transition_1$

目的　修改初始跃迁以包括变量的初始化

参数

trans　AS_1 初始跃迁

dict　字典 *Dict*，它包含相关的 *VarD* 描述符和附加在连续信号上的 AS_0 变量名（该名字对所有进程都一样）

结果　修改过的 AS_1 跃迁

算法

第 1—2 行　抽取所有表示局部变量的那些 *Qual* 的集合。

第 3 行　修改跃迁以包括对局部变量的缺省赋值。

add-default-assign(*trans*, *descrset*)(*dict*) ≜ **(3.7.4)**

```
 1  if descrset = {} then
 2    trans
 3   else
 4    (let qual ∈ descrset in
 5     let trans' = add-default-assign(trans, descrset \ {qual})(dict) in
 6     let mk-VarD(,,, expr1, qual) = dict(qual) in
 7     if expr1 = nil then
 8       trans'
 9      else
10       (let as1 id = make-as1-identifier(qual)(dict) in
11        let stmt = mk-Task-node1(mk-Assignment-statement1(as1 id, expr1)) in
12        let mk-Transition1(actl, term) = trans' in
13        let trans'' = mk-Transition1(stmt ↷ actl, term) in
14        if is-SyntypeD(dict(qual)) then
15          (let range = s-Range-condition1(dict(qual)) in
16           let mk-Range-condition1(, cset) = range in
17           if cset = {} then
18             trans''
19            else
20             (let des = mk-Decision-node1(expr1, {mk-Decision-answer1(range, trans'')},
21                                          mk-Else-answer1(trans')) in
22              mk-Transition1(⟨⟩, des)))
23         else
24          trans''))
```

type: *Transition*$_1$ *Qual*-**set** → *Dict* → *Transition*$_1$

目的

修改初始跃迁以包括作用域单位的局部变量的缺省赋值。因为对附属于出口变量的隐式变量也有一个 *VarD* 描述符，所以隐式出口变量被自动地初始化。

参数

trans	将被修改的跃迁
descrset	局部变量的 *Qual*

结果

修改后的跃迁

算法

第 1 行	当全部处理完时，不再修改该跃迁。
第 4 行	令 *qual* 表示在 *Qual* 集合中的一个元素。
第 5 行	构造跃迁，该跃迁就是经过修改的 *trans* 以包括对其余的 *Qual* 的变量初始化。
第 6 行	分解变量的描述符，以便透露初始表达式（$expr_1$）和变量的类别。请注意，在导出该初始表达式时没有用 *qual*（这一工作已经做过了）。
第 7 行	如果描述符不含初始表达式，则返回没有修改过的跃迁。
第 10 行	令 $as_1 id$ 表示变量的 AS_1 标识符。
第 11 行	令 *stmt* 表示含有缺省赋值的任务。
第 12—13 行	令 *trans''*表示被修改的跃迁。
第 14—17 行	如果变量类别是同义类型的，则必须构造一判定。
第 15 行	令 *range* 表示同义类型的 AS_1 范围条件。
第 16—17 行	如果在范围 *range* 中的条件项 *Condition-item* 的集合为空，则不要构造同义类型，因而返回修改过的跃迁。

第 20—22 行　　返回由一空动作表和一个 AS_1 判定所组成的新跃迁。此 AS_1 判定仅在范围条件被满足的情况下能够执行缺省赋值。

3.7.1　在回答中插入终端符

insert-state-term(**mk-***Statebody*$_0$(*idl*, *stlist*, *tnm*)) ≙　　(3.7.1.1)

```
1  (let stlist' = ⟨insert-spec-term(stlist[i]) | 1 ≤ i ≤ len stlist⟩ in
2   mk-Statebody0(idl, stlist', tnm))
```

type:　*Statebody*$_0$ → *Statebody*$_0$

目的　　在一 AS_0 状态中所包含的判定回答和任选中插入终端符

参数　　此状态含有

id　　此状态的状态名表

结果　　修改过的 AS_0 状态

算法

第 1 行　　在所有的输入跃迁中插入终端符，并且

第 2 行　　（再一次）构造 AS_0 状态。

insert-spec-term(*stspec*) ≙　　(3.7.1.2)

```
1  cases stspec:
2   (mk-Inputspec0(vars, enab, trans)
3      → mk-Inputspec0(vars, enab, insert-trans-term(trans)),
4    mk-Contspec0(expr, prio, trans)
5      → mk-Contspec0(expr, prio, insert-trans-term(trans)),
6    mk-Priinput0(vars, trans)
7      → mk-Priinput0(vars, insert-trans-term(trans)),
8    T → stspec)
```

type:　*Statespec*$_0$ → *Statespec*$_0$

目的　　在一输入跃迁中所包含的判定回答和任选中插入终端符

参数

stspec　　输入跃迁

结果　　修改过的 AS_0 输入跃迁

算法

第 2 行　　如果该输入跃迁是一个输入节点，则在后随输入结点的跃迁（*trans*）中插入终端符。

第 4 行　　如果输入跃迁是一连续信号，则在后随连续信号的跃迁（*trans*）中插入终端符。

第 6 行　　如果输入跃迁是一优先级输入，则在后随该优先级输入的跃迁(*trans*)中插入终端符。

第 8 行　　否则（在保存节点的情况下），返回未改变的跃迁。

insert-trans-term(mk-*Transition*$_0$(*actl*, *term*)) ≙ (3.7.1.3)

```
1  (if actl = ⟨⟩ then
2     mk-Transition0(⟨⟩, term)
3    else
4     (let mk-Actstmt0(l, act) = hd actl in
5      if is-Decision0(act) ∨ is-Option0(act) then
6        (let (answerl, els) = cases act:
7                                (mk-Decision0(, ansl, els)
8                                   → (ansl, els),
9                                 mk-Option0(, ansl, els)
10                                  → (ansl, els)) in
11        let (actl′, answerl′, els′) = insert-answer-term(tl actl, term, answerl, els) in
12        let act′ = cases act:
13                     (mk-Decision0(q, , )
14                        → mk-Actstmt0(l, mk-Decision0(q, answerl′, els′)),
15                      mk-Option0(q, , )
16                        → mk-Actstmt0(l, mk-Option0(q, answerl′, els′))) in
17        let mk-Transition0(actl″, t) = insert-trans-term(mk-Transition0(actl′, term)) in
18        mk-Transition0(⟨act′⟩ ⌢ actl″, t))
19       else
20        if tl actl = ⟨⟩ ∧ term = nil then
21          exit("§2.6.7.1: 在跃迁中丢失终端符")
22         else
23          (let mk-Transition0(actl″, t) =
24               insert-trans-term(mk-Transition0(tl actl, term)) in
25           mk-Transition0(⟨hd actl⟩ ⌢ actl″, t))))
```

type: *Transition*$_0$ → *Transition*$_0$

目的 在一动作表中的判定回答和任选节点中插入终端符

参数 此跃迁含有

actl 动作表

term 后随动作表的任选终端符

结果 修改过的 AS$_0$ 跃迁

算法

第 1 行 当全部处理完时，返回含有空动作表的跃迁（本函数是递归的）。

第 4 行 分解第一个动作语句。

第 5 行 如果动作是一判定或一任选，则

第 6—9 行 从判定或任选中抽取回答表（*answerl*）和 else 部分（*els*）。

第 11 行 在回答表和 else 部分中插入终端符。如果插入了一个终端符（一个汇接 join），则把动作表的其余部分（**tl** *actl*）修改为 *actl′* 以包括一个标号。

第 12—15 行 构造修改过的判定或任选节点。

第 17 行 在动作表的其余部分中插入终端符。

第 18 行 返回包含修改过的判定或任选（*act′*）的跃迁、动作表的其余部分（*actl″*）和终端符。

第 20—25 行 如果第一个动作不是判定或任选节点，则

第 20—21 行 如果动作是最后一个动作并且没有终端符，则是一个错误。

第 23—25 行 在动作表的其余部分中插入终端符，并且组合修改过的跃迁。

insert-answer-term(*actl*, *term*, *answerl*, *els*) $\triangleq$ (3.7.1.4)

```
 1  (if answerl = ⟨⟩ then
 2     if els = nil then
 3       (actl, ⟨⟩, nil)
 4      else
 5       (let mk-Elsepart0(trans) = els in
 6        let (actl', trans') =
 7            cases trans:
 8             (nil
 9                → (let (acl, term') = insert-join(actl, term) in
10                     (acl, mk-Transition0(⟨⟩, term'))),
11              mk-Transition0(actl'', term'')
12                → if term'' = nil then
13                     (let (actl''', term''') = insert-join(actl, term) in
14                      (actl''', insert-trans-term(mk-Transition0(actl'', term'''))))
15                   else
16                    (actl, insert-trans-term(trans))) in
17        (actl', ⟨⟩, mk-Elsepart0(trans')))
18    else
19     (let (actl', answerrest, els') = insert-answer-term(actl, term, tl answerl, els) in
20      let mk-Answer0(vset, trans) = hd answerl in
21      let (actlist, trans'') =
22          cases trans:
23           (nil
24              → (let (acl, term) = insert-join(actl', term) in
25                   (acl, mk-Transition0(⟨⟩, term))),
26            mk-Transition0(actl'', term'')
27              → if term'' ≠ nil then
28                   (let (actl''', term''') = insert-join(actl', term) in
29                    (actl''', insert-trans-term(mk-Transition0(actl'', term'''))))
30                 else
31                  (actl', insert-trans-term(trans))) in
32      (actlist, ⟨mk-Answer0(vset, trans'')⟩ ⌢ answerrest, els')))
```

type: $Actstmt_0^*$ [$Termstmt_0$] $Answer_0^*$ [$Elsepart_0$] $\rightarrow$ $Actstmt_0^*$ $Answer_0^*$ [$Elsepart_0$]

目的

在一判定的回答或任选节点中插入终端符

参数

actl	跟随判定或任选节点的动作表
term	跟随判定或任选节点和跟随动作表的终端符
answerl	回答表
els	任选的 else 部分

结果

- 跟随判定或任选节点的动作表。如果动作表非空并且如果在回答中和在 else 部分中丢失了任一个终端符，则修改动作表使之包括一个标号。
- 回答表，各个回答都含有终端符。
- 包含一终端符的 else 部分（除非省略了 else 部分）。

算法

第 1—17 行	当通过回答表时，则修改该 else 部分。
第 2 行	如果省略了 else 部分，则返回动作表和回答的空表，并没有 else 部分
第 5 行	如果规定了 else 部分，则分解它。

第 6—17 行　令 *actl'* 表示跟随判定或任选节点的新动作表，并令 *trans'* 表示在 else 部分中含有一终端符的新跃迁。

第 8 行　如果没有规定跃迁，则构造一个汇接（*term*）并在跟随判定或任选节点的动作表（*acl*）中插入一个标号。else 部分的跃迁是一空动作表和汇接（第 10 行）。

第 11—14 行　如果在 else 部分中规定了跃迁，但省略了终端符，则在跟随判定或任选的动作表（*actl'''*）中插入一个标号，构造一个汇接（*term'''*）并在该跃迁动作表中插入终端符（第 14 行）。

第 16 行　如果规定了一个含有一终端符的跃迁，则在该跃迁的动作表中插入终端符，并且不要修改跟随判定或任选的动作表（*actl*）。

第 17 行　返回修改后的动作表、一个空回答表和修改后的 else 部分。

第 19 行　在其余的回答中插入终端符。

第 20 行　分解表中的第一个回答。

第 21—31 行　对回答表中的第一个回答进行处理，方法和在第 6—16 行中规定的对 else 部分的处理方法一样。

第 32 行　返回跟随判定或任选的新动作表、第一个被修改的回答（和其余被修改回答汇合）以及修改过的 else 部分（如果规定了的话）。

insert-join(*actl*, *term*) ≙　**(3.7.1.5)**

```
 1  (if actl = ⟨⟩ then
 2     if term = nil then
 3        exit("§2.6.7.1: 在跃迁中丢失终端符")
 4      else
 5       (let mk-Termstmt0(, t) = term in
 6        (⟨⟩, mk-Termstmt0(nil, t)))
 7    else
 8     (let mk-Actstmt0(l, act) = hd actl in
 9      let l' = if l = nil then create-unique-name() else l in
10      (⟨mk-Actstmt0(l', act)⟩ ⌢ tl actl, mk-Termstmt0(nil, mk-Join0(l')))))
```

type:　$Actstmt_0^*\ [Termstmt_0] \rightarrow Actstmt_0^*\ Termstmt_0$

目的　构造一个动作表和后随的终端符，构造连接到动作表中第一个动作的一个汇接，或如果动作表空的话，则构造一个连接到终端符的汇接。该汇接在没有终结判定的回答中用作为一终端符（参见插入回答项函数 *insert-answer-term*）。

参数

actl　给定的动作表

term　跟随动作表的终端符

结果　一个动作表（其第一个动作和所构造的汇接有相同的标号）和构造的汇接

算法

第 1 行　如果动作表为空，则

第 2—6 行　终端符必须出现。如果出现的话，则

第 5—6 行　返回一个终端符(但未加标号)，它和在跟随判定或任选的跃迁中规定的终端符相同。

第 8 行　分解跟在包围的判定或任选后面的动作表中的第一个动作。

第 9 行　从该动作抽取标号。如果该动作无标号，则创建一个。

第 10 行　返回包含该标号的新动作表，并返回终端符语句，它包含连到那个标号的汇接。

3.7.2 标号字典 Labeldict 的建立

build-state-labeldict(*statebody*)(*labeldict*) ≙ (3.7.2.1)

```
1  (if statebody = ⟨⟩ then
2     labeldict
3   else
4    (let mk-Statebody0(, specl, ) = hd statebody in
5     let specl' = ⟨specl[i] | 1 ≤ i ≤ len specl ∧ ¬is-Savespec0(specl[i])⟩ in
6     let labeldict' = build-spec-labeldict(specl')(labeldict) in
7     build-state-labeldict(tl statebody)(labeldict')))
```

type: $Statebody_0^* \rightarrow Labeldict \rightarrow Labeldict$

目的 从一个进程、过程或服务的状态体，收集标号和与这些标号有关的跃迁以构成一个标号字典 *Labeldict*（参见 *Labeldict* 的定义）

参数

statebody 状态表

labeldict 至今所收集到的标号（本函数是递归的）

结果 构造的标号字典 *Labeldict*

算法

第 1 行 当全部处理完时，返回构造的标号字典 *labeldict*。

第 4 行 分解表中第一个状态。

第 5 行 抽取不保存的那些输入跃迁。

第 6 行 更新标号字典 *labeldict* 以便包括在这些输入跃迁中的标号。

第 7 行 收集其余状态中的标号。

build-spec-labeldict(*speclist*)(*labeldict*) ≙ (3.7.2.2)

```
1  (if speclist = ⟨⟩ then
2     labeldict
3   else
4    (let trans = s-Transition0(hd speclist) in
5     let labeldict' = build-trans-labeldict(trans)(labeldict) in
6     build-spec-labeldict(tl speclist)(labeldict')))
```

type: $Statespec_0^* \rightarrow Labeldict \rightarrow Labeldict$

目的 从附加在一状态上的跃迁（输入跃迁）收集标号和与这些标号相关的跃迁以形成一个标号字典 *Labeldict*（参见 *Labeldict* 的定义）

参数

speclist 输入跃迁表

labeldict 至今所收集到的标号（本函数是递归的）

结果 构造的标号字典 *Labeldict*

算法

第 1 行　　当全部处理完时，返回构造的标号字典 *Labeldict*。
第 4—5 行　　更新标号字典 *labeldict* 以包括第一个输入跃迁中的标号。
第 6 行　　为其余的输入跃迁收集标号。

build-trans-labeldict(mk-*Transition*$_0$(*actl*, *term*))(*labeldict*) ≙ (3.7.2.3)

```
1  (if actl = ⟨⟩ then
2     (if term = nil then
3        labeldict
4      else
5       (let mk-Termstmt0(l, ) = term in
6        if l = nil then
7          labeldict
8         else
9          if l ∈ dom labeldict then
10           exit("§2.6.6: 标号相同")
11         else
12          labeldict + [l ↦ mk-Transition0(⟨⟩, term)]))
13   else
14    (let mk-Actstmt0(l, a) = hd actl in
15     let labeldict' =
16        if l = nil then
17          labeldict
18         else
19          if l ∈ dom labeldict then
20            exit("§2.6.6: 标号相同")
21           else
22            labeldict + [l ↦ mk-Transition0(actl, term)] in
23     let labeldict'' =
24        cases a:
25        (mk-Decision0(, ansl, els)
26           → build-answerlist-labeldict(ansl, els)(labeldict'),
27         mk-Option0(, ansl, els)
28           → build-answerlist-labeldict(ansl, els)(labeldict'),
29         T → labeldict') in
30     build-trans-labeldict(mk-Transition0(tl actl, term))(labeldict'')))
```

type: *Transition*$_0$ → *Labeldict* → *Labeldict*

目的　　从一个跃迁收集标号和与它们相关的跃迁以构成一个标号字典 *Labeldict*（参见 *Labeldict* 的定义）

参数　　一个跃迁含有

actl	动作语句表
term	终端符语句
labeldict	至今所收集到的标号

结果　　构造的标号字典 *Labeldict*

算法

第 1—12 行　　当处理完动作语句表时，考虑处理终端符。
第 5 行　　分解终端符语句以便抽取标号（*l*）。
第 6 行　　如果省略了标号，则返回构造的标号字典。
第 9 行　　如果此标号在 *Labeldict* 中已经有了，则此标号已被使用了两次。

第 12 行　　返回 *Labeldict*，它已被更新以包括终端符语句中的标号。跟随此标号的跃迁 $Transition_0$ 有一空的动作语句表和一个终端符语句。

第 14 行　　分解此表中第一个动作语句。

第 15—22 行　　令 *labeldict'* 表示更新的标号字典，它包括第一个动作语句的标号（如果它出现并且标号是不同的）。

第 23—29 行　　令 *labeldict''* 表示进一步更新的 *labeldict*，它包括在判定情况下的回答的标号或任选动作中的标号。

第 30 行　　收集其余动作语句中的标号。

build-answerlist-labeldict(*answerl*, *els*)(*labeldict*) ≙ **(3.7.2.4)**

```
 1  (if answerl = ⟨⟩ then
 2     if els = nil then
 3        labeldict
 4      else
 5       (let mk-Elsepart0(trans) = els in
 6        build-trans-labeldict(trans)(labeldict))
 7    else
 8     (let mk-Answer0(, trans) = hd answerl in
 9      let labeldict' = build-trans-labeldict(trans)(labeldict) in
10       build-answerlist-labeldict(tl answerl, els)(labeldict')))
```

TYPE: $Answer_0^*$ [$Elsepart_0$] → *Labeldict* → *Labeldict*

目的　　从一判定的回答或任选动作收集标号和与标号有关的跃迁，以便构成一个标号字典 *Labeldict*（参见 *Labeldict* 的定义）

参数

answerl　　回答表

els　　任选的 else 部分

labeldict　　至今所收集到的标号

结果　　构造的 *Labeldict*

算法

第 2—6 行　　当处理完回答表时，考虑 else 部分。

第 2 行　　如果此判定或任选动作不包含 else 部分，则返回此 *labeldict*。

第 5 行　　从 else 部分抽取跃迁。

第 6 行　　更新标号字典 *labeldict* 以包括 else 部分的跃迁中的标号。

第 8 行　　抽取表中第一个回答的跃迁。

第 9 行　　更新 *labeldict* 以包括跃迁的标号。

第 10 行　　收集其余回答的标号。

3.7.3 状态字典 statedict 的建立

build-statedict(*statelist*, *scope*)(*statedict*) ≜ (3.7.3.1)

```
1  (if statelist = ⟨⟩ then
2     statedict
3   else
4    (let mk-Statebody0(statenml, stspec, tailname) = hd statelist in
5     let speclist = ⟨(scope, stspec[i]) | 1 ≤ i ≤ len stspec⟩ in
6      cases statenml:
7      (mk-Statenamelist0(nmlist)
8         → if tailname ≠ nil ∧ elems nmlist ≠ {tailname} then
9              exit("§2.6.3: 状态的结尾名必须和起始名相同")
10            else
11             (let statedict' = insert-state-names(nmlist, speclist)(statedict) in
12              build-statedict(tl statelist, scope)(statedict')),
13       mk-Starredlist0(nmlist)
14         → if tailname ≠ nil then
15              exit("§2.6.3: 结尾状态名不允许出现在星号状态中")
16            else
17             (let totaldict = build-statedict(tl statelist, scope)(statedict) in
18              insert-starred(nmlist, speclist)(totaldict)))))
```

type: $Statebody_0^*$ *Qual* → *Statedict* → *Statedict*

目的 从一个进程、过程或服务的状态体，收集这些状态和与它们相关的跃迁以便构成一个状态字典 *Statedict*（参见 *Statedict* 的定义）

参数

statebody	状态表
scope	是一个限定表 *Qual*，表示包含这些状态的作用域单位。在状态字典 *Statedict* 中每一个跃迁必须包括精心制造跃迁的上下文信息(因为若干服务的状态字典 *Statedict* 被合并成一个状态字典 *Statedict* 后才把它们转换到 AS_1)。
statedict	至今所考虑的状态（本函数是递归的）。

结果 构造的状态字典 *Statedict*

算法

第 1 行	当全部处理完时，返回构造的状态字典 *Statedict*。
第 4 行	分解在表中的第一个状态。
第 5 行	构造此状态的 *Spec* 表（参见状态字典 *Statedict* 的定义），即此状态的上下文信息和输入跃迁组成的偶对的表。
第 7—12 行	如果在此状态中规定了一个状态名表，则
第 9 行	如果规定了一个尾名，则状态名表必须含有与此相同的名字。
第 11 行	对表中的每个状态名，增加一个状态描述符 *StateD* 到字典 *Statedict* 中。
第 12 行	对其余那些状态构造 *StateD* 描述符。
第 13—18 行	如果规定了一个星号状态名字表，则
第 14 行	不允许用尾名。
第 17 行	为其余状态构造状态字典 *Statedict*（完整的 *statedict*），然后
第 18 行	增加星号状态的输入跃迁。

insert-state-names(*namelist*, *stspec*)(*statedict*) ≜ **(3.7.3.2)**

```
1  (if namelist = ⟨⟩ then
2     statedict
3    else
4     (let nm = hd namelist in
5      let stspec' =
6          if nm ∈ dom statedict then
7            (let mk-StateD(specl, ) = statedict(nm) in
8             specl)
9           else
10           ⟨⟩ in
11     let statedict' = statedict + [nm ↦ mk-StateD(stspec ⌢ stspec', nil)] in
12     if nm ∈ elems tl namelist then
13       exit("§2.6.3: 状态中的状态名必须是不同的 ")
14      else
15       insert-state-names(tl namelist, stspec)(statedict')))
```

type: *Statenamelist*$_0$ *Speclist* → *Statedict* → *Statedict*

目的 更新状态字典 *Statedict* 以便包括一个状态节点的输入跃迁

参数

namelist 在状态节点中规定的状态名字表

stspec 一个输入描述 *Spec* 表，它包含输入跃迁，并且（对于表中的每个状态）它被加入到 *Statedict* 中。

结果 更新的状态字典 *Statedict*

算法

第 1 行 当处理完状态名字表时，返回更新的状态字典 *Statedict*（本函数是递归的）。

第 4 行 令 *nm* 表示表中的第一个名字。

第 5—8 行 如果那个名字已有一个在 *statedict* 中的项目，则 *stspec'* 表示与该项目相关的输入跃迁，否则 *stspec'* 就表示输入跃迁的空表。

第 11 行 令 *statedict'* 表示更新的状态字典 *Statedict*，以便包括已经在那里的输入跃迁（*stspec'*）和新的输入跃迁（*stspec*）。在 *StateD* 中的 **nil** 表示状态不是由一个进口表达式隐含的隐式状态（参见 *Statedict* 的定义）。

第 12—13 行 在此状态中规定的状态名表必须包含不同的状态名。

第 15 行 处理状态名表中其余的名字。

insert-starred(*namelist*, *stspec*)(*totaldict*) ≜ **(3.7.3.3)**

```
1  (if card elems namelist ≠ len namelist ∨
2     ¬(elems namelist ⊆ dom totaldict) ∨
3     elems namelist = dom totaldict ∨
4     totaldict = [] then
5      exit("§4.4: 非法的星号状态 ")
6     else
7      (let restset = dom totaldict − elems namelist in
8       let namelist' = ⟨nm | nm ∈ dom restset⟩ in
9        insert-state-names(namelist', stspec)(totaldict)))
```

type: *Statenamelist*$_0$ *Speclist* → *Statedict* → *Statedict*

目的 更新状态字典 *Statedict* 以包括来自星号状态的输入跃迁

参数

namelist 在星号状态中提及的名字

stspec 包含星号状态的输入跃迁的 *Spec*

结果 更新的状态字典 *Statedict*

算法

第 1 行 在状态名字表中的名字必须各不相同，并且

第 2 行 它们必须不是此状态名的唯一出现，并且

第 3 行 它们必须不包括所有状态，并且

第 4 行 它们至少必须有一个是不带星号的状态。

第 7 行 抽取由星号状态复盖的状态集合。

第 8 行 把集合变换为一个表。

第 9 行 为导出的状态表增加 *Spec* 到 *Statedict* 中（用通常的方法）。

3.7.4 扩展星号输入、星号保存和隐式的跃迁

在这一节中，把输入节点和保存节点的星号从进程体、过程体或服务体中删除，并控制每个状态至多包含一个星号。因为允许条件、连续信号和进口表达式尚未删除，星号的替换不包括空队列信号和由进口隐含的信号。

$$remove\text{-}asterisk\text{-}input\text{-}and\text{-}save(statedict)(dict) \triangleq \quad \textbf{(3.7.4.1)}$$

1 $[stnm \mapsto remove\text{-}asterisk\text{-}from\text{-}state(statedict(stnm))(dict) \mid stnm \in \textbf{dom}\ statedict]$

type: $Statedict \rightarrow Dict \rightarrow Statedict$

目的 从一个进程、过程或服务的状态体中删除星号输入和星号保存

参数

statedict 表示状态体的状态字典 *Statedict*

结果 修改过的 *Statedict*

算法 构造状态字典 *Statedict*（映射），其中每个状态已删除了它的星号输入和星号保存

remove-asterisk-from-state(**mk-***StateD*(*speclist*,))(*dict*) ≙ (3.7.4.2)

```
1  (let (speclist', saveset, inputset, priinput) =
2       remove-asterisk-from-spec(speclist, {}, {}, {}, false)(dict) in
3   let (inputsignals, prisignals) =
4       (let qual = process-or-service-level(dict(SCOPEUNIT)) in
5        cases dict(qual):
6         (mk-ProcessD(, sig, , )      → (sig, {}),
7          mk-ServiceD(, , , sig, psig) → (sig, psig))) in
8   if priinput = prisignals ∧ (saveset ∪ inputset) ⊆ inputsignals then
9     (let (expimpinput, ) = import-export-signals(dict) in
10     let implicitinput = inputsignals \ saveset \ priinput \ inputset \ expimpinput in
11     let implicittrans = as0-implicit-transitions(implicitinput)(dict) in
12     mk-StateD(speclist' ⌢ implicittrans, nil))
13    else
14     exit("§2.6.3: 状态节点中的信号不是良形式的"))
```

type: *StateD* → *Dict* → *StateD*

目的

从一个状态删去星号输入和星号保存，并把未规定的信号的隐含跃迁包括在输入描述表 *Speclist* 中

参数

状态字典 *Statedict* 的描述符（*StateD*）包含

speclist　是输入描述表 *Speclist*，它包含此状态的 $Statespec_0$

结果

修改过的状态描述符 *StateD*

算法

第 1 行　构造修改过的输入描述表 *Speclist*，并抽取在保存中规定的信号、在输入中规定的信号和在优先级输入中规定的信号。

第 3—7 行　抽取完整的有效输入信号集合和进程或服务的优先级信号。

第 8 行　在状态的优先级输入中的信号必须与服务的优先级信号相同。在保存中规定的信号和在其它输入中的信号必须包括在完整的有效输入信号集合中。

第 9 行　令 *expimpinput* 表示在出口和进口速记中收到的隐含的 xtQUERY 和 xtREPLY 信号和 emptyq 信号。

第 10 行　需要为之构造隐含跃迁的信号就是从完整的有效输入信号集中减去下列信号后的信号。应减去的信号是：保存中的信号、高优先级信号、输入中的信号和隐含信号。

第 11 行　构造隐含跃迁。

第 12 行　构造修改过的状态描述符 *StateD*。

as_0-*implicit-transitions*(*signalqualset*)(*dict*) ≙ (3.7.4.3)

```
1  if signalqualset = {} then
2    ⟨⟩
3   else
4    (let signalqual ∈ signalqualset in
5     let inputvars = as0-inputvars({signalqual})(dict) in
6     let input = mk-Inputspec0(inputvars, nil,
7                               mk-Transition0(⟨⟩, mk-Termstmt0(nil, mk-Nextstate0(nil)))) in
8     (⟨dict(SCOPEUNIT)), input⟩ ⌢ as0-implicit-transitions(signalqualset \ {signalqual})(dict))
```

type: *Signalqual-set* → *Dict* → *Speclist*

目的　　构造一个状态的隐含跃迁

参数

signalqualset　　是信号集合，要为这些信号构造隐含跃迁

结果　　包含隐含跃迁的输入描述表 *Speclist*

算法

第 1 行　　当全部处理完时，返回输入的 *Speclist*。
第 4 行　　令 *Signalqual* 表示要处理的下一个信号。
第 5 行　　为此信号构造一个参数表。
第 6 行　　为此信号构造输入跃迁，该跃迁仅包含表示下一个状态的划线。
第 8 行　　构造一个输入描述符 *Spec*，并把它与其余信号的输入描述符 *Speclist* 汇合。

remove-asterisk-from-spec(*speclist*, *sigset*, *saveset*, *priset*, *asterisk*)(*dict*) $\triangleq$ (3.7.4.4)

```
1  (if speclist = ⟨⟩ then
2     (⟨⟩, sigset, saveset, priset)
3    else
4     (let (q, spec) = hd speclist in
5       cases spec:
6        (mk-Savespec0(sigl)
7           → if is-Starred0(sigl) then
8                if asterisk then
9                  exit("§4.7: 状态具有多于一个星号输入或保存")
10                 else
11                 (let (specrest, inputtot, savetot, pritot) =
12                       remove-asterisk-from-spec(tl speclist, sigset, saveset, priset, true)(dict) in
13                  let sigq = all-input-signals(dict(SCOPEUNIT))(dict) in
14                  let (expimpinput, ) = import-export-signals(dict) in
15                  let sigq' = sigq \ inputtot \ savetot \ pritot \ expimpinput in
16                  let as0siglist = ⟨as0-id(sig) | sig ∈ sigq'⟩ in
17                  let spec' = if as0siglist = ⟨⟩ then
18                                 ⟨⟩
19                              else
20                                 ⟨(q, mk-Savespec0(as0siglist))⟩ in
21                  (specrest ⌢ spec', inputtot, sigq' ∪ savetot, pritot))
22              else
23              (let sigq = transform-signallist(sigl)(dict) in
24               let (specrest, inputtot, savetot, pritot) =
25                     remove-asterisk-from-spec(tl speclist, sigset, saveset ∪ sigq, priset, asterisk)(dict) in
26               if sigq ∩ (sigset ∪ saveset ∪ priset) = {} then
27                  (specrest ⌢ ⟨hd speclist⟩, inputtot, savetot, pritot)
28                else
29                 exit("§2.6.3: 状态中的信号不是分离的")),
```

```
30      mk-Priinput0(inp, ),
31      mk-Inputspec0(inp, enab, trans)
32        → if is-Starred0(inp) then
33            if asterisk then
34              exit("§4.7: 状态具有多于一个星号输入或保存")
35             else
36              (let (specrest, inputtot, savetot, pritot) =
37                   remove-asterisk-from-spec(tl speclist, sigset, saveset, pritot, true)(dict) in
38               let sigq = all-input-signals(dict(SCOPEUNIT))(dict) in
39               let (expimpinput, ) = import-export-signals(dict) in
40               let sigq' = sigq \ inputtot \ savetot \ pritot \ expimpinput in
41               let inpvars = as0-inputvars(sigq')(dict) in
42               let spec' = if inpvars = ⟨⟩ then
43                                ⟨⟩
44                           else
45                                ⟨(q, mk-Inputspec0(inpvars, enab, trans))⟩ in
46              (specrest ⌢ spec', sigq' ∪ inputtot, savetot, pritot))
47           else
48            (let sigq = {signal-qual(id)(dict) | mk-Inputvars0(id, ) ∈ elems inp} in
49             let (sigset', priset') = if is-Priinput0(spec) then
50                                          (sigset, priset ∪ sigq)
51                                      else
52                                          (sigset ∪ sigq, priset) in
53             let (specrest, inputtot, savetot, pritot) =
54                 remove-asterisk-from-spec(tl speclist, sigset', saveset, priset', asterisk)(dict) in
55             if sigq ∩ (sigset ∪ saveset ∪ priset) = {} then
56              (specrest ⌢ ⟨hd speclist⟩, inputtot, savetot, pritot)
57             else
58              exit("§2.6.3: 状态中的信号不是分离的")),
59      T → remove-asterisk-from-spec(tl speclist, sigset, saveset, priset, asterisk)(dict))))
```

type: $Statespec_0^*$ *Signalqual*-set *Signalqual*-set *Signalqual*-set *Bool* → *Dict* →
$Statespec_0^*$ *Signalqual*-set *Signalqual*-set *Signalqual*-set

目的

从一个状态中删除星号输入和星号保存。除输入节点外，此函数也返回在输入节点中提及的信号集、在保存节点中提及的信号集和在优先级输入节点中提及的信号集。

参数

speclist	剩余的要被处理的输入和保存的表（本函数是递归的）
sigset	在一个输入节点中至今已被识别的信号的限定表 *Qual*
saveset	在一个保存节点中至今已被识别的信号的限定表 *Qual*
priset	在一个优先级输入节点中至今已被识别的信号的限定表 *Qual*
asterisk	是一个标志，用来表示至今是否已经识别到一个星号保存或星号输入

结果

至今所构造的输入和保存的修改表、完整的输入信号的 *Qual* 集合、完整的保存信号的 *Qual* 集合和在状态中收到的高优先级信号的集合

算法

第 1 行　当全部处理完时，返回完整的输入信号的 *Qual* 集合、完整的保存信号的 *Qual* 集合和高优先级信号集合。

第 2 行　分解状态的第一个 *Spec*。

第 6—27 行　　如果在第一个输入描述符 *Spec* 中的 *State* $spec_0$ 是一个保存结点，则

第 6 行　　如果在此保存节点中规定了一个星号，则

第 8 行　　如果已经遇到了一个星号保存或一个星号输入，则是一个错误。

第 11 行　　从其余的输入和保存节点中删除星号输入和保存。**true** 表示现在已经遇到了一个星号保存（或输入）。*inputtot* 和 *savetot* 是显式规定的输入信号和保存信号的完整集合。

第 13 行　　抽取包含在服务或进程的完整有效输入信号集合中的信号 *Qual*。

第 14—15 行　　在保存节点中规定的信号集合是完整有效的输入信号集（*sigq*）中减去下列信号后的信号集合：在输入节点中规定的信号 *inputtot*、在其它保存节点中规定的信号 *savetot*、优先级输入信号和在进口（或出口）中的隐含的信号。

第 16 行　　把信号的 *Qual* 的集合转换为一个 AS_0 的标识符表。

第 17 行　　如果要有保存的信号，就构造新的保存的 *Spec*。

第 21 行　　返回该 *Spec*，与其余的 *Spec* 汇合，返回完整的输入信号集合，返回完整的保存信号集合和返回高优先级信号集合。

第 23 行　　如果保存包含一个信号表，则变换此信号表以便获得规定的信号（*sigq*）。

第 24 行　　从其余的输入节点和保存节点删除星号输入和星号保存。在这些输入和保存节点中，更新至今所获得的保存信号集以包括这一保存节点的信号。

第 26 行　　在保存中的信号必须与该状态中其它信号不相交。

第 27 行　　把 *Spec* 增加到结果的 *Spec* 表中（并且不再改变它）。

第 31—59 行　　如果第一个输入描述符 *Spec* 中的 *Statespec*$_0$ 是一个输入节点，则

第 32 行　　如果在输入节点中规定了一个星号，则

第 33 行　　如果已经遇到了一个星号保存或星号输入，则是一个错误。

第 36 行　　从其余的输入和保存节点中删除星号输入和保存。**true** 表示现在已经遇到了一个星号输入（或保存）。

第 38 行　　抽取信号的 *Qual*，这些信号包含在服务和进程的完整的有效输入信号集中。

第 39—40 行　　在输入节点中要规定的信号集合是完整有效的输入信号集（*sigq*）中减去下列信号后的信号集合：在其它输入节点中规定的信号 *inputtot*、在保存结点中规定的信号（*savetot*）、优先级输入信号和在进口（或出口）中隐含的信号。

第 41 行　　把信号的 *Qual* 的集合转换为 *Inputvars*$_0$ 的表，其中接收这些值的变量被略去(**nil**)。

第 42 行　　如果有信号没有被显式地描述，构造新的输入 *Spec*。

第 46 行　　返回该 *Spec*，与其余的 *Spec* 汇合，返回完整的输入信号集合和完整的保留信号集合。

第 48 行　　如果输入包含由多个 *Inputvars*$_0$ 组成的表，则抽取它们所包含的信号的 *Qual*。

第 49 行　　如果此输入是优先级输入，则把信号加到优先级输入集（*priset*）中，否则把信号加到正常的信号集中（*sigset*）。

第 53 行　　从其余的输入和保存节点中删除星号输入和星号保存。

第 55 行　　输入中的信号集合必须与该状态中其它信号集合不相交。

第 56 行　　把 *Spec* 增加到结果的 *Spec* 表中（并且不再改变它）。

第 59 行　　如果该 *Statespec*$_0$ 是一个连续信号，则继续处理下一个 *Statespec*$_0$。

as$_0$-*inputvars*(*starinput*)(*dict*) ≙ (3.7.4.5)

```
 1  (if starinput = {} then
 2     ⟨⟩
 3   else
 4    (let qual ∈ starinput in
 5     let (nm, ) = qual[len qual] in
 6     let id = mk-Id0(⟨⟩, nm) in
 7     let t = cases dict(qual):
 8             (mk-SignalD(tli, , ) → tli,
 9              mk-TimerD(tli, ) → tli) in
10     let vnum = len t in
11     let inpu = mk-Inputvars0(id, ⟨nil | 1 ≤ i ≤ vnum⟩) in
12     ⟨inpu⟩ ⌢ as0-inputvars(starinput \ {qual})(dict)))
```

type: *Signalqual*-**set** → *Dict* → *Inputvars*$_0^*$

目的 为在一个星号输入节点中收到的信号或在隐含的跃迁中接收的信号构造 AS$_0$ 变量表

参数

starinput 进程的有效输入信号集合，其中的信号既不是在一输入节点中已经显式规定的，也不是在优先级输入节点中或保存节点中已经显式规定的

算法

第 1 行 当全部处理完时，什么也不返回（本函数是递归的）。
第 4 行 令 *qual* 表示此集合中的一个信号的 *Qual*。
第 5 行 抽取此信号的名字。
第 6 行 构造此信号的标识符。
第 7—9 行 令 *t* 表示此信号或定时器的类别表。
第 10 行 令 *vnum* 表示由此信号传递的值的个数。
第 11 行 构造 AS$_0$ 输入变量表，其中每个变量都是 **nil**（省略了），并且
第 12 行 把它和其余的那些信号的 AS$_0$ 输入变量表一起返回。

3.7.5 连续信号和允许条件的扩展

remove-cont-enable-from-statelist(*stateset*)(*dict*, *statedict*) ≙ (3.7.5.1)

```
1  (if stateset = {} then
2     statedict
3   else
4    (let state ∈ stateset in
5     let mk-StateD(speclist, ) = statedict(state) in
6     let statedict' = remove-cont-enable-from-state(speclist, state)(dict, statedict) in
7     remove-cont-enable-from-statelist(stateset \ {state})(dict, statedict')))
```

type: *Statename*$_0$-**set** → *Dict Statedict* → *Statedict*

目的 从状态字典 *Statedict* 删除所有的连续信号和允许条件，办法是用 *ContenablestateD* 描述符来替换适当的状态描述符 *StateD* 以及为隐含状态增加新的状态描述符 *StateD*。

参数

stateset 是状态名集合。在这些状态中，将删除连续信号和允许条件。初始时，这个集合包括所有的状态（本函数是递归的）。

结果 修改过的状态字典 *Statedict*

算法

第 1 行 当全部处理完时，返回修改过的状态字典 *Statedict*。

第 4—5 行 令 *Speclist* 表示剩余状态之一的 *Speclist*。

第 6 行 在那个状态中删除连续信号和允许条件。

第 7 行 在其余的状态中删除连续信号和允许条件。

remove-cont-enable-from-state(*speclist*, *stnm*)(*dict*, *statedict*) $\triangleq$ (3.7.5.2)

```
 1  (let enable = (∃(, sp) ∈ elems speclist)(is-Inputspec0(sp) ∧ s-Enabling0(sp) ≠ nil) in
 2   let cont = (∃(, sp) ∈ elems speclist)(is-Contspec0(sp)) in
 3   if ¬enable ∧ ¬cont then
 4     statedict
 5    else
 6     (let (emptyqid, formu1nm, formu2nm) = dict(GLOBALNAMES) in
 7      let formu1 = mk-Id0(⟨⟩, formu1nm),
 8          formu2 = mk-Id0(⟨⟩, formu2nm) in
 9      let ptrans = pretrans(emptyqid, formu1) in
10      let speclist' =
11          if cont
12            then expand-continuous(stnm, speclist, emptyqid, formu1, formu2)(dict)
13            else speclist in
14      if ¬enable then
15        (let newstnm = create-unique-name() in
16         let preterm = mk-Transition0(ptrans, mk-Nextstate0(newstnm)) in
17         statedict + [stnm     ↦ mk-ContenablestateD(preterm),
18                      newstnm ↦ mk-StateD(speclist', nil)])
19       else
20        (let emptyqinput =
21             if cont then
22               ⟨⟩
23              else
24               (let term = mk-Termstmt0(nil, mk-Nextstate0(stnm)) in
25                ⟨emptyqtrans(term, emptyqid, formu1, formu2)(dict)⟩) in
26         let (mk-Transition0(decision, ), statedict') =
27             expand-enable(speclist' ⌢ emptyqinput)(statedict) in
28         let preterm = mk-Transition0(ptrans ⌢ ⟨decision⟩, nil) in
29         statedict' + [stnm ↦ mk-ContenablestateD(preterm)])))
```

type: *Speclist Name*$_0$ → *Dict Statedict* → *Statedict*

目的 通过修改状态字典 *Statedict* 从一个状态中删除连续信号和允许条件

参数

speclist 包含此状态的输入跃迁的 *Speclist*

stnm 此状态的名字

结果　　修改过的状态字典 *Statedict*

算法

第 8—10 行　　如果在连续信号中省略了优先级，则不能规定其它连续信号。

第 1—4 行　　如果此状态既不包含允许条件也不包含连续信号，则不改变状态字典 *Statedict*。

第 6 行　　抽取空队列信号的 AS_0 标识符和与空队列信号一起使用的两个变量的 AS_0 名字。

第 7—8 行　　构造这两个变量的 AS_0 标识符。

第 9 行　　构造 AS_0 动作表，它更新“唯一的”变量（*formu*1）和输出空队列信号。

第 10—13 行　　通过修改 *Speclist*（*speclist*）来扩展连续信号。

第 14 行　　如果此状态不包含允许条件，则

第 15 行　　为此状态构造一个新名字（该状态包含连续信号）。

第 16 行　　构造一跃迁，该跃迁必须在每一个包含此状态的下一状态中进行解释。

第 17 行　　用两个描述符来更新状态字典 *Statedict*。第一个描述符用于下一状态节点的变换中。当下一状态节点包含名字 *stnm* 时，则用 *preterm* 替换该下一状态节点。第二个描述符在把包含连续信号的状态（*stnm*）变换到 AS_1 时使用。

第 20—29 行　　扩展允许条件。

第 20—25 行　　构造空队列输入，除非它已经包括在 *speclist'* 中（即，除非此状态包含连续信号）。当一个状态不含连续信号时，跟随空队列的跃迁可认为是较简单的，因为在那种情况中不存在判定节点（仅隐含回到原来状态的下一状态）。

第 26 行　　扩展允许条件。扩展的结果包括一个组合的判定动作（*decision*），它测试允许条件的表达式，还包括一个状态字典 *Statedict*，它被扩展以包括所涉及到的所有隐含状态的状态描述符 *StateD*。

第 28 行　　构造要在合适的下一状态节点中解释的跃迁（如第 16—17 行所示），并且

第 29 行　　用在状态字典 *Statedict* 中的 *ContenablestateD* 替换此状态的状态描述符 *StateD*（如第 16—17 行所示）。

expand-continuous(*statenm*, *speclist*, *emptyqid*, *formu*1, *formu*2)(*dict*) ≙　　**(3.7.5.3)**

```
1  if (∃(,stsp) ∈ elems speclist)(is-Contspec0(stsp) ∧ s-Priority0(stsp) = nil) ∧
2    card {(,stsp) ∈ elems speclist | is-Contspec0(stsp)} > 1 then
3    exit("§4.11: 如果省略优先级,则只能指定一个连续信号")
4    else
5    (let priomap = collect-priority-info(speclist, len speclist)(dict) in
6     let conttrans = build-cont-trans(speclist, statenm, priomap) in
7     let contterm = mk-Transition0(conttrans, nil) in
8     let normtrans = ⟨speclist[i] | i ∈ ind speclist ∧ ¬is-Contspec0(speclist[i])⟩ in
9     let emptyqinput = emptyqtrans(contterm, emptyqid, formu1, formu2)(dict) in
10    normtrans ⌢ ⟨emptyqinput⟩)
```

type:　$Statename_0\ Speclist\ Sigid_0\ Varid_0\ Varid_0 \to Dict \to Speclist$

目的　　修改包含连续信号的状态的输入跃迁

参数

statenm　　包含连续信号的状态名

speclist　　包含输入跃迁的 *Spec* 表

emptyqid　　空队列信号的 AS_0 标识符

*formu*1，*formu*2　　用于连续信号模型中的变量的 AS_0 标识符

结果　　修改过的 *Spec* 表

算法

第 5 行　　构造一个映射表，它包含一个信号的优先级和此信号在 *Speclist* 中的位置（标引号）之间的关系。

第 6—7 行　　构造组合判定动作，它按优先级给定的次序测试各种表达式。

第 8 行　　从 *speclist* 删除包含连续信号的输入跃迁。

第 9 行　　构造空队列输入跃迁，它测试是否为正确的空队列信号（即，测试 *formu*1 是否等于 *formu*2），并且该跃迁使用在“真”分枝中的组合判定（*contterm*）。

第 10 行　　把空队列输入跃迁加到不含连续信号的输入跃迁中。

pretrans(*emptyqid*, *formu*1*id*) $\triangleq$ **(3.7.5.4)**

```
1  (let lit1 = mk-Name0("1", nil),
2       self = mk-Selfexpr0() in
3   let infix = mk-Infixexpr0(formu1id, PLUS, lit1) in
4   let update = mk-Assignstmt0(formu1id, infix) in
5   let task = mk-Task0(⟨update⟩),
6       outtoself = mk-Output0(⟨mk-Outputsig0(emptyqid, ⟨formu1id⟩)⟩, self, ⟨⟩) in
7   ⟨mk-Actstmt0(nil, task), mk-Actstmt0(nil, outtoself)⟩)
```

type:　$Sigid_0\ Varid_0 \rightarrow Actstmt_0^+$

目的　　构造动作语句：

```
TASK    formu1id := formu1id + 1;
OUTPUT emptyqid(formu1id) TO SELF;
```

算法

第 1 行　　构造整数字面值“1”。

第 2 行　　构造 PiD 自身的表达式。

第 3 行　　构造 formu1id＋1。

第 4 行　　构造 formu1id：＝formu1id＋1。

第 5 行　　构造 TASK formu1id：＝formu1id＋1。

第 6 行　　构造 OUTPUT emptyqid(formu1id) TO SELF。

第 7 行　　返回这两个动作，作为动作语句（无标号）。

emptyqtrans(*conttrans*, *emptyqid*, *formu1id*, *formu2id*)(*dict*) ≙ **(3.7.5.5)**

```
1  (let emptyqinput = mk-Inputvars0(emptyqid, ⟨formu2id⟩) in
2   let ans1 = mk-Answer0(⟨mk-Condition0(nil, mk-Name0("TRUE", nil))⟩, conttrans),
3       ans2 = mk-Answer0(⟨mk-Condition0(nil, mk-Name0("FALSE", nil))⟩,
4                         mk-Termstmt0(nil, mk-Nextstate0(nil))) in
5   let decision = mk-Decision0(mk-Infixexpr0(formu1id, EQ, formu2id),
6                               ⟨ans1, ans2⟩, nil) in
7   let emptytrans = mk-Transition0(⟨mk-Actstmt0(nil, decision)⟩, nil) in
8   (dict(SCOPEUNIT), mk-Inputspec0(⟨emptyqinput⟩, nil, emptytrans)))
```

type: *Transition*$_0$ *Siqid*$_0$ *Varid*$_0$ *Varid*$_0$ → *Dict* → *Spec*

目的　　构造输入跃迁：

```
INPUT emptyqid(formu2id);
 DECISION (formu1id = formu2id)
  (true)  : conttrans;
  (false) : NEXTSTATE(-);
 ENDDECISION;
```

并返回含有输入跃迁的 *Spec*

参数

conttrans	是对应于“true”分枝的跃迁。如果此状态不含连续信号（但包含允许条件），则它包含一个回到原来状态的下一状态，否则它包含在函数 *build-cont-trans* 中构造的组合判定。在“false”分枝中的下一状态意味着在隐式状态（而不是原来的状态）下继续（处理），输入跃迁和该状态相关联。
emptyqid	空队列信号的 AS_0 标识符
formu1id, *formu2id*	用于连续信号和允许条件中的变量的 AS_0 标识符

build-cont-trans(*speclist*, *stnm*, *priomap*) ≙ **(3.7.5.6)**

```
1  (let i ∈ dom priomap be s.t. ¬(∃n ∈ dom priomap)(n < i) in
2   let trans = if priomap \ {i} = []
3               then mk-Transition0(⟨⟩, mk-Termstmt0(nil, mk-Nextstate0(stnm)))
4               else build-cont-trans(speclist, stnm, priomap \ {i}) in
5   let index = priomap(i) in
6   let (scope, mk-Contspec0(expr, , exittrans)) = speclist[index] in
7   let yesval = mk-Condition0(nil, mk-Id0(⟨⟩, mk-Name0("TRUE", nil))),
8       noval = mk-Condition0(nil, mk-Id0(⟨⟩, mk-Name0("FALSE", nil))) in
9   let yestrans = mk-Answer0(⟨yesval⟩, exittrans),
10      notrans = mk-Answer0(⟨noval⟩, trans) in
11  let decision = mk-Decision0(mk-Scopeexpr0(scope, expr), ⟨yestrans, notrans⟩, nil) in
12  mk-Transition0(⟨mk-Actstmt0(nil, decision)⟩, nil))
```

type: *Speclist* *Statename*$_0$ (N_1 ⇸ N_1) → *Transition*$_0$

目的　　构造组合判定，它按优先级给定的次序对连续信号中的表达式进行测试与分枝

参数

speclist	包含连续信号的 *Spec* 表
stnm	这些连续信号附属的（显式）状态的名字

priomap　　连续信号的优先级和它们在 *speclist* 中的位置（标引号）之间的关系

结果　　组合的判定动作语句

算法

第 1 行　　令 i 表示在优先级映射表中具有最低值的项目（即，在映射表 *priomap* 的剩余元素之中的最高优先级）。

第 2—4 行　　如果这个值具有最低优先级（即，如果它是映射表中仅剩下的），则令 *trans* 表示终端符语句，该语句包含的下一状态转向老的显式状态（第 3 行），否则令 *trans* 表示其余优先级（第 4 行）的判定动作。

第 5—6 行　　令 *scope* 表示定义此状态的作用域单位的限定表 *Qual*（请回忆，在进行这个变换之前，多个服务的状态字典 *Statedict* 已被合并，而且表达式的求值依赖于实际的服务）。令 *expr* 表示连续信号中的表达式，并令 *exittrans* 表示连续信号中的跃迁。

第 7—12 行　　构造包含下列判定的跃迁：

```
DECISION (expr):
 (true)  : exittrans;
 (false) : trans;
ENDDECISION
```

此处规定了 *expr* 将在由 *scope* 表示的作用域单位的范围内被变换到 AS_1。该 *scope* 可以是一个进程、一个过程或一个服务。

collect-priority-info(*speclist*, *index*)(*dict*) $\triangleq$　　(3.7.5.7)

```
 1  (if index = 0 then
 2     []
 3   else
 4    (let emap = collect-priority-info(speclist, index − 1)(dict) in
 5     let (scope, inp) = speclist[index] in
 6     let dict′ = [q ↦ dict(q) | q ∈ dom dict ∧ ¬is-SynD(dict(q))] + [SCOPEUNIT ↦ scope] in
 7      cases inp:
 8       (mk-Contspec0(, prio, )
 9          → (let prio′ = if prio = nil then 1 else eval-simple-expr(prio, "INTEGER")(dict′) in
10             if prio′ ∈ dom emap then
11               exit("§4.11: 几个连续信号具有相同的优先级")
12             else
13              emap + [prio′ ↦ index]),
14        T → emap)))
```

type:　$Speclist\ N_0 \to Dict \to (N_1 \xrightarrow{m} N_1)$

目的　　构造一个映射表，它把连续信号的优先级映射到包含此连续信号的 *Speclist* 中的 *Spec* 的标引值

参数

speclist　　包含连续信号的 *Speclist*

index　　是当前对 *speclist* 的标引值。初始时，标引值等于此表的长度。当标引值达到零时，递归终止。

结果　　构造的优先级映射表

算法

第 1 行　　当全部处理完时，返回空映射表。
第 4 行　　收集其余的连续信号的优先级。
第 5 行　　抽取定义连续信号的作用域单位的 *Qual* 和输入跃迁。
第 6 行　　用此作用域单位的 *Qual* 更新字典 *Dict*，使得优先级表达式可以在正确的范围中求值。还从字典 *Dict* 中排除同义词，因为在优先级构件中，规定一个同义词是非法的。
第 8—13 行　　如果该输入跃迁是一个连续信号，则确定该连续信号的优先级。如果省略优先级，则设置优先级为“1”，否则计算该简单表达式的值。
第 10—11 行　　这里不允许存在具有相同值的另一个连续信号。
第 13 行　　把此优先级加入到包含其余优先级的映射表中。
第 14 行　　如果输入跃迁不是一个连续信号，则返回包含其余连续信号的优先级的映射表。

***expand-enable*(*speclist*)(*statedict*)** $\triangleq$ **(3.7.5.8)**

```
1  (let enabset = {speclist[i] | i ∈ ind speclist ∧ is-Inputspec0(speclist[i]) ∧
2                                  s-Enabling0(s-Statespec0(speclist[i])) ≠ nil} in
3   let normtrans = ⟨speclist[i] | 1 ≤ i ≤ len speclist ∧ speclist[i] ∉ enabset⟩ in
4   build-enable-decision(normtrans, enabset)(statedict))
```

type : *Speclist* → *Statedict* → *Transition*$_0$ *Statedict*

目的　　构造被包围的判定动作，该动作模拟一个允许条件状态，并用所隐含的隐式状态来更新状态字典 *Statedict*

参数

speclist　　包含一个状态的输入跃迁的 *Speclist*

结果　　一个包含嵌套的判定动作的跃迁和更新过的状态字典 *Statedict*

算法

第 1—2 行　　令 *enabset* 表示包含允许条件的 *Spec* 的集合，并且
第 3 行　　令 *normtrans* 表示不包含允许条件的 *Spec* 的集合。
第 4 行　　构造判定动作和隐式状态。

build-enable-decision(*normtrans*, *enabset*)(*statedict*) ≙ (3.7.5.9)

```
1  (let falsevalue = mk-Condition0(nil, mk-Id0(⟨⟩, mk-Name0("FALSE", nil))) in
2   let truevalue = mk-Condition0(nil, mk-Id0(⟨⟩, mk-Name0("TRUE", nil))) in
3   let enab ∈ enabset in
4   let (scope, mk-Inputspec0(vl, expr, tr)) = enab in
5   let sigl = ⟨sigid | (∃varl ∈ elems vl)(s-Sigid0(varl) = sigid)⟩ in
6   let newtrans = normtrans ⌢ ⟨(scope, mk-Savespec0(sigl))⟩ in
7   let newtrans' = normtrans ⌢ ⟨(scope, mk-Inputspec0(vl, nil, tr))⟩ in
8   let (trans0, statedict') =
9       if enabset \ {enab} = {}
10        then build-implicit-state(newtrans)(statedict)
11        else build-enable-decision(newtrans, enabset \ {enab})(statedict) in
12  let (trans1, statedict'') =
13      if enabset \ {enab} = {}
14        then build-implicit-state(newtrans')(statedict')
15        else build-enable-decision(newtrans', enabset \ {enab})(statedict') in
16  let ans0 = mk-Answer0(falsevalue, trans0) in
17  let ans1 = mk-Answer0(truevalue, trans1) in
18  let decexpr = mk-Scopeexpr0(scope, expr) in
19  let decision = mk-Decision0(decexpr, ⟨ans0, ans1⟩, nil) in
20  let actstmt = mk-Actstmt0(nil, decision) in
21  (mk-Transition0(⟨actstmt⟩, nil), statedict''))
```

type: *Speclist Spec*-**set** → *Statedict* → *Transition*$_0$ *Statedict*

目的

参见函数 *expand-enable*

参数

normtrans 是一个 *Speclist*，它包含不带允许条件的输入跃迁。

enabset 是一个 *Spec* 集合，它包含带允许条件的输入跃迁。

结果

参见函数 *expand-enable*

算法

第 1 行 构造“false”情况的标号。

第 2 行 构造“true”情况的标号。

第 3 行 从 *Spec* 集合中抽取一个 *Spec*，该集合表示带有允许条件的输入跃迁。

第 4 行 分解此 *Spec*。

第 5 行 从输入节点抽取信号标识符。

第 6 行 构成一个新的 *Speclist*，它包含旧 *Speclist* 和一个 *Savespec*$_0$。

第 7 行 构成一个新的 *Speclist*，它包含旧的 *Speclist* 和一个 *Inputspec*$_0$。

第 8—11 行 为“false”情况构造一棵判定树。

第 10 行 如果 *enab* 是最后一个允许条件，则构造一个判定，引向下一状态 *Nextstate*$_0$。

第 11 行 如果 *enab* 不是最后一个允许条件，则构造一个判定，引向其它的判定。

第 12—15 行 为“true”情况构造一棵判定树。

第 16 行 构造“false”回答。

第 17 行 构造“true”回答。

第 18 行 构造判定表达式。

第 19 行 构造判定。

第 20 行　　构造动作语句。

第 21 行　　返回新的跃迁和更新的状态字典 *Statedict*。

build-implicit-state(*alltrans*)(*statedict*) ≜ **(3.7.5.10)**

```
1  (let stnm = create-unique-name() in
2   let trans = mk-Transition0(⟨⟩, mk-Termstmt0(nil, mk-Nextstate0(stnm))) in
3   (trans, statedict + [stnm ↦ mk-StateD(alltrans, nil)]))
```

type: *Speclist* → *Statedict* → *Transition*$_0$ *Statedict*

目的　　创建一个新的隐式状态，它带有完整的 *Speclist*，并且更新状态字典 *Statedict*

参数

alltrans　　完整的 *Speclist*

结果　　参见函数 *expand-enable*

算法

第 1 行　　为新的隐含状态创建一个 AS_0 名字。

第 2 行　　构造一个跃迁，它所含有的下一状态就是该隐式状态。

第 3 行　　返回此跃迁和更新的状态字典 *Statedict*。

3.7.6　状态和输入节点的变换

transform-statelist(*transformedstates*)(*dict*) ≜ **(3.7.6.1)**

```
1  (let statedict = dict(STATEDICT) in
2   if dom statedict = transformedstates then
3     ({}, dict)
4    else
5     (let stnm ∈ (dom statedict \ transformedstates) in
6      if is-ContenablestateD(statedict(stnm)) then
7        transform-statelist(transformedstates ∪ {stnm})(dict)
8       else
9        (let mk-StateD(speclist, importinf) = statedict(stnm) in
10        let (currentstate, impact) = if importinf = nil then (stnm, ) else importinf in
11        let (saveset1, inputset1, dict') = transform-statespecl(currentstate, speclist)(dict) in
12        let inputset1' =
13            if importinf = nil ∨ impact = nil
14              then inputset1
15              else insert-importact(impact, inputset1) in
16        let statenode = mk-State-node1(name-to-name1(stnm), saveset1, inputset1') in
17        let (bodyrest, dict'') = transform-statelist(transformedstates ∪ {stnm})(dict') in
18        ({statenode} ∪ bodyrest, dict'')))
```

type: $Statename_0$-**set** → *Dict* → $State\text{-}node_1$-**set** *Dict*

目的　　把一个进程或过程体的状态变换到 AS_1。通过考虑状态字典 *Statedict* 中的每一个状态来变换这些状态。隐含在服务、连续信号和允许条件中的隐式状态已经包含在要被变换的 AS_0 状态中。在变换过程中，由进口表达式隐含的新状态可被加到状态字典 *Statedict* 中。

参数

transformedstates 已经被变换的状态的 AS_0 名字的集合。当开始应用函数 *transform-statelist* 时，此集合是空的。

结果 AS_1 状态和一个更新的 *Dict*，由用于进程或过程的进口表达式中所隐含的隐式变量（**IMPLIED** 项）的信息来更新，还更新 *Dict* 以包括在进程或过程中用到的输出信号（*OUTSIGNALS* 项）。

算法

第 1 行 从字典 *Dict* 中抽取状态字典 *Statedict*。

第 2 行 当变换过的状态都是 *Statedict* 中的状态时，则返回。

第 5 行 令 *stnm* 表示一个尚未变换的状态。

第 6—7 行 如果此状态包含连续信号或允许条件，则该状态已被若干隐式状态所替换，因此该状态没有被变换。

第 9 行 分解该状态的 *Statedict* 描述符。如果 *importinf* 不是 **nil**，则该状态是一个进口状态（参见 *Importstateinf* 的定义）。

第 10 行 令 *currentstate* 表示该状态的名字，它将被插在输入跃迁的表示下一状态的划线的位置上。如果当前处理的状态是隐含于进口表达式中的隐式状态，则该状态名出现在状态描述符 *StateD* 中。

第 11 行 变换该状态的输入和保存。

第 12—14 行 如果该状态是隐含于进口中的隐式状态，则使用隐式进口变量的 AS_1 动作必须插入到所变换的状态中。在变换进口表达式时，这一动作没有插入，因为进口状态是在表达式的变换过程中构造的。

第 16 行 构造 AS_1 状态节点。

第 17 行 变换其余的状态。

第 18 行 把所构造的状态与其余的状态汇合到一起，并且返回更新过的 *Dict*。

transform-statespecl(*stnm*, *statespeclist*)(*dict*) ≙ **(3.7.6.2)**

```
1  (let (saveset, statespeclist') =
2      if (∃i ∈ ind statespeclist)(is-Savespec0(s-Statespec0(statespeclist[i]))) then
3        (let i be s.t. (is-Savespec0(s-Statespec0(statespeclist[i]))) in
4         let mk-Savespec0(sigl) = s-Statespec0(statespeclist[i]) in
5         let sigqualset = transform-signallist(sigl)(dict) in
6         (sigqualset, ⟨statespeclist[n] | 1 ≤ n ≤ len statespeclist ∧ n ≠ i⟩))
7       else
8        ({},statespeclist) in
9  let dict' = dict + [SCOPEUNIT ↦ process-level(dict(SCOPEUNIT))] in
10 let varset = {qual ∈ dom dict | is-VarD(dict(qual)) ∧ process-level(qual) = dict'(SCOPEUNIT)} in
11 let xtQUERYspec0 = as0-xtQUERY-inputs(varset, stnm)(dict) in
12 let (as1 nodeset, dict') = transform-input(stnm, statespeclist' ⌢ xtQUERYspec0)(dict) in
13 let saveset1 = mk-Save-signalset1(make-as1 idset(saveset)(dict)) in
14 (saveset1, as1 nodeset, dict'))
```

type: *Statename*$_0$ *Speclist* → *Dict* → *Save-signalset*$_1$ *Input-node*$_1$-**set** *Dict*

目的 把一个状态变换到 AS_1

参数

stnm 该状态的名字，它被插入到该状态的下一状态划线的位置上

statespeclist 一个 *Speclist*，它包含该状态的输入跃迁

结果 AS_1 保存结点和 AS_1 输入节点

算法

第 1—8 行　令 *saveset* 表示一个 *Signalqual* 的集合，它包含保存在此状态中的信号。又令 *statespeclist'* 表示修改过的 *Speclist*，其中的保存已被消去（如果有的话）。

第 2 行　如果在 *Speclist* 中有一个保存，则

第 3—4 行　从此 *Speclist* 中抽取该保存。

第 5 行　变换此保存中的信号。

第 6 行　从此 *Speclist* 中删除此保存。

第 9—10 行　更新 **SCOPEUNIT** 以表示周围的进程。*dict'* 仅用于抽取此进程的局部变量（第 10 行）。

第 11 行　为 xtQUERY 信号构造 AS_0 输入跃迁。

第 12 行　变换此输入跃迁，该处已增加了 xtQUERY 输入。

第 13 行　从 *Signalqual* 集合构造 AS_1 保存节点。

第 14 行　返回此保存节点和输入节点。

$as_0\text{-}xtQUERY\text{-}inputs(exportset, stnm)(dict) \triangleq$ **(3.7.6.3)**

```
1  (if exportset = {} then
2     ⟨⟩
3   else
4    (let qual ∈ exportset,
5         term = mk-Nextstate0(stnm),
6         mk-VarD(tq,,export,,) = dict(qual) in
7     if export = nil then
8       as0-xtQUERY-inputs(exportset \ {qual}, stnm)(dict)
9      else
10      (let (,nm) = qual[len qual],
11           mk-SignalnamesD(imp, xq, xr) = exportmap((tq, nm)) in
12       let sigelem = ⟨mk-Outputsig0(mk-Id0(⟨⟩, xr), ⟨mk-Id0(⟨⟩, imp))⟩⟩ in
13       let xroutput = mk-Output0(sigelem, mk-Senderexpr0(), ⟨⟩) in
14       let xrtrans = mk-Transition0(⟨mk-Actstmt0(nil, xroutput)⟩, term) in
15       let xqinput = mk-Inputspec0(⟨mk-Inputvars0(mk-Id0(⟨⟩, xq), ⟨⟩)⟩, nil, xrtrans) in
16       ⟨(dict(SCOPEUNIT), xqinput)⟩ ⌢ as0-xtQUERY-inputs(exportset \ {qual}, stnm)(dict))))
```

type: *Qual*-set $Name_0 \rightarrow Dict \rightarrow Speclist$

目的　为进程（与 Z.100 的 § 4.13 中所定义的一样）的所有隐含于出口中的信号创建 AS_0 输入节点

参数

exportset　在出口进程中定义的变量的限定表 *Qual* 的集合

stnm　将要附加隐式输入的状态的名字

结果　参见输入描述符表 *Speclist* 定义

算法

第 1 行　当全部处理完时，什么也不返回（本函数是递归的）。

第 4 行　令 *qual* 表示此集合中的一个变量的 *Qual*。

第 5 行　构造跟随隐式输入的 AS_0 的下一个状态节点。

第 6 行　分解此变量的描述符。

第 7 行　如果它不是一个出口变量，则继续考虑下一变量。

第 10 行　　抽取出口变量的名字。

第 11 行　　分解在具有名字（*nm*）和类别（*tq*）的系统中所有出口变量的公共的信号名描述符 *SignalnamesD*（参见 *Exportmap* 的定义）。*imp* 表示附加到出口变量的隐式变量的名字，*xq* 是 xtQUERY 信号的名字，*xr* 是 xtREPLY 信号的名字。

第 13 行　　创建发送 xtREPLY 信号的 AS_0 输出节点。

第 14 行　　创建直接跟随输入节点的跃迁。

第 15 行　　创建包含 xtQUERY 信号的输入节点。

第 16 行　　返回一个 *Spec*，它包含完整的输入跃迁（*xqinput*），并与其余的出口变量的 *Speclist* 汇合。

transform-input(*stnm*, *statespecl*)(*dict*) ≙ **(3.7.6.4)**

```
1  (if statespecl = ⟨⟩ then
2     ({},dict)
3    else
4     (let (scope, node) = hd statespecl in
5      let labeldict' =
6          if is-ServiceD(dict(scope))
7            then s-Labeldict(dict(scope))
8            else dict(LABELDICT) in
9      let dict' = dict + [SCOPEUNIT ↦ scope,
10                         LABELDICT ↦ labeldict'] in
11      cases node:
12       (mk-Priinput0(inputvars, trans),
13        mk-Inputspec0(inputvars, , trans)
14          → (let (as1 set', dict'') = transform-input-transition(stnm, inputvars, trans)(dict') in
15             let (as1 rest, dict''') = transform-input(stnm, tl statespecl)(dict'') in
16             (as1 set' ∪ as1 rest, dict''')))))
```

type: $Statename_0$ *Speclist* → *Dict* → $Input\text{-}node_1$-**set** *Dict*

目的　　变换一个状态的输入跃迁并用隐式变量的信息和用输出信号来更新 *dict*

参数

stnm　　此状态的名字

statespecl　　此状态的 *Speclist*

结果　　此状态的 AS_1 输入节点的集合

算法

第 1 行　　当全部处理完时，返回空集。

第 4 行　　分解要考虑的下一个 *Spec*。

第 5 行　　抽取此作用域单位的标号字典 *labeldict*。如果输入跃迁源于一个服务（在它们被合并之前），则从该服务描述符抽取它。

第 9 行　　更新字典 *Dict* 中的作用域单位 **SCOPEUNIT** 以表示输入跃迁的出发地的作用域单位，并且更新标号字典 **LABELDICT** 以表示此作用域单位的标号字典 *Labeldict*。

第 12—13 行　　分解当前的输入跃迁。

第 14 行　　为 AS_0 输入节点中的所有信号创建一个 AS_1 输入节点集合。

第 15 行　　变换其余的输入跃迁。

第 16 行　　把新创建的输入节点与其余的（AS_1 输入节点）合并。

transform-input-transition(*stnm*, *inpvars*, *trans*)(*dict*) ≜ **(3.7.6.5)**

```
1  (if inpvars = ⟨⟩ then
2     ({}, dict)
3   else
4    (let (vl1, sigq) = transform-inputvars(hd inpvars)(dict) in
5     let as1 sigid = make-as1-identifier(sigq)(dict) in
6     let (trans1, dict') = transform-transition(stnm, trans, ⟨⟩)(dict) in
7     let inputnode = mk-Input-node1(as1 sigid, vl1, trans1) in
8     let (inprest, dict'') = transform-input-transition(stnm, tl inpvars, trans)(dict') in
9     ({inputnode} ∪ inprest, dict'')))
```

type: $Statename_0\ Inputvars_0^+\ Transition_0 \rightarrow Dict \rightarrow Input\text{-}node_1\text{-}set\ Dict$

目的　　把一个输入跃迁变换到 AS_1

参数

stnm	状态的名字
inpvars	输入的由参数和信号组成的偶对的表
trans	跟随输入的跃迁

结果　　是 AS_1 输入节点的集合。输入节点的个数和 AS_0 输入节点中的偶对（信号和参数）数一样多。

算法

第 1 行　　当处理过所有的信号和参数偶对时，则返回空集。

第 4 行　　从 AS_0 信号和参数的第一个偶对中构造 AS_1 参数表（vl_1）和此信号的 *Qual*。

第 5 行　　构造此信号的 AS_1 标识符。

第 6 行　　变换此跃迁。

第 7 行　　为此信号构造 AS_1 输入节点。

第 8 行　　为其余的信号偶对构造 AS_1 输入节点。

第 9 行　　返回这些 AS_1 输入节点，与其余的 AS_1 输入节点汇合，并返回参数。

transform-inputvars(**mk**-*Inputvars*$_0$(*sigid*, *varl*))(*dict*) ≜ **(3.7.6.6)**

```
1  (let squal = signal-qual(sigid)(dict) in
2   let tquallist = cases dict(squal):
3                     (mk-SignalD(tquallist, , ) → tquallist,
4                      mk-TimerD(tquallist, )  → tquallist) in
5   if (len varl = len tquallist) then
6     (let vl = transform-inputvarlist(varl, tquallist)(dict) in
7      (vl, squal))
8    else
9     exit("§2.6.4: 输入节点中的参数的个数出错 "))
```

type: $Inputvars_0 \rightarrow Dict \rightarrow [Variable\text{-}identifier_1]^*\ Signalqual$

目的　　构造输入节点的 AS_1 变量表，并且抽取输入信号的 *Qual*

参数

sigid	AS_0 信号标识符
varl	AS_0 变量表

算法

第 1 行	令 *squal* 表示此信号的 *Qual*。
第 2—4 行	令 *tquallist* 表示此信号的参数类别表。
第 5 行	类别表的长度必须等于变量表的长度。
第 6—7 行	把变量表变换到 AS_1，并把它和信号的 *Qual* 一起返回。

transform-inputvarlist(*varl*, *sortlist*)(*dict*) ≜ **(3.7.6.7)**

```
1  (if varl = ⟨⟩ then
2     ⟨⟩
3   else
4    (let restvl = transform-inputvarlist(tl varl, tl sortlist)(dict) in
5     if hd varl = nil then
6        ⟨nil⟩ ⌢ restvl
7      else
8       (let vqual = get-visible-variable(hd varl, {get-parent(hd sortlist)(dict)}, false)(dict) in
9        let mk-VarD(,,,, nqual) = dict(vqual) in
10       let as1 id = make-as1-identifier(nqual)(dict) in
11       ⟨as1 id⟩ ⌢ restvl)))
```

type: $[Varid_0]^*\ Sortqual^* \to Dict \to [Variable\text{-}identifier_1]^*$

目的 构造将附加给一个输入节点的 AS_1 变量表

参数

varl	AS_0 变量表
sortlist	信号的类别表

算法

第 1 行	当全部处理完时，返回空表。
第 4 行	变换表尾。
第 5 行	如果省略了第一个变量，则返回一个 **nil** 实物。
第 8 行	通过查看上下文抽取第一个标识符的 *Qual*。
第 10 行	构造 AS_1 变量标识符。
第 11 行	返回此 AS_1 变量标识符，与表中其余部分的变量标识符汇合。

insert-importact(*impact*, *signalset*$_1$) ≜ **(3.7.6.8)**

```
1  (let mk-Input-node1(replyid, val, trans) ∈ signalset1 in
2   let mk-Transition1(nodes, term) = trans in
3   if is-Decision-node1(impact) then
4     {mk-Input-node1(replyid, val, mk-Transition1(nodes, impact))}
5    else
6     {mk-Input-node1(replyid, val, mk-Transition1(nodes ⌢ ⟨impact⟩, term))})
```

type: $(Graph\text{-}node_1 \mid Decision\text{-}node_1)\ Input\text{-}node_1\text{-}\mathbf{set} \to Input\text{-}node_1\text{-}\mathbf{set}$

目的

在包含 xtREPLY 信号的 AS_1 输入节点中插入一个 AS_1 动作。构造这个动作与包含进口表达式的一个表达式的变换相关。因为进口表达式被扩展到 AS_0 状态，要等到该状态被变换之后才能插入该动作。

参数

impact	将被插入的 AS_1 动作
$signalset_1$	仅由 xtREPLY 输入节点组成的 AS_1 输入结点集合

结果

修改过的 AS_1 输入节点集合，它包含 xtREPLY 输入节点

算法

第 1 行	分解 AS_1 输入节点。
第 2 行	分解所包含的跃迁。
第 3 行	如果要插入的节点是一个判定（在一个提问包含进口表达式的情况下），则
第 4 行	用此判定动作替换终端符。否则
第 6 行	把此动作附加到动作表中。

3.7.7 跃迁串的变换

transform-transition(*tnm*, mk-*Transition*$_0$(*actl*, *term*), *nodes*$_1$)(*dict*) ≜ (3.7.7.1)

```
 1  (if actl = ⟨⟩ then
 2     (let mk-Termstmt0(, t) = term in
 3      cases t:
 4      (mk-Nextstate0(stnm)
 5         → if tnm = nil ∧ stnm = nil
 6              then exit("§4.9: 在初始跃迁中的下一状态划线")
 7              else (let statedict = dict(STATEDICT) in
 8                    let stnm' = if stnm = nil then tnm else stnm in
 9                    let nextstate =
10                       if tnm = nil ∧ is-ServiceD(dict(dict(SCOPEUNIT))) ∧ stnm' ∉ dom statedict then
11                         (let (,servicelist) = dict(SERVICES) in
12                          extract-initial-state(servicelist, stnm', ⟨⟩)(dict))
13                        else
14                         if is-ServiceD(dict(dict(SCOPEUNIT))) ∧ stnm' ∉ dom statedict
15                           then extract-servicestate(tnm, stnm')(dict)
16                           else stnm' in
17                    if nextstate ∈ dom statedict then
18                      if is-ContenablestateD(statedict(nextstate)) then
19                        (let trans = s-Transition0(statedict(nextstate)) in
20                         transform-transition(tnm, trans, nodes1)(dict))
21                       else
22                        (mk-Transition1(nodes1, mk-Nextstate-node1(name-to-name1(nextstate))), dict)
23                     else
24                      exit("§2.6.7.2.1: 下一个状态中的名字必须表示一个已定义的状态")),
25      mk-Stop0()
26        → (mk-Transition1(nodes1, mk-Stop-node1()), dict),
27      mk-Return0()
28        → (mk-Transition1(nodes1, mk-Return-node1()), dict),
29      mk-Join0(l)
30        → (let labeldict = dict(LABELDICT) in
31           if l ∈ dom labeldict then
32             (let trans = labeldict(l) in
33              transform-transition(tnm, trans, nodes1)(dict))
34            else
35             exit("§2.6.7.2.2: 在汇合中的标号未被定义"))))
36   else
37    (let (tran1, dict') = transform-act(tnm, actl, term, nodes1)(dict) in
38     cases tran1:
39      (mk-Transition1(,)
40         → (tran1, dict'),
41       T → transform-transition(tnm, mk-Transition0(tl actl, term), nodes1 ⌢ tran1)(dict'))))
```

type: [*Statename*$_0$] *Transition*$_0$ *Graph-node*$_1$* → *Dict* → *Transition*$_1$ *Dict*

目的

把一个跃迁变换到 AS$_1$

参数

tnm 是包含该跃迁的状态的名字。如果 *tnm* 是 **nil**，则该跃迁是一个初始跃迁。

actl 该跃迁的动作表

term 该跃迁的终端符

*nodes*1 已经被变换的节点来自跃迁的）

结果

AS$_1$ 的跃迁

算法

第 1—35 行　当处理完该动作表时，则变换终端符。

第 4 行　如果终端符是下一状态，且如果它包含在初始跃迁中，那么它就不能包含一个划线。

第 7 行　令 *statedict* 表示此进程的状态字典 *Statedict*。

第 8 行　令 *stnm'* 表示下一个状态中的名字，如果它被指定的话，否则它表示导向下一状态的状态的名字。

第 9—16 行　令 *nextstate* 表示下一状态的名字。

第 9—12 行　如果它是初始跃迁，作用域单位是一个服务，且它不是导向一个进口状态的下一状态，则通过考虑所有服务的初始终端符中的下一状态节点来抽取下一状态的状态名。

第 14 行　如果作用域单位是一个服务，且此状态名不出现在状态字典 *Statedict* 中（因为服务中的显式状态已被合并，并且在 *statedict* 中的每个新状态都有一个不同的新名字），则通过考虑旧的服务状态名和新的服务状态名来抽取这个不同的新状态。

第 17 行　必须要规定下一状态的名字。

第 18—20 行　如果下一状态导向一个已被许多允许条件状态所替换的状态，则状态字典 *Statedict* 仅含有一个跃迁，并且这个跃迁替换了这个下一状态节点。

第 22 行　构造 AS_1 变迁，它包含所变换的动作（$nodes_1$）和下一状态节点。

第 25 行　如果终端符是一个停止（stop），则构造包含一个停止节点的 AS_1 跃迁。

第 27 行　如果终端符是一个返回，则构造包含一个返回节点的 AS_1 跃迁。

第 29 行　如果终端符是一个汇接（join），则令 *labeldict* 表示此作用域单位的标号字典 *Labeldict*。

第 31 行　必须定义一个标号。

第 32 行　抽取所定义标号处的跃迁。

第 33 行　用此跃迁替换该汇接（join）。

第 37 行　如果在此跃迁中有更多的动作，则变换动作表中的第一个动作。请注意，虽然仅变换了第一个动作，但给出了全部的动作表作为参数。

第 39 行　如果变换第一个动作的结果是一完整的跃迁，则第一个动作包含进口表达式并且返回该跃迁。

第 41 行　否则变换其余的动作。

extract-initial-state(*servicelist*, *nextstate*, *statelist*)(*dict*) ≙ (3.7.7.2)

```
1  (if servicelist = ⟨⟩ then
2     (let (statetuplemap, ) = dict(SERVICES) in
3      if statelist ∈ dom statetuplemap then
4         statetuplemap(statelist)
5        else
6         exit("§2.6.7.2.1: 下一状态中的名字必须表示一个已定义的状态 "))
7    else
8     if hd servicelist = dict(SCOPEUNIT) then
9        extract-initial-state(tl servicelist, ..., statelist ⌢ ⟨nextstate⟩)(dict)
10       else
11        (let mk-ServiceD(trans, , , , ) = dict(hd servicelist) in
12         let mk-Transition0(, mk-Termstmt0(, mk-Nextstate0(stnm))) = trans in
13         extract-initial-state(tl servicelist, nextstate, statelist ⌢ ⟨stnm⟩)(dict)))
```

type: *Servicetuple Statename*$_0$ *Statename*$_0$* → *Dict* → *Statename*$_0$

目的　　在下列情况下抽取第一个（组合的）状态：这些状态包含合并的服务状态，而其中的一个服务包含一个初始跃迁。

参数

servicelist	服务的 *Qual* 表，它和状态名元组（*statelist*）相匹配
nextstate	下一服务状态的名字
statelist	状态名元组

结果　　导出的（唯一的）状态名

算法

第 1 行	当已经考虑所有服务的初始的下一个状态后，则
第 2 行	令 *statetuplemap* 表示状态名元组和相关的各个不同的名字之间的关系。
第 3 行	构造的下一状态的状态名元组必须包括在 *statetuplemap* 中，否则就有一个下一状态含有一个未定义的状态名。
第 4 行	返回与服务的各个下一状态对应的各个不同的状态名。
第 8—9 行	如果要考虑的服务就是当前的这个服务，则在考虑其余服务之前把下一状态名加到状态名元组里。在当前服务的下一状态里的名字不再由此函数所用（因此“…”）。
第 11 行	令 *trans* 表示要考虑的服务的初始跃迁（它有一个空动作表）。
第 12 行	令 *stnm* 表示在此状态的初始下一状态节点中的状态名。
第 13 行	在考虑其余的服务之前，把此状态名加到状态名元组中。

extract-servicestate(*oldstate*, *stnm*)(*dict*) $\triangleq$ **(3.7.7.3)**

```
 1  (let (statetuplemap, servicelist) = dict(SERVICES) in
 2   let statelist be s.t. statelist ∈ dom statetuplemap ∧ statetuplemap(statelist) = oldstate in
 3   let index be s.t. servicelist[index] = dict(SCOPEUNIT) in
 4   let newstatelist = ⟨statelist[i] | 1 ≤ i < index⟩ ⌢
 5       ⟨stnm⟩ ⌢
 6       ⟨statelist[i] | index < i ≤ len statelist⟩ in
 7   if newstatelist ∈ dom statetuplemap then
 8     statetuplemap(newstatelist)
 9    else
10     exit("§2.6.7.2.1: 下一状态中的名字必须表示一个已定义的状态"))
```

type: $Statename_0\ Statename_0 \rightarrow Dict \rightarrow Statename_0$

目的　　从一个旧的（但唯一）状态名和一个新的服务状态名抽取在状态字典 *Statedict* 中定义的一个唯一的状态的名字

算法

第 1 行	令 *statetuplemap* 表示状态名元组和各个不同的状态名之间的关系。令 *servicelist* 表示与状态名元组相匹配的服务的 *Qual* 表。
第 2 行	令 *statelist* 表示与旧状态名对应的状态名元组。
第 3 行	令 *index* 表示对当前服务的标引号。

第 4—6 行　　在按标引号所指的位置上用新状态名改写状态名元组。

第 7—8 行　　如果这个状态名元组有一唯一的状态名，则返回该唯一名字，否则新的服务状态名没有确定。

3.7.8　动作语句的变换

本小节说明动作语句到 AS_1 的变换。对每个变换函数，给出了下列项目：包含某个动作语句的状态的名字，跟随该动作语句的那些动作语句，跟随其余动作语句的终端符和已经被变换的那些动作语句（参见函数 *transform-act*）。动作语句中的进口表达式由 AS_1 动作语句中的隐式变量来表示。如果此动作语句包含任何进口表达式，则返回一个完整的跃迁 $Transition_1$，它包含先前已变换的节点和导向隐式状态的下一状态。并且更新状态字典 *Statedict* 以包括含有隐式状态信息的状态描述符 *StateD*、跟随此动作语句的那些动作语句、跟随那些动作语句的终端符、跟随那些动作语句的状态的名字和变换的节点（参见 *StateD* 的定义，尤其是 *Importstateinf* 的定义）。

transform-act$(tnm, actl, term, nodes_1)(dict)$ $\triangleq$ **(3.7.8.1)**

```
1  (let mk-Actstmt0(, act) = hd actl in
2   cases act:
3    (mk-Task0(stmtl)  → transform-stmtlist(tnm, stmtl, tl actl, term, nodes1)(dict),
4     mk-Output0(,,)   → transform-output(tnm, act, tl actl, term, nodes1)(dict),
5     mk-Prioutput0()  → transform-prioutput(tnm, act, tl actl, term, nodes1)(dict),
6     mk-Create0(,)    → transform-create(tnm, act, tl actl, term, nodes1)(dict),
7     mk-Decision0(,,) → transform-decision(tnm, act, nodes1)(dict),
8     mk-Option0(,,)   → transform-option(tnm, act, nodes1)(dict),
9     mk-Reset0()      → transform-reset(tnm, act, tl actl, term, nodes1)(dict),
10    mk-Set0()        → transform-set(tnm, act, tl actl, term, nodes1)(dict),
11    mk-Export0(idl)  → (⟨transform-export(idl[i])(dict) | 1 ≤ i ≤ len idl⟩, dict),
12    mk-Call0(,)      → transform-call(tnm, act, tl actl, term, nodes1)(dict)))
```

type：　$[Statename_0]\ Actstmt_0^+\ [Termstmt_0]\ Graph\text{-}node_1^* \to Dict \to (Transition_1 \mid Graph\text{-}node_1^+)\ Dict$

目的　　把一个动作语句变换到 AS_1

参数

tnm　　是动作语句所属的状态的名字。该名字用于替换表示下一状态的划线。

actl　　是动作语句表。该表中的第一个动作语句是要被变换的，如果动作语句包含进口表达式，则仅使用其余的动作语句。

term　　是跟随动作语句表的终端符。如果动作语句包含进口表达式，则仅使用该终端符。

*nodes*1　　先前的已经被变换的节点

结果　　如果此动作语句不包含进口表达式，则结果为一个图节点 $Graph\text{-}node_1$，否则结果是一个跃迁 $Transition_1$，此跃迁终止于隐式进口状态。更新字典 *Dict* 以包括输出信号和隐式变量的信息。

算法

第 1—2 行　　变换一个动作（*act*），它或者是

第 3 行　　一个任务。

第 4 行　　或是一个输出。
第 5 行　　或是一个优先级输出。
第 6 行　　或是一个创建请求。
第 7 行　　或是一个判定。
第 8 行　　或是一个任选。
第 9 行　　或是一个定时器复位。
第 10 行　　或是一个定时器设置。
第 11 行　　或是一个出口。
第 12 行　　或是一个过程调用。
请注意，不再使用 $Actstmt_0$ 中的标号，因为有关它的信息已包含在 *Labeldict* 中。

$transform\text{-}stmtlist(tnm, stmtl, actl, term, nodes_1)(dict) \triangleq$ **(3.7.8.2)**

```
1  (if stmtl = ⟨⟩ then
2     (nodes1, dict)
3   else
4     (let (as1 task, d) = cases hd stmtl:
5                            (mk-Assignstmt0(,)
6                               → transform-assign(hd stmtl)(dict),
7                             mk-Text0(text)
8                               → (transform-informal(text), dict)) in
9      if d(IMPORTLIST) = ⟨⟩ then
10       transform-stmtlist(tnm, tl stmtl, actl, term, nodes1 ⌢ as1 task)(d)
11     else
12      (let rest = if tl stmtl = ⟨⟩ then ⟨⟩ else ⟨mk-Task0(tl stmtl)⟩ in
13       let (outputtrans, d') = transform-import(tnm, as1 task, rest ⌢ actl, term)(d) in
14       transform-transition(tnm, outputtrans, nodes1)(d'))))
```

type: $[Statename_0]\ (Assignstmt_0^* \mid Text_0^*)\ Actstmt_0^*\ [Termstmt_0]\ Graph\text{-}node_1^* \rightarrow Dict \rightarrow (Transition_1 \mid Graph\text{-}node_1^+)\ Dict$

目的　　把一个语句表变换到 AS_1

参数

stmtl　　要变换的语句表
tnm,
actl,
term,
$nodes_1$　　参见函数 *transform-act* 及本小节的引言

结果　　参见函数 *transform-act* 及本小节的引言

算法

第 1 行　　当全部处理完时，返回 AS_1 任务节点表，其中每个节点包含一个语句，并返回更新的 *Dict*。
第 4—7 行　　令 $as_1 task$ 表示此表中第一个语句的任务。
第 5 行　　如果第一个语句是赋值语句，则变换它。
第 7 行　　如果第一个语句是非形式正文，则变换它并维持字典 *Dict* 不变。
第 9—10 行　　如果此语句中无进口表达式，则变换其余的语句，否则
第 12 行　　构造一个包含其余语句的任务（如果有的话）。
第 13 行　　更新状态字典 *Statedict* 以包括隐式状态的信息，并且构造一个跃迁，该跃迁包含先前已变换过的动作语句、xtQUERY 信号的输出和引向此隐式状态的下一状态。

第 14 行　　变换所构造的跃迁。

transform-informal(*text*) ≙ (3.7.8.3)

```
1  (let mk-Text0(t) = text in
2   mk-Task-node1(mk-Informal-text1(t)))
```

type: $Text_0 \rightarrow Informal\text{-}text_1$

目的　　把非形式正文变换为它的 AS_1 表示法

transform-assign(mk-*Assignstmt*$_0$(*vid*, *expr*))(*dict*) ≙ (3.7.8.4)

```
1  (if is-Id0(vid) then
2     (let sortset = all-visible-sorts(dict) in
3      let (,sortset',) = transform-expr(expr, EXPRESSION, sortset)(dict) in
4      let qual = get-visible-variable(vid, sortset', false)(dict) in
5      let mk-VarD(q,,,,vqual) = dict(qual) in
6      let (as1tree,,d) = transform-expr(expr, EXPRESSION, {get-parent(q)(dict)})(dict) in
7      let as1id = make-as1-identifier(vqual)(dict) in
8      let as1assign = mk-Assignment-statement1(as1id, as1tree) in
9      (mk-Task-node1(as1assign), d))
10    else
11     transform-modifyassign(vid, expr)(dict))
```

type: $Assignstmt_0 \rightarrow Dict \rightarrow Task\text{-}node_1\ Dict$

目的　　把一个赋值语句变换到 AS_1

参数　　该赋值语句包括

vid　　要赋值的变量

expr　　右边的表达式

结果　　AS_1 赋值语句和更新的 *Dict*，它包括 *expr* 中的进口表达式和隐式变量的信息

算法

第 1 行　　如果变量是标识符，则

第 2—4 行　　构造它的 *Qual*，办法是抽取与表达式相匹配的类别（第 3 行）并在按上下文求解右边（表达式）时，使用这些类别（第 4 行）。

第 5 行　　分解变量描述符。令 *q* 表示它的类别的 *qual*，并令 *vqual* 表示它的唯一限定表 *Qual*（在不同的服务中是不同的）。

第 6 行　　变换右边表达式，该式的类别必须是 *q*。

第 7 行　　从它的唯一的限定表 *Qual* 构造 AS_1 标识符。

第 8 行　　构造 AS_1 赋值语句。

第 9 行　　构造并返回包含此赋值语句的任务。

第 11 行　　如果此变量不是一个标识符，则该赋值语句是 **MODIFY**！或字段修改运算符的一个速记符。

transform-modifyassign(*vid*, *expr*)(*dict*) ≙ (3.7.8.5)

```
 1  (let makeexpr(var) =
 2      if is-Indexedvar0(var) then
 3        mk-Operatorapp0(makeexpr(s-Variable0(var)), s-Exprlist0(var))
 4       else
 5        if is-Fieldvar0(var) then
 6          mk-Selectexpr0(makeexpr(s-Expr0(var)), s-Name0(var))
 7         else
 8          var in
 9   cases vid:
10    (mk-Indexedvar0(vid', exprlist)
11       → (let opid = mk-Id0(⟨⟩, mk-Name0("MODIFY", EXCLAMATION)) in
12          let vid'' = makeexpr(vid') in
13          let righthandside = mk-Operatorapp0(opid, ⟨vid''⟩ ⌢ exprlist ⌢ ⟨expr⟩) in
14          let (as1 node, dict') =
15              (trap exit with (nil, dict) in
16               transform-assign(mk-Assignstmt0(vid', righthandside))(dict)) in
17          let (as1 node', dict'') =
18              if is-Id0(exprlist[len exprlist]) ∧
19               s-Qualifier0(exprlist[len exprlist]) = ⟨⟩ then
20                (let field = s-Name0(exprlist[len exprlist]) in
21                 let vid'' = if len exprlist = 1 then
22                                 vid'
23                             else
24                                 mk-Indexedvar0(vid', ⟨exprlist[i] | 1 ≤ i ≤ len exprlist − 1⟩) in
25                 let vid''' = mk-Fieldvar0(vid'', field) in
26                 trap exit with (nil, dict) in
27                 transform-assign(mk-Assignstmt0(vid''', expr))(dict))
28               else
29                (nil, dict) in
30          (as1 node = nil ∧ as1 node' ≠ nil
31             → (as1 node', dict''),
32           as1 node ≠ nil ∧ as1 node' = nil
33             → (as1 node, dict'),
34           T → exit("§5.5.3: 修改的赋值句没有扩展或有多重扩展 "))),
35     mk-Fieldvar0(vid', mk-Name0(str, ))
36       → (let opid = mk-Id0(⟨⟩, mk-Name0(str ⌢ "MODIFY", EXCLAMATION)) in
37          let righthandside = mk-Operatorapp0(opid, ⟨vid', expr⟩) in
38          transform-assign(mk-Assignstmt0(vid', righthandside))(dict))))
```

type: $Variable_0\ Expr_0 \to Dict \to Task\text{-}node_1\ Dict$

目的

把一个被标引变量或一个字段变量的赋值语句变换到 AS_1

参数

vid 左边的变量

expr 右边的表达式

结果

包含赋值语句速记符的 AS_1 表示法的任务结点

算法

第 1—8 行 构造一个递归函数，该函数把一个被标引的左边转换到一个被标引的右边（即一个运算符的应用），并把一个字段的左边转换到一个字段的右边（一个 $Selectexpr_0$）。

第 10—16 行 扩展一个被标引变量赋值语句的速记符。例如

v(e1)(e2, e3):=e4

被扩展为

v(e1):=MODIFY!(v(e1),e2,e3,e4)

该式在被变换到 AS_1 前，经过函数 *transform-assign* 处理被扩展为

v:=MODIFY!(v, e1, MODIFY!(v(e1), e2, e3, e4))

如果变换失败，可以捕捉到错误，因为该赋值语句可能是其它速记符号（构造于第17—29）。

第 11 行　构造 MODIFY！运算符的标识符，并且

第 12 行　令 *vid''* 表示转换的左边。

第 13 行　构造新的右边。

第 16 行　变换新的赋值语句。

第 17—29 行　扩展字段选择赋值语句的括号的形式。例如

v(e1)(e2,e3):=e4

被扩展为

v(e1)(e2)!e3:=e4

(如果 e3 是一个非限定标识符的话)。这个构造的字段变量 $Fieldvar_0$ 被变换到 AS_1。如果变换失败，可以捕捉到错误，因为该赋值语句可能是其它速记符号(构造于第 11—16 行)。

第 17—19 行　除非表达式表中最后一个表达式是一个非限定标识符，否则该速记符是不可能的。

第 20 行　抽取字段的名字。

第 21—24 行　如果表达式表中述有表达式，则构造另一个标引变量 $Indexedvar_0$，否则返回变量 $Variable_0$ (*vid'*)。

第 25 行　构造字段选择构件。

第 26—27 行　试变换字段选择赋值语句。

第 30—34 行　必须是两种可能的良形式扩展中的一种，且只能是一种。

第 35—38 行　扩展一个字段选择赋值语句，例如

a!b!c:=e

被扩展为

a!b:=cMODIFY!(a!b,e)

该式在它被变换到 AS_1 之前，经过函数 *transform-assign* 处理被扩展为

a:=bMODIFY!(a, cMODIFY!(a!b, e))

第 36 行　构造修改运算符的标识符，该运算符可修改带有拼字 *str* 的字段。

第 37 行　构造新的右边。

第 38 行　变换新的赋值语句。

transform-reset(*tnm*, *resetnode*, *actl*, *term*, $nodes_1$)(*dict*) ≙ **(3.7.8.6)**

```
1  (let mk-Reset0(resetlist) = resetnode in
2   let (actl1, d) = transform-reset-elems(tnm, resetlist, actl, term, nodes1)(dict) in
3   if tl resetlist = ⟨⟩ ∨ is-Transition1(actl1) then
4     (actl1, d)
5    else
6     transform-reset(tnm, mk-Reset0(tl resetlist), actl, term, nodes1 ⌢ actl1)(d))
```

type: [$Statename_0$] $Reset_0$ $Actstmt_0$* [$Termstmt_0$] $Graph\text{-}node_1$* →
Dict → (*Transition*$_1$ | $Graph\text{-}node_1$*) *Dict*

目的　　把复位动作变换到 AS_1

参数

tnm,
actl,
term,
$nodes_1$　　参见变换动作函数 *transform-act*
resetnode　　要变换的复位动作

结果　　参见变换动作函数 *transform-act*

算法

第 1 行　　令 *resetlist* 表示要复位的定时器表。
第 2 行　　变换第一个要被复位的定时器。
第 3 行　　如果该定时器复位是表中最后的一个（本函数是递归的）或者如果返回一个跃迁（因为表达式表中的进口出口表达式），则停止变换（如果表达式表包含进口表达式，则在变换隐式进口状态时，变换其余的定时器）。
第 6 行　　否则变换其余的定时器复位。

transform-reset-elems(*tnm*, *resetlist*, *actl*, *term*, $nodes_1$)(*dict*) ≙ **(3.7.8.7)**

```
1  (let mk-Resetelem0(timerid, actparm) = hd resetlist in
2   let squal = get-visible-qual(timerid, SIGNAL)(dict) in
3   if is-SignalD(dict(squal)) then
4     exit("§2.8: 在复位动作中的标识符不是一个定时器")
5    else
6     (let mk-TimerD(tplist, ) = dict(squal) in
7      if len actparm = len tplist then
8        (let (as1 actparm, d) = transform-actparms(tplist, actparm, EXPRESSION)(dict) in
9         let as1 id = make-as1-identifier(squal)(dict) in
10        let as1 tree = mk-Reset-node1(as1 id, as1 actparm) in
11        if d(IMPORTLIST) = ⟨⟩ then
12          (⟨as1 tree⟩, d)
13         else
14          (let rest = if tl resetlist = ⟨⟩ then ⟨⟩ else ⟨mk-Reset0(tl resetlist)⟩ in
15           let (outputtrans, d') = transform-import(tnm, as1 tree, rest ⌢ actl, term)(d) in
16           transform-transition(tnm, outputtrans, nodes1)(d')))
17       else
18        exit("§2.8: 在复位动作中参数表的长度必须等于定义中规定的长度")))
```

type: [$Statename_0$] $Resetelem_0$* $Actstmt_0$* [$Termstmt_0$] $Graph\text{-}node_1$* →
Dict → (*Transition*$_1$ | $Graph\text{-}node_1$*) *Dict*

目的　　把一个定时器复位变换到 AS_1

参数　　参见函数 *transform-reset*

结果　　参见函数 *transform-reset*

算法

第 1 行　　分解 21 条要变换的 $Resetelem_0$。
第 2 行　　抽取规定的标识符的 *Qual*。
第 3 行　　此标识符必须表示一个定时器信号。
第 6 行　　令 *tplist* 表示为定时器规定的表达式表的类别。
第 7 行　　在表达式表中的表达式的个数必须和定时器的类别数目相同。
第 8 行　　变换这些表达式。
第 9 行　　构造定时器的 AS_1 标识符。
第 10 行　　构造 AS_1 复位节点。
第 11 行　　如果在表达式表中无进口表达式，则
第 12 行　　返回 AS_1 复位节点，否则
第 14 行　　为其余的定时器的复位构造一个 AS_0 复位动作。该动作将附属于隐含进口状态。
第 15 行　　用进口状态来更新 *Dict*，并构造引向进口状态的 AS_0 跃迁。
第 16 行　　变换该跃迁。

$$transform\text{-}set(tnm, setnode, actl, term, nodes_1)(dict) \triangleq \qquad (3.7.8.8)$$

```
1  (let mk-Set0(setlist) = setnode in
2   let (actl1, d) = transform-set-elems(tnm, setlist, actl, term, nodes1)(dict) in
3   if tl setlist = ⟨⟩ ∨ is-Transition1(actl1) then
4     (actl1, d)
5    else
6     transform-set(tnm, mk-Set0(tl setlist), actl, term, nodes1 ↷ actl1)(d))
```

type: $[Statename_0]\ Set_0\ Actstmt_0^*\ [Termstmt_0]\ Graph\text{-}node_1^* \rightarrow Dict \rightarrow (Transition_1 \mid Graph\text{-}node_1^*)\ Dict$

目的　　把一个设置动作变换到 AS_1

参数

tnm,
actl,
term,
$nodes_1$　　参见函数 *transform-act*
setnode　　要变换的设置动作

结果　　参见函数 *transform-act*

算法

第 1 行　　令 *setlist* 表示要设置的定时器表。
第 2 行　　变换第一个要设置的定时器。
第 3 行　　如果该定时器的设置是表中的最后一个（此函数是递归的），或者如果返回一个跃迁（由于表达式表中的进口出口表达式），则停止变换（如果表达式表中包含进口表达式，则当变换隐式进口状态时，将变换其余的定时器）。

第 6 行　　否则变换其余的定时器设置。

transform-set-elems(*tnm*, *setlist*, *actl*, *term*, *nodes*$_1$)(*dict*) ≙ (3.7.8.9)

```
1  (let mk-Setelem0(timeexpr, timerid, actparm) = hd setlist in
2   let timesort = get-predef-sort("TIME")(dict) in
3   let (as1 expr, d) = transform-expr(timeexpr, EXPRESSION, {timesort})(dict) in
4   let squal = get-visible-qual(timerid, SIGNAL)(dict) in
5   if is-SignalD(dict(squal)) then
6     exit("§2.8: 在设置动作中的标识符不是一个定时器")
7    else
8     (let mk-TimerD(tplist, ) = dict(squal) in
9      if len actparm = len tplist then
10       (let (as1 actparm, d') = transform-actparms(tplist, actparm, EXPRESSION)(d) in
11        let as1 id = make-as1-identifier(squal)(dict) in
12        let as1 tree = mk-Set-node1(as1 expr, as1 id, as1 actparm) in
13        if d'(IMPORTLIST) = ⟨⟩ then
14          (⟨as1 tree⟩, d')
15         else
16          (let rest = if tl setlist = ⟨⟩ then ⟨⟩ else ⟨mk-Set0(tl setlist)⟩ in
17           let (outputtrans, d'') =
18               transform-import(tnm, as1 tree, rest ↷ actl, term)(d') in
19           transform-transition(tnm, outputtrans, nodes1)(d'')))
20      else
21       exit ("§2.8: 在设置动作中参数表的长度必须等于定义中规定的长度")))
```

type: [*Statename*$_0$] *Setelem*$_0$* *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* →
Dict → (*Transition*$_1$ | *Graph-node*$_1$*) *Dict*

目的　　把一定时器设置变换到 AS$_1$

参数　　参见函数 *transform-set*

结果　　参见函数 *transform-set*

算法

第 1 行　　分解将要变换的设置元素 *Setelem*$_0$。
第 2 行　　令 *timesort* 表示时间类别的限定表 *Qual*。
第 3 行　　变换规定的时间表达式。
第 4 行　　抽取指定的标识符的限定表 *Qual*。
第 5 行　　本标识符必须表示一个定时器信号。
第 8 行　　令 *tplist* 表示为定时器规定的表达式的类别。
第 9 行　　在表中的表达式个数必须和定时器的类别的数目相同。
第 10 行　　变换这些表达式。
第 11 行　　构造此定时器的 AS$_1$ 标识符。
第 12 行　　构造 AS$_1$ 设置节点。
第 13 行　　如果在表达式表中或在时间表达式中没有进口表达式，则
第 14 行　　返回 AS$_1$ 设置节点，否则
第 16 行　　为其余的定时器的设置构造一个 AS$_0$ 设置动作。该动作将附属于隐式进口状态。
第 17 行　　用此进口状态更新 *Dict*，并且构造导向进口状态的 AS$_0$ 跃迁。
第 19 行　　变换该跃迁。

transform-export(*id*)(*dict*) ≙ (3.7.8.10)

```
1  (let sortset = all-visible-sorts(dict) in
2   let qual = get-visible-variable(id, sortset, true)(dict) in
3   let mk-SignalnamesD(impnm, , ) = exportmap((s-Sortqual(dict(qual)), s-Name0(id))) in
4   let as1 id = make-as1-identifier(qual)(dict),
5       as1 id' = make-as1-identifier(impnm)(dict) in
6   ⟨mk-Task-node1(mk-Assignment-statement1(as1 id', as1 id))⟩)
```

type: $Varid_0 \rightarrow Dict \rightarrow Task\text{-}node_1$

目的 把一个出口动作变换为一个 AS_1 赋值语句

参数

id 出口变量的 AS_0 标识符

结果 一个 AS_1 任务，它包含所构造的赋值语句

算法

第 1—2 行 利用上下文构造出口变量的 *Qual*。

第 3 行 令 *impnm* 表示附在出口变量上的隐式变量的名字。

第 4—5 行 构造显式出口变量的 AS_1 标识符($as_1 id$)和隐式出口变量的 AS_1 标识符($as_1 id'$)，并且

第 6 行 返回一个任务，它把显式变量赋给隐式变量。

3.7.8.1 输出节点的变换

transform-prioutput(*tnm*, mk-*Prioutput*$_0$(*outputsigs*), *actl*, *term*, *nodes*$_1$)(*dict*) ≙ (3.7.8.1.1)

```
1  (let self = mk-Selfexpr0() in
2   let (actl1, d) = transform-output-elems(tnm, outputsigs, self, ⟨⟩, actl, term, nodes1, true)(dict) in
3   if tl outputsigs = ⟨⟩ ∨ is-Transition1(actl1) then
4     (actl1, d)
5    else
6     transform-prioutput(tnm, mk-Prioutput0(tl outputsigs), actl, term, nodes1 ↷ actl1)(d))
```

type: $[Statename_0]\ Prioutput_0\ Actstmt_0^*\ [Termstmt_0]\ Graph\text{-}node_1^* \rightarrow$
$Dict \rightarrow (Transition_1 \mid Graph\text{-}node_1^*)\ Dict$

目的 把一个优先级输出动作变换为一个 AS_1 输出动作的序列（每个提到的信号需要一个输出动作）

参数

tnm,

actl,

term,

$nodes_1$ 参见变换动作函数 *transform-act*

outputsigs 包含在优先级输出动作中的一个输出表

结果 参见函数 *transform-act*

算法

第 1—6 行 此算法和变换复位函数 *transform-reset* 中的方法相同。对于函数 *transform-output-elems*（第 2 行），变元 **true** 表示这些信号是高优先级信号。

transform-output$(tnm, outnode, actl, term, nodes_1)(dict) \triangleq$ **(3.7.8.1.2)**

```
1  (let mk-Output0(outputsigs, expr, via) = outnode in
2   let (actl1, d) = transform-output-elems(tnm, outputsigs, expr, via, actl, term, nodes1, false)(dict) in
3   if tl outputsigs = ⟨⟩ ∨ is-Transition1(actl1) then
4     (actl1, d)
5    else
6     transform-output(tnm, mk-Output0(tl outputsigs, expr, via), actl, term, nodes1 ⌢ actl1)(d))
```

type: $[Statename_0]\ Output_0\ Actstmt_0^*\ [Termstmt_0]\ Graph\text{-}node_1^* \rightarrow$
$Dict \rightarrow (Transition_1 \mid Graph\text{-}node_1^*)\ Dict$

目的 把一个输出动作变换为一个 AS_1 输出动作的序列（每个提到的信号需要一个输出动作）

参数

tnm,
actl,
term,
$nodes_1$ 参见变换动作函数 *transform-act*
outputsigs 包含在输出动作中的输出表

结果 参见函数 *transform-act*

算法

第 1—3 行 本算法和变换复位函数 *transform-reset* 的方法相同，对于函数 *transform-output-elems*（第 2 行），变元 **false** 表示这些信号不是高优级信号。

transform-output-elems(*tnm*, *sigs*, *expr*, *via*, *actl*, *term*, *nodes*$_1$, *isprioutput*)(*dict*) ≜ **(3.7.8.1.3)**

```
 1  (let mk-Outputsig0(sigid, actparm) = hd sigs in
 2   let instsort = get-predef-sort("PID")(dict) in
 3   let (as1 expr, d) =
 4       if expr = nil
 5         then (nil, dict)
 6         else transform-expr(expr, EXPRESSION, {instsort})(dict) in
 7   let squal = signal-qual(sigid)(dict) in
 8   let qual = process-or-service-level(dict(SCOPEUNIT)) in
 9   let existreceiver = (∃q ∈ dom dict)(is-ServiceD(dict(q)) ∧
10                                       get-sur(q) = get-sur(qual) ∧
11                                       squal ∈ s-Priinputset(dict(q))) in
12   if (is-ProcessD(dict(qual)) ∧ isprioutput) ∨
13     (isprioutput ≠ existreceiver) then
14      exit("§4.10: 非法使用高优先级信号")
15     else
16      (let qualset = transform-via(squal, via)(dict) in
17       let as1 pathidset = make-as1 idset(qualset)(dict) in
18       let dict' = d + [OUTSIGNALS ↦ dict(OUTSIGNALS) ∪ {squal}] in
19       if is-TimerD(dict(squal)) then
20         exit("§2.7.4: 在输出动作中的标识符必须表示一个信号")
21        else
22         (let mk-SignalD(tplist, , ) = dict(squal) in
23          if len actparm = len tplist then
24            (let (as1 actparm, d') =
25                 transform-actparms(tplist, actparm, EXPRESSION)(dict') in
26             let as1 id = make-as1-identifier(squal)(dict) in
27             let as1 tree = mk-Output-node1(as1 id, as1 actparm, as1 expr, as1 pathidset) in
28             if d'(IMPORTLIST) = ⟨⟩ then
29               (⟨as1 tree⟩, d')
30              else
31               (let rest = if tl sigs = ⟨⟩ then
32                             ⟨⟩
33                           else
34                             ⟨mk-Output0(tl sigs, expr, via)⟩ in
35                let (outputtrans, d'') =
36                    transform-import(tnm, as1 tree, rest ⌢ actl, term)(d') in
37                transform-transition(tnm, outputtrans, nodes1)(d'')))
38           else
39            exit("§2.7.4: 对输出动作变元个数不正确")))))
```

type: [*Statename*$_0$] *Outputsig*$_0$* [*Expression*$_1$] *Via*$_0$ *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* *Bool* → *Dict* → (*Transition*$_1$ | *Graph-node*$_1$*) *Dict*

目的　　把一个输出或优先级输出变换到 AS$_1$。

参数

tnm,
actl,
term,
nodes$_1$　　参见变换动作函数 *transform-act*

sigs　　是包含在输出动作中的输出表。在本函数中仅变换该表的第一个元素。

expr　　是目的表达式。如果此输出是优先级输出，则 *expr* 是 SELF。

via 是包含在输出节点中的信道或信号路由标识符所组成的表。如果此输出是一个优先级输出，则此表为空。

结果 参见变换动作函数 *transform-act*

算法

第 1 行 令 *sigid* 表示要变换的输出的信号标识符，令 *actparm* 表示相关的实参表。

第 2 行 抽取 PiD 类别的限定表 *Qual*。

第 3—6 行 如果规定了目的表达式，则把它变换到 AS_1。它必须是 PiD 类别（*instsort*）的。

第 7 行 令 *squal* 表示信号的限定表 *Qual*。

第 8 行 令 *qual* 表示外围的服务或进程（不是过程）的限定表 *Qual*。

第 9—11 行 *existreceiver* 为真，如果在同一进程中（第 10 行）存在一个服务（*q*），它可接收在一个高优先级输入节点中的信号。

第 12—13 行 如果此输出包含在一个进程中，则它必须不是一个高优先级输出（第 12 行），并且如果它是一个高优先级信号，则必须存在一个接收服务，反之亦然。

第 16—17 行 构造 AS_1 VIA 集。

第 18 行 更新字典 *Dict* 的 **OUTSIGNALS** 项目以包括输出信号的 *squal*。

第 19 行 规定的信号必须不是定时器信号。

第 22 行 令 *tplist* 表示附在信号上的类别表。

第 23 行 实参表的长度必须等于类别表的长度。

第 24 行 变换这些实在参数。

第 26 行 构造信号的 AS_1 标识符。

第 27 行 构造 AS_1 输出结点。

第 28—29 行 如果在目的表达式中和实在参数中没有进口表达式，则返回此输出节点。否则

第 31 行 构造其余输出的一个 AS_0 输出动作，该动作将附属于隐式进口状态。

第 35 行 用进口状态更新 *Dict* 并且构造导向该进口状态的 AS_0 跃迁。

第 37 行 变换此跃迁。

transform-via(*signalqual*, *pathlist*)(*dict*) ≙ (3.7.8.1.4)

```
1  (let processlevel = process-level(dict(SCOPEUNIT)) in
2   let servicelevel = process-or-service-level(dict(SCOPEUNIT)) in
3   let mk-ProcessD(, sset, , connectmap) = dict(processlevel) in
4   let explicitsigroutes = s-Explicitroutes(dict(get-sur(processlevel))) in
5   if pathlist = ⟨⟩ then
6     if ¬explicitsigroutes then
7       {}
8      else
9       if (∃mk-SignalrouteD(endp1, endp2, sigset1, sigset2) ∈ dom dict)
10          ((processlevel = endp1 ∧ signalqual ∈ sigset1) ∨
11           (processlevel = endp2 ∧ signalqual ∈ sigset2)) then
12         {}
13        else
14         if signalqual ∈ sset then
15           {}
16          else
17           exit("§2.7.4: 输出中的信号没有可以接收的接收者")
18    else
19     (let id = hd pathlist in
20      let qual = (trap exit with get-visible-qual(id, CHANNEL)(dict) in
21                  get-visible-qual(id, SIGNALROUTE)(dict)) in
22      let qualrest = if tl pathlist = ⟨⟩ then
23                       {}
24                     else
25                      transform-via(signalqual, tl pathlist)(dict) in
26      if qual ∈ qualrest then
27        exit("§2.7.4: 在VIA集合中信号路由或信道被指定两次")
28       else
29        cases dict(qual):
30         (mk-SignalrouteD(endp1, endp2, sigset1, sigset2)
31            → (if (processlevel = endp1 ∧ signalqual ∈ sigset1) ∨
32                  (processlevel = endp2 ∧ signalqual ∈ sigset2) then
33                if connectmap ≠ [] ∧ is-ServiceD(dict(servicelevel)) ∧
34                  ¬(∃q ∉ connectmap(qual))
35                    ((let mk-SignalrouteD(ep1, ep2, sigs1, sigs2) = dict(q) in
36                      (servicelevel = ep1 ∧ signalqual ∈ sigs1) ∨
37                      (servicelevel = ep2 ∧ signalqual ∈ sigs2))) then
38                 exit("§4.10.2: 服务信号路由没连接到VIA集合中的信号路由")
39                else
40                 if is-ProcessD(dict(servicelevel)) ∧ connectmap ≠ []
41                   then exit("§2.10.2: 当定义服务信号路由时，VIA非法的外部服务")
42                   else {qual} ∪ qualrest
43               else
44                exit("§2.7.4: 非法VIA集合")),
```

```
45      mk-ChannelD(,,,,)
46        → if explicitsigroutes then
47            exit("2.7.4: 如果规定了信号路由，则不能在VIA中描述信道")
48          else
49           (let blockqual = get-sur(processlevel) in
50            let mk-BlockD(,,, cmap) = dict(blockqual) in
51            if qual ∉ dom cmap
52              then exit("§2.7.4: 在VIA(经由)集合中的信道未连接到功能块上")
53              else (let squalset =
54                      {q ∈ cmap(qual) | (let mk-SignalrouteD(endp1, endp2, sigset1, sigset2) =
55                                           dict(q) in
56                                         (processlevel = endp1 ∧ signalqual ∈ sigset1) ∨
57                                         (processlevel = endp2 ∧ signalqual ∈ sigset2))} in
58                    if squalset = {}
59                      then exit("§2.7.4: 信号不能由VIA集合中的信道传递")
60                      else squalset ∪ qualrest)))))
```

type: *Signalqual* Via_0 → *Dict* → *Qual*-**set**

目的

把在 VIA 构件中规定的信道或信号路由标识符表变换为一个信号路由的 *Qual* 集合

参数

signalqual	在输出中规定的信号的 *Qual*
pathlist	AS_0 的经过路径表

结果

信号路由的 *Qual* 集合

算法

第 1 行	令 *processlevel* 表示外围进程的（不是过程或服务）的 *Qual*。
第 4 行	如果信号路由已在外围功能块中规定了，则令 *explicitsigroutes* 等于 **true**。
第 5 行	如果没有规定信道标识符或信号路由标识符，则
第 6 行	如果对外围功能块没有规定信号路由，则返回空集。否则
第 9—11 行	如果存在一个信号路由，它包括规定的信号并且以外围进程作为它的一个端点，则返回空集（它表示该信号可以经由任意路径）。否则
第 14 行	如果规定了一些显式信号路由并且该信号不包括在它们之中，则该信号必须作为外围进程的有效输入信号集合的一部分。
第 19—59 行	如果在 VIA 中规定了信号路由或信道，则
第 19 行	令 *id* 表示在表中的第一个信道或信号路由的标识符。
第 20—21 行	把此标识符看作为一个信号路由，在 *Dict* 中查找此标识符。如果失败，则此标识符必定表示一个信道，并且在 *Dict* 中查找作为一个信道的这个标识符。
第 22 行	变换在经过路径表中其余的信道或信号路由。
第 26 行	如果信号路由或信道也出现在经过路径表其余部分的 *Qual* 中，则该信道或信号路由已被规定两次。
第 30—44 行	如果该 *Qual* 表示一个信号路由，则该信号必须包括在信号路由中，外围进程必须是它的一个端点。并且如果规定了服务的信号路由（即连接映射表 *connectmap* 非空），则

必有一个服务的信号路由与该服务和该信号路由相连（第 33—38 行）。

第 40 行　　如果规定了服务信号路由，则 VIA 不能用于进程级上的过程中。

第 45 行　　如果 *Qual* 表示一个信道，则

第 46—47 行　　一定是因为信号路由没有规定是显式的。

第 49 行　　令 *blockqual* 表示外围功能块的 *Qual*。

第 50 行　　令 *cmap* 表示此功能块的 *BlockconnectionD*。

第 51—52 行　　在 VIA 中的信道必须连接到功能块。

第 53—57 行　　令 *squalset* 表示和信道相连的信号路由的 *Qual*，并且信号是走出的信号。

第 58—59 行　　必须至少有一条这样的信号路由。

第 60 行　　返回选出的信号路由，把它与 VIA 集合中其余部分的信号路由汇合。

3.7.8.2　创建节点的变换

transform-create(*tnm*, **mk-***Create*$_0$(*pid*, *exprlist*), *actl*, *term*, *nodes*$_1$)(*dict*) ≙　　(3.7.8.2.1)

```
 1  (let pqual = get-visible-qual(pid, PROCESS)(dict) in
 2   let bqual = get-sur(pqual) in
 3   let as1id = make-as1-identifier(pqual)(dict) in
 4   let mk-ProcessD(tplist,,,) = dict(pqual) in
 5   if get-sur(process-level(dict(SCOPEUNIT))) ≠ bqual then
 6     exit("§2.7.2: 创建的进程必须被定义在同一个功能块中")
 7    else
 8     (if len tplist ≠ len exprlist then
 9        exit("§2.7.2: 实在参数表的长度必须等于形式参数表的长度")
10       else
11        (let (actparmlist, d) = transform-actparms(tplist, exprlist, EXPRESSION)(dict) in
12         let as1tree = mk-Create-request-node1(as1id, actparmlist) in
13         if d(IMPORTLIST) = ⟨⟩ then
14           (⟨as1tree⟩, d)
15          else
16           (let (outputtrans, d') = transform-import(tnm, as1tree, actl, term)(d) in
17            transform-transition(tnm, outputtrans, nodes1)(d')))))
```

type:　[*Statename*$_0$] *Create*$_0$ *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* →
Dict → (*Transition*$_1$ | *Graph-node*$_1$+) *Dict*

目的　　把一个创建请求动作变换到 AS$_1$

参数

tnm,

actl,

term,

nodes$_1$　　参见变换动作函数 *transform-act*

pid　　包含在创建请求动作中的进程标识符

exprlist　　实参表

结果　　参见变换动作函数 *transform-act*

算法

第 1 行　　令 *pqual* 表示进程标识符的 *Qual*。

第 2 行　　令 *bqual* 表示外围功能块的 *Qual*。

第 3 行　　构造进程标识符的 AS_1 标识符。

第 4 行　　令 *tplist* 表示实际参数的类别。

第 5 行　　放置创建请求节点的进程的外围功能块必须和进程标识符的外围功能块是同一功能块。

第 8 行　　类别表的长度必须等于实参表的长度。

第 11 行　　变换实在参数。

第 12 行　　构造 AS_1 创建请求节点。

第 13—14 行　　如果实在参数表不包含进口表达式，则返回构造的 AS_1 创建请求节点。

第 16 行　　否则用进口状态来更新 *Dict*，并且构造导向进口状态的 AS_0 跃迁。

第 17 行　　变换该跃迁。

transform-actparms(*tplist*, *exprlist*, *context*)(*dict*) $\triangleq$ **(3.7.8.2.2)**

```
1  (if tplist = ⟨⟩ then
2   ⟩ (⟨⟩,dict)
3     else
4      (let (as1 expr, , d) =
5           if hd exprlist = nil
6             then (nil, , dict)
7             else transform-expr(hd exprlist, context, {hd tplist})(dict) in
8       let (as1 rest, drest) = transform-actparms(tl tplist, tl exprlist, context)(d) in
9       (⟨as1 expr⟩ ⌢ as1 rest, drest)))
```

type: $ParameterD^*\ Actparmlist_0\ Context \rightarrow Dict \rightarrow [Term_1 \mid Expression_1]^*\ Dict$

目的　　把一个输出动作的实参表或一个创建请求动作的实参表或一个运算符变元表的实参表变换到 AS_1

参数

tplist　　实在参数的类别

exprlist　　实在参数表

context　　是变换表达式的上下文。在输出动作或创建请求动作的情况下，上下文总是 **EXPRESSION**。

结果　　在输出动作或创建请求动作的情况下，结果是一些任选表达式组成的表。在运算符实在参数的情况中结果是由一些项或表达式组成的表。*Dict* 可能被更新过以包括实参表中进口表达式的有关信息。

算法

第 1—2 行　　当全部处理完时，返回空表和（未改变的）字典 *Dict*。

第 4—7 行　　变换第一个实在参数，如果它在实参表中的话。

第 8 行　　变换其余的实在参数。

第 9 行　　返回与其余实在参数汇合的第一个实在参数。

3.7.8.3 调用节点的变换

transform-call(*tnm*, mk-*Call*$_0$(*procid*, *actparmlist*), *actl*, *term*, *nodes*$_1$)(*dict*) ≙ **(3.7.8.3.1)**

```
1  (let pqual = get-visible-qual(procid, PROCEDURE)(dict) in
2   let mk-ProcedureD(formparmlist, nqual) = dict(pqual) in
3   let as1 id = make-as1-identifier(nqual)(dict) in
4   if len actparmlist = len formparmlist then
5     (let (as1 actparm, d) = transform-actparmlist(formparmlist, actparmlist)(dict) in
6      let as1 tree = mk-Call-node1(as1 id, as1 actparm) in
7      if d(IMPORTLIST) = ⟨⟩ then
8        (⟨as1 tree⟩, d)
9       else
10       (let (outputtrans, d') = transform-import(tnm, as1 tree, actl, term)(d) in
11        transform-transition(tnm, outputtrans, nodes1)(d')))
12    else
13     exit("§2.7.3: 过程参数表的长度必须等于形式参数表的长度"))
```

type: [*Statename*$_0$] *Call*$_0$ *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* →
Dict → (*Transition*$_1$ | *Graph-node*$_1$*) *Dict*

目的　　把一个过程调用变换到 AS$_1$

参数

procid	过程的标识符
actparmlist	实参表
tnm,	
actl,	
term,	
nodes$_1$	参见变换动作函数 *transform-act*

结果　　参见变换动作函数 *transform-act*

算法

第 1 行	构造此过程的限定表 *Qual*。
第 2 行	令 *formparmlist* 表示形式参数的描述符，*nqual* 表示唯一的 *Qual*（参见 *Newqual* 的定义）。
第 3 行	构造此过程的 AS$_1$ 标识符。
第 4 行	形参表的长度必须等于实参表的长度。
第 5 行	变换实参表。
第 6 行	构造 AS$_1$ 调用节点。
第 7—8 行	如果在实在参数中没有进口表达式，则返回此调用节点。否则
第 10 行	更新状态字典 *Statedict* 以包括隐式进口状态，并返回导向隐式状态的 AS$_0$ 跃迁。
第 11 行	变换构造的跃迁。

transform-actparmlist(*formparmlist*, *actparmlist*)(*dict*) ≙ (3.7.8.3.2)

```
 1  (if formparmlist = ⟨⟩ then
 2     (⟨⟩,dict)
 3   else
 4     (let (actparm, dict') =
 5        if hd actparmlist = nil then
 6          (⟨nil⟩, dict)
 7        else
 8          cases hd formparmlist:
 9           (mk-InDescr(tqual)
10              → (let (as1 expr, , d) =
11                      transform-expr(hd actparmlist, EXPRESSION, {tqual})(dict) in
12                 (as1 expr, d)),
13            mk-InoutDescr()
14              → transform-inoutactparm(hd formparmlist, hd actparmlist)(dict)) in
15     let (actparmrest, drest) =transform-actparmlist(tl formparmlist, tl actparmlist)(dict') in
16     (⟨actparm⟩ ⌢ actparmrest, drest)))
```

type: *FormparmD** *Actparmlist*$_0$ → *Dict* → [*Expression*$_1$]* *Dict*

目的　　把一个过程的实在参数表变换为一个 AS_1 表达式表

参数

formparmlist	形式参数描述符表
actparmlist	实在参数表

结果　　是 AS_1 表达式表和一个字典 *Dict*。此 *Dict* 可能更新过以包括所含进口表达式的信息（参见变换动作函数 *transform-act*）。

算法

第 1—2 行	当全部处理完时，返回空参数表（此函数是递归的）。
第 4—14 行	令 *actparm* 表示第一个实在参数的 AS_1 表达式。
第 5—6 行	如果省略了这个实在参数，则 *actparm* 是 **nil**。否则
第 9—12 行	如果此参数是一个 IN 参数，则变换实参表达式，它的类别必须是 *tqual* 所指定的类别。
第 13—14 行	如果参数是一个 IN/OUT 参数，则变换该 IN/OUT 参数。
第 15—15 行	变换其余的实参。
第 16 行	返回与其余实参相汇合的第一个实参。

transform-inoutactparm(**mk-***InoutDescr*(*tqual*), *expr*)(*dict*) ≙ (3.7.8.3.3)

```
1  (if is-Id0(expr) then
2     (let vqual = get-visible-variable(expr, {get-parent(tqual)(dict)}, false)(dict) in
3      let mk-VarD(tq, , , , nqual) = dict(vqual) in
4      let as1 id = make-as1-identifier(nqual)(dict) in
5      if tq = tqual then
6        (as1 id, dict)
7       else
8        exit("§2.7.3: 对于IN/OUT形式参数和实在参数必须规定相同的类别"))
9    else
10    exit("§2.7.3: IN/OUT实在参数必须是一个变量标识符"))
```

type: *InoutDescr Expr*$_0$ → *Dict* → *Expression*$_1$ *Dict*

目的 把一个 IN/OUT 实在参数变换到 AS_1

参数 一个 IN/OUT 描述符包含

tqual 它的类别的限定表 *Qual*

expr 实在参数

结果 AS_1 实在参数

算法

第 1 行 此实参必须是一标识符。

第 2 行 构造此标识符的限定表 *Qual*。

第 3 行 令 *tq* 表示它的类别或同义类别，令 *nqual* 表示它的 *Newqual*。

第 4 行 构造其 AS_1 标识符。

第 5—8 行 变量的类别（*tq*）必须等于形参的类别。在此情况中，必须规定相同的类别引用的标识符，也就是一个类别与这个类别的同义类型是不相容的。

3.7.8.4 判定节点的变换

$transform\text{-}decision(tnm, decision, nodes_1)(dict) \triangleq$ (3.7.8.4.1)

```
(let mk-Decision0(quest, anslist, elsepart) = decision in
 let isinformalanswers =
     (∀mk-Condition0(v1, v2) ∈ elems anslist)(v1 = nil ∧ is-Stringterm0(v2) ∧ s-Qualifier0(v2) = ⟨⟩) in
 let isinformalquestion = is-Stringterm0(quest) ∧ s-Qualifier0(quest) = ⟨⟩ in
 let isinformaldecision = isinformalanswers ∧ isinformalquestion in
 let qualset = all-visible-sorts(dict) in
 let (,tpset,) = transform-expr(quest, EXPRESSION, qualset)(dict) in
 let tp = if isinformaldecision then
             nil
          else
           if isinformalquestion then
             (trap exit with get-answer-sort(anslist, qualset)(dict) in
              get-answer-sort(anslist, tpset)(dict))
            else
             get-answer-sort(anslist, tpset)(dict) in
 let (as1 trans, d'') = transform-answers(tnm, anslist, tp)(dict) in
 let (els, d''') =
     if elsepart = nil then
       (nil, d'')
      else
       (let mk-Elsepart0(trans) = elsepart in
        let (trans1, dict') = transform-transition(tnm, trans, ⟨⟩)(d'') in
        (mk-Else-answer1(trans1), dict')) in
 let (as1 quest',, d) =
     if isinformalquestion then
       (let mk-String0(str) = s-String0(quest) in
        if isinformaldecision then
          (mk-Informal-text1(str), {}, dict)
         else
          (trap exit with (mk-Informal-text1(str), {}, dict) in
           transform-expr(quest, EXPRESSION, {tp})(d''')))
      else
       transform-expr(quest, EXPRESSION, {tp})(d''') in
 let decisionnode1 = mk-Decision-node1(as1 quest', as1 trans, els) in
 if d(IMPORTLIST) = ⟨⟩ then
   (mk-Transition1(nodes1, decisionnode1), d)
  else
   (let (outputtrans, d') = transform-import(tnm, decisionnode1, ⟨⟩, nil)(d) in
    transform-transition(tnm, outputtrans, nodes1)(d')))
```

type: $[Statename_0]\ Decision_0\ Graph\text{-}node_1^* \rightarrow Dict \rightarrow Transition_1\ Dict$

目的

把判定节点变换到 AS_1

参数

tnm,

decision,

nodes$_1$

参见变换动作函数 *transform-act*。与其它动作的变换函数不同，当判定结束一个跃迁时，函数 *transform-decision* 不把跟随此判定的动作表当作变元，因为在这一阶段的所有回答都包含终端符。

结果

是一个含有 *nodes*$_1$ 的跃迁，该跃迁要么导向判定结点，要么导向隐式进口状态。

算法

第 1 行　　分解判定动作。

第 2 行　　令 *isinformalanswers* 为真，如果所有回答都包含一个非限定字符串的话。

第 4 行　　令 *isinformalquestion* 为真，如果问题中的表达式包含一非限定字符串的话。

第 5 行　　令 *isinformaldecision* 为真，如果此判定仅包含非形式正文（字符串）的话。

第 6 行　　令 *qualset* 表示在该位置可见的所有类别的 *Qual*。

第 7 行　　抽取判定问题的可能类别。

第 8—15 行　　令 *tp* 表示回答的类别。如果它是一个非形式判定，则 *tp* 等于 **nil**。否则就要通过查看这些回答来得出其类别，并因此限制此问题所允许的类别集合。此外，如果问题是一字符串（第 11 行），则任何可见类别可以匹配回答的集合（第 12 行），条件是它从与该问题匹配的类别的集合中没有导出回答的类别。

第 16 行　　给定了回答的类别（*tp*，如果整个判定是非形式的，则它为 **nil**），变换这些回答

第 17—23 行　　变换 else 部分。

第 21 行　　令 *trans* 表示 else 部分的跃迁。

第 22—23 行　　变换该跃迁，并且构造 AS_1 的 else 部分。

第 24—33 行　　变换该问题。

第 24 行　　如果该问题是一个非限定字符串，则在下述两种情况下它表示非形式正文。情况 1：如果整个判定是非形式的（第 27 行）；情况 2：如果给定该回答的类别 *tp*，作为字符串字面值的问题的变换失败（第 31 行）。

第 34 行　　构造 AS_1 的判定节点。

第 35—36 行　　如果该问题不包含进口表达式，则返回包含前面的节点和判定节点的跃迁。否则

第 38 行　　更新状态字典 *Statedict* 以包括隐式进口状态，并构造导向此隐式进口节点的 AS_0 跃迁。

第 39 行　　变换构造的跃迁。

get-answer-sort(*anslist*, *tpset*)(*dict*) ≜ **(3.7.8.4.2)**

```
1  (if anslist = ⟨⟩ then
2     if card tpset ≠ 1 then
3        exit("§2.2.2: 对判定动作不存在适当的类别")
4      else
5       (let tp ∈ Qual be s.t. tp ∈ tpset in
6        tp)
7   else
8    (let mk-Answer0(vlist, ) = hd anslist in
9     let isinformal′ =
10        len vlist = 1 ∧
11        (let mk-Condition0(v1, v2) = hd vlist in
12         v1 = nil ∧ is-Stringterm0(v2) ∧ s-Qualifier0(v2) = ⟨⟩) in
13    let (tset, ) = if isinformal′ then (tpset, ) else transform-valueset(tpset, vlist)(dict) in
14    get-answer-sort(tl anslist, tset)(dict))
```

type: $Answer_0^+$ *Sortqual*-**set** → *Dict* → [*Sortqual*]

目的　　抽取一个判定回答的类别

参数

anslist 附在判定动作上的回答表（除 else 部分外）

tpset 是可能的回答的类别集合。在对函数 *get-answer-sort* 的每次递归调用期间，该集合被当前回答的允许类别所限制。初始应用时，*tpset* 表示判决问题的可能类别，或者如果判定问题是非形式的，则 *tpset* 表示所有可见的类别。

结果 判别动作的类别的限定表 *Qual*

算法

第 1—5 行 当处理完该回答表时，则可能的类别集合必须仅含有一个元素，并且返回此元素。

第 8 行 分解表中第一个回答。

第 9—12 行 令 *isinformal'* 为真，如果回答包含一个非限定字符串。

第 13 行 令 *tset* 表示在已经考虑第一个（下一个）回答后这些回答的可能类别的集合。如果回答是一非限定字符串，则在决定回答的类别时，它是不考虑的。

第 14 行 考虑其余的回答。

transform-answers(*tnm*, *anslist*, *tp*)(*dict*) ≙ **(3.7.8.4.3)**

```
1  (if anslist = {} then
2     ({}, dict)
3    else
4     (let mk-Answer0(vlist, trans) = hd anslist in
5      let (,as1tree) =
6          if tp = nil then
7            ({}, nil)
8           else
9            (trap exit with ({},nil) in
10            transform-valueset({tp}, vlist)(dict)) in
11     let (trans1, dict') = transform-transition(tnm, trans, ⟨⟩)(dict) in
12     let (restrans, dict'') = transform-answers(tnm, tl anslist, tp)(dict') in
13     if as1tree = nil then
14       (let mk-Condition0(, mk-Stringterm0(, lit)) = hd vlist in
15        ({mk-Informal-text1(lit)} ∪ restrans, dict''))
16      else
17       ({mk-Decision-answer1(as1tree, trans1)} ∪ restrans, dict'')))
```

type: [*Statename*$_0$] *Answer*$_0^*$ [*Sortqual*] → *Dict* → *Decision-answer*$_1$-**set** *Dict*

目的 把一个判定的回答或一个任选的回答变换到 AS$_1$

参数

tnm 参见变换动作函数 *transform-act*

anslist 要变换的回答表

tp 是回答的类别。如果 *tp* 等于 **nil**，则该问题和所有的回答都是由字符串组成的（即判定是非形式的）。

结果 AS$_1$ 回答的集合

算法

第 1 行 当处理完回答表时，返回空集（本函数是递归的）。

第 4 行 令 *vlist* 和 *trans* 表示在回答表中第一个回答的条件和跃迁。

第 5—10 行 试变换这些条件。如果失败了，则这些条件必须含有非形式正文，因为已经检查了

vlist 的一致性（在函数 *get-answer-sort* 中）。

第 11 行　　变换第一个回答的跃迁。

第 12 行　　变换其余的回答。

第 13—15 行　　如果当前条件表的变换失败了，则条件表必须是表示非形式正文。

第 15 行　　把此条件作为非形式正文进行变换，并返回所得到的回答，与其余的 AS_1 回答汇合。

第 17 行　　否则返回已变换的回答，与其余的 AS_1 回答汇合。

3.7.8.5 任选的变换

$$\textit{transform-option}(\textit{tnm}, \textbf{mk-}\textit{Option}_0(\textit{quest}, \textit{anslist}, \textit{elsepart}), \textit{nodes}_1)(\textit{dict}) \triangleq \qquad (3.7.8.5.1)$$

```
1  (let qualset = all-visible-sorts(dict) in
2   let (,tpset,) = transform-expr(quest, CONSTANT, qualset)(dict) in
3   let tp = get-answer-sort(anslist, tpset)(dict) in
4   let trans = eval-option-trans(tp, quest, anslist, elsepart)(dict) in
5   transform-transition(tnm, trans, nodes1)(dict))
```

type：　$[Statename_0]\ Option_0\ Graph\text{-}node_1^* \rightarrow Dict \rightarrow Transition_1\ Dict$

目的　　处理一个任选动作并返回所得到的跃迁

参数

tnm，

$nodes_1$　　参见变换动作函数 *transform-act*

算法

第 1 行　　令 *qualset* 表示在此位置上可见的所有类别的 *Qual*。

第 2 行　　变换此任选的问题。*tpset* 表示该问题允许的类别。

第 3 行　　令 *tp* 表示此任选的类别。它是通过检查各个回答导出的（与函数 *transform-decision* 中所用的方法相同）。

第 4 行　　选择与该问题匹配的跃迁。

第 5 行　　变换选择的跃迁。

eval-option-trans(*tp*, *question*, *anslist*, *elsepart*)(*dict*) ≙ (3.7.8.5.2)

```
1  (if anslist = ⟨⟩ then
2     if elsepart = nil then
3        exit("§4.3.4: 没有一个任选的回答与该任选的问题匹配")
4       else
5        s-Transition0(elsepart)
6     else
7      (let mk-Answer0(vlist, trans) = hd anslist in
8       if match-option-answer(tp, vlist, question)(dict) then
9          if (∃ans ∈ elems tl anslist)(match-option-answer(tp, s-Conditionlist0(ans), question)(dict)) then
10            exit("§4.3.4: 不止一个任选的回答与该问题匹配")
11           else
12            trans
13        else
14         eval-option-trans(tp, question, tl anslist, elsepart)(dict)))
```

type: *Sortqual Expr*$_0$ *Answer*$_0$* [*Elsepart*$_0$] → *Dict* → *Transition*$_0$

目的 选择与任选问题匹配的一个任选动作的回答

参数

tp	问题的类别的限定表 *Qual*
question	任选的问题
anslist	任选回答表
elsepart	任选的 else 部分

结果 包含在匹配的回答中的跃迁

算法

第 1—5 行	如果没有回答与问题匹配，则必须规定一个 else 部分，并且如果如此，则返回包含在 else 部分中的跃迁。
第 7 行	分解表中第一个（下一个）回答。
第 8 行	如果此回答与问题匹配，则
第 9—12 行	它必须是唯一的匹配回答，并且如果如此，则返回包含在该回答中的跃迁。
第 14 行	否则继续搜索一个匹配的答案。

match-option-answer(*tp*, *vlist*, *question*)(*dict*) ≙ (3.7.8.5.3)

```
1  (let as0operator = build-answer-operator(tp, vlist, question) in
2   eval-simple-expr(as0operator, "BOOLEAN")(dict))
```

type: *Sortqual Conditionlist*$_0$ *Expr*$_0$ → *Dict* → *Bool*

目的 测试一个任选条件表是否与此任选的问题匹配

参数

tp	问题和条件的类别
vlist	条件表
question	任选的问题

结果　　如果条件表与问题匹配，则结果为真。

算法

第 1 行　　从问题和条件表构造一布尔运算符的应用。如果条件表与该问题匹配，应用的结果为真。

第 2 行　　计算运算符的应用。

build-answer-operator$(tp, vlist, question) \triangleq$ **(3.7.8.5.4)**

```
1  (let mk-Condition0(op, const) = hd vlist in
2   let orop = mk-Qualop0(⟨⟩, mk-Quotedop0(OR)) in
3   let op' = if op = nil then EQ else op in
4   let as0op = if op' ∈ {NE, EQ, GT, LT, LE, GE} then
5                  mk-Operatorapp0(mk-Qualop0(tp, mk-Quotedop0(op')), ⟨question, const⟩)
6                 else
7                  (let andop = mk-Qualop0(⟨⟩, mk-Quotedop0(AND)) in
8                   let lesseqop = mk-Qualop0(tp, mk-Quotedop0(LE)) in
9                   mk-Operatorapp0(andop, ⟨mk-Operatorapp0(lesseqop, ⟨op', question⟩),
10                                         mk-Operatorapp0(lesseqop, ⟨question, const⟩)⟩)) in
11  if tl vlist = ⟨⟩
12    then as0op
13    else mk-Operatorapp0(orop, ⟨as0op, build-answer-operator(tp, tl vlist, question)⟩))
```

type:　*Sortqual Conditionlist*$_0$ *Expr*$_0$ → *Operatorapp*$_0$

目的　　构造一布尔运算符，如果一个任选的回答与该任选的问题匹配，则该运算符给出布尔字面值 true。

参数

tp　　问题的类别

vlist　　包含在回答中的条件表

question　　任选的一个问题

结果　　构造的 AS$_0$ 运算符的应用

算法

第 1 行　　分解条件表中的第一个条件。

第 2 行　　构造前缀“OR”运算符名字。

第 3 行　　如果没有规定运算符或表达式作为条件中的第一个元素，那么就和规定了相等运算符（EQ）一样。

第 4 行　　如果在 *Condition*$_0$ 中的第一个元素是一个关系运算符，则构造运算符

tp op’ (question, const)

其中，tp 是运算符的限定符，op′ 是此关系运算符，而（question，const）是变元表。

第 7—10 行　　如果 *Condition*$_0$ 中的第一个元素是一个表达式，则此条件反映一个范围，构造的运算符则为

“AND”（tp “<=”（op’, question），tp “<=”（question, const））

其中，tp 是限定符，op′ 是下界，const 是上界。

第 11 行　　如果本条件是表中最后一个条件，则返回构造的运算符应用。否则

第 13 行　　构造一个运算符的应用，它把先前的运算符应用的结果与从条件表其余部分构造的运算符应用的结果进行“OR”（或）运算。

3.8 通用的 AS_1 创建函数

make-as$_1$*-identifier*(*q*)(*dict*) $\triangleq$ **(3.8.1)**

```
1  (let qual = make-new-qual(q)(dict) in
2   let nameq = get-sur(qual) in
3   let (,nm) = qual[len qual] in
4   let as1 nameq = make-as1-qual(nameq) in
5   mk-Identifier1(as1 nameq, name-to-name1(nm)))
```

type: *Qual* → *Dict* → *Identifier*$_1$

目的　　从一个限定表 *Qual*（*q*）构造并返回对应的 AS_1 标识符

算法

第 1 行　　从字典 *Dict* 描述符中抽取 *Newqual*（参见 *Newqual* 的定义），即，在 *Qual* 中插入各个不同的名字。

第 2 行　　令 *nameq* 表示它的限定符（即外围作用域的 *Qual*）。

第 3 行　　抽取此实体的名字部分。

第 4 行　　把此 *Qual* 变换为一个 AS_1 限定符。

第 5 行　　构造 AS_1 标识符，它包含名字串或中缀运算符的 AS_1 表示法。

make-as$_1$*-qual*(*qual*) $\triangleq$ **(3.8.2)**

```
1  if tl qual = ⟨⟩ then
2    ⟨⟩
3   else
4   (let (q, nm) = hd qual in
5    let as1 nm = name-to-name1(nm) in
6    let as1 q = cases q:
7                (SYSTEM
8                   → mk-System-qualifier1(as1 nm),
9                 BLOCK
10                  → mk-Block-qualifier1(as1 nm),
11                PROCESS
12                  → mk-Process-qualifier1(as1 nm),
13                PROCEDURE
14                  → mk-Procedure-qualifier1(as1 nm),
15                TYPE
16                  → mk-Sort-qualifier1(as1 nm),
17                SIGNAL
18                  → mk-Signal-qualifier1(as1 nm),
19                SUBSTRUCTURE
20                  → mk-Block-substructure-qualifier1(as1 nm)) in
21   let q' = make-as1-qual(tl qual) in
22   (as1 q ⌢ q'))
```

type: *Qual* → *Qualifier*$_1$

目的　　从一个限定表 *Qual*（*qual*）构造并返回一个 AS_1 限定符

算法

第 1 行　　当全部处理完时，返回空限定符（本函数是递归的）。

第 4 行　　令 *q* 和 *nm* 表示限定表 *qual* 中的第一个元素的实体类和名字。

第 5 行　　令 as_1nm 表示该名字的 AS_1 表示法。

第 7 行　　如果此实体类就是系统，则构造一个 AS_1 系统限定符元素。

第 9 行　　如果实体类是一个功能块，则构造一个 AS_1 功能块的限定符元素。

第 11 行　　如果实体类是一个进程，则构造一个 AS_1 进程限定符元素。

第 13 行　　如果实体类是一个过程，则构造一个 AS_1 过程限定符元素。

第 15 行　　如果实体类是类别，则构造一个 AS_1 类别限定符元素。

第 17 行　　如果实体类是一个信号，则构造一个 AS_1 信号限定符元素。

第 19 行　　如果实体类是一个功能块子结构，则构造一个 AS_1 功能块子结构限定符。其它两种实体类服务和信道子结构在函数 *make-new-qual* 中已经把它们从 *Qual* 中删除了。

第 21—22 行　　把该 *Qual* 的其余部分变换到 AS_1，并把它与此元素汇合。

make-new-qual(*qual*)(*dict*) ≙ **(3.8.3)**

```
 1  (let level = dict(SCOPEUNIT) in
 2   let qual' = cases dict(qual):
 3               (mk-TimerD(, q)
 4                  → q,
 5                mk-ProcedureD(, q)
 6                  → q,
 7                mk-SortD(, , , q)
 8                  → q,
 9                mk-SyntypeD(, q, , )
10                  → q,
11                mk-VarD(, , , , q)
12                  → q,
13                mk-OperatorD(, , q, )
14                  → q,
15                mk-ChannelD(, , , , sub)
16                  → if sub = nil then qual else convert-channel-qual(sub, level),
17                mk-ChannelsubD(bqual)
18                  → bqual ⌢ ⟨qual[len qual]⟩,
19                mk-ServiceD(, , , , )
20                  → get-sur(qual),
21                T → qual) in
22  let qualelem = qual'[len qual'] in
23  if tl qual' = ⟨⟩ then
24    ⟨qualelem⟩
25   else
26    (let qualrest =
27        if is-OperatorD(dict(qual)) ∨ is-LiteralD(dict(qual))
28          then get-sur(get-sur(qual'))
29          else get-sur(qual') in
30     make-new-qual(qualrest)(dict) ⌢ ⟨qualelem⟩))
```

type:　*Qual* → *Dict* → *Qual*

目的　　在一个实体的 *Qual* 中插入各个不同的名字。由于有多个服务，也由于要避免运算符的重载，各个不同的名字是需要的。

结果　　新的唯一 *qual*

算法

第 1 行　　令 *level* 表示定义此实体的作用域单位的 *Qual*。

第 3—21 行　　在此实体的描述符中找到了新的 *Qual*。只有在服务、信道和信道子结构中可以被定义的实体才被重新命名。(如果信道包含一个子结构 (*sub*≠**nil**)，则它们被重新命名。并且信道子结构被重新命名为一个功能块子结构)。对于一个服务的 *Qual*，返回包围进程的限定符 *Qual*。

第 22—29 行　　取出定义该实体的作用域单位的 *qual* (*qualrest*)，并重新命名那个 *Qual*。如果旧的 *Qual* 表示一个运算符或一个字面值，则在新 *Qual* 中不要使用 **TYPE** 限定符元素。

第 30 行　　返回与偶对（实体类和该实体的名字）汇合的重新命名的作用域单位。

$convert\text{-}channel\text{-}qual(sub, level) \triangleq$ (3.8.4)

```
1  (let (b1, c1, b2, c2) = sub in
2   if (∃q)(level = b1 ⌢ q) then
3     get-sur(b1) ⌢ ⟨(CHANNEL, c1)⟩
4    else
5     get-sur(b2) ⌢ ⟨(CHANNEL, c2)⟩)
```

type: *Newchannels Qual* → *Qual*

目的　　包含一个子结构的信道用两个新信道表示，每个新信道都导向由一个功能块子结构表示的子结构。当使用此信道的名字时（在 VIA 集合中的一个标识符里），该名字由一个新信道名代替。用哪一个新信道名取决于使用信道的位置。函数 *convert-channal-qual* 返回这样一个新信道的 *Qual*。请注意，在信道子结构内部不允许引用旧信道名。

参数

sub　　关于两个新信道的信息，此信息取自信道描述符

level　　用于信道的作用域单位

结果　　新信道的 *Qual*

算法

第 1 行　　令 *b*1 表示一个功能块端点，*c*1 表示连到该端点的新信道，*b*2 表示另一个功能块端点，*c*2 表示连到 *b*2 端点的新信道。

第 2—5 行　　如果 *level* 表示包含在 *b*1 中的作用域单位，则返回与那个功能块相连的信道的限定表 *Qual*，否则返回另一个端点的 *Qual*。

$make\text{-}as_1 idset(idset)(dict) \triangleq$ (3.8.5)

```
1  {make-as1-identifier(id)(dict) | id ∈ idset}
```

type: *Qual*-**set** → *Dict* → *Identifier*$_1$-**set**

目的　　从一个 *Qual* 的集合构造对应的 AS_1 标识符集

make-as$_1$*tree*(*node*, *name*, *as*$_1$*set*, *parm*1, *parm*2, *parm*3)(*dict*) ≙ (3.8.6)

```
 1  (let as1name = name-to-name1(name),
 2       data = dict(DATATYPEDEF) in
 3   if sorts-have-values(data)(dict) then
 4     (let blocks = {elem | elem ∈ as1set ∧ is-Block-definition1(elem)},
 5          channels = {elem | elem ∈ as1set ∧ is-Channel-definition1(elem)},
 6          signals = {elem | elem ∈ as1set ∧ is-Signal-definition1(elem)},
 7          sigroutes = {elem | elem ∈ as1set ∧ is-Signal-route-definition1(elem)},
 8          syntypes = {elem | elem ∈ as1set ∧ is-Syn-type-definition1(elem)},
 9          procedures = {elem | elem ∈ as1set ∧ is-Procedure-definition1(elem)},
10          processs = {elem | elem ∈ as1set ∧ is-Process-definition1(elem)},
11          views = {elem | elem ∈ as1set ∧ is-View-definition1(elem)},
12          timers = {elem | elem ∈ as1set ∧ is-Timer-definition1(elem)},
13          vars = {elem | elem ∈ as1set ∧ is-Variable-definition1(elem)} in
14      cases node:
15       (SYSTEM
16          → mk-System-definition1(as1name, blocks, channels, signals, data, syntypes),
17        BLOCK
18          → mk-Block-definition1(as1name, processs, signals, parm1, sigroutes, data, syntypes, parm2),
19        SUBSTRUCTURE
20          → mk-Block-substructure-definition1(as1name, blocks, parm1, channels, signals, data, syntypes),
21        PROCESS
22          → mk-Process-definition1(as1name, parm1, parm2, procedures, signals, data,
23                                   syntypes, vars, views, timers, parm3),
24        PROCEDURE
25          → mk-Procedure-definition1(as1name, parm1, procedures, data, syntypes, vars, parm2)))
26    else
27     exit("§5.2.1: 数据类型定义中的类别没有值"))
```

type: *Quot* [*Name*$_0$] *Decl*$_1$-**set** [*Channel-to-route-connection*$_1$ | *Channel-connection*$_1$ | *Number-of-instances*$_1$ | *Procedure-formal-parameter*$_1$*] [*Process-formal-parameter*$_1$* | *Procedure-graph*$_1$ | *Block-substructure-definition*$_1$] [*Process-graph*$_1$] → *Dict* → *Decl*$_1$

目的　　构造系统的、功能块的、功能块子结构的、进程的或过程的作用域单位的 AS$_1$ 表示法

参数

node	要构造的作用域单位的种类
name	作用域单位的名字
as$_1$*set*	包含在作用域单位中的定义的集合
*parm*1, *parm*2, *parm*3	是根据不同的作用域单位有所不同的一些参数。例如，如果作用域单位是一进程，则 *parm*1 是实例数 *Number-of-instances*$_1$。

结果　　AS$_1$ 的系统、功能块、功能块子结构、进程或过程的定义

算法

第 1 行	抽取名字的 AS$_1$ 表示法。
第 2 行	抽取作用域单位的数据类型定义 *Data-type-definition*$_1$。
第 3 行	在数据类型定义 *Data-type-definition*$_1$ 中的每个类别必须至少有一个值。
第 4—13 行	把定义集合划分为一组功能块定义组成的集合。

第 5 行　　信道定义的集合。
第 6 行　　信号定义的集合。
第 7 行　　信号路由定义的集合。
第 8 行　　同义词定义的集合。
第 9 行　　过程定义的集合。
第 10 行　　进程定义的集合。
第 11 行　　视见定义的集合。
第 12 行　　定时器定义的集合。
第 13 行　　变量定义的集合。
第 15—16 行　　构造一个系统定义。
第 17—18 行　　构造一个功能块定义。
第 19—20 行　　构造一个功能块子结构定义。
第 21—23 行　　构造一个进程定义。
第 24—25 行　　构造一个过程定义。

$$sorts\text{-}have\text{-}values(\mathbf{mk}\text{-}Data\text{-}type\text{-}definition_1(,,snmset, signatureset,,))(dict) \triangleq \quad (3.8.7)$$

```
1  (let level = dict(SCOPEUNIT) in
2   (∀nm ∈ snmset)
3    ((∃sign ∈ signatureset)
4      ((let mk-Identifier1(qual, signm) = s-Result1(sign) in
5        level = qual ∧ nm = signm))))
```

type: $Data\text{-}type\text{-}definition_1 \rightarrow Dict \rightarrow Bool$

目的　　测试一个作用域单位的一个数据类型定义对每一类别是否都有值，即，是否至少存在该类别的一个字面值或者存在一个运算符返回该类别的值。

参数　　数据类型定义包含

snmset　　由此数据类型定义定义的类别的集合
signatureset　　此数据类型定义的字面值和运算符标记的集合

结果　　如果成功，则结果为真。

算法
第 2 行　　对类别集合中的所有类别名字，必须满足在标记集合中至少有一个标记使得
第 4—5 行　　结果的类别是局部定义的，并且它的名字与类别名（*nm*）相同。

make-as$_1$*-concaxioms*(*qualset*)(*dict*) ≜ **(3.8.8)**

```
1  (if qualset = {} then
2     {}
3    else
4     (let qual ∈ qualset in
5      let (,nm) = qual[len qual] in
6      let sort = get-sur(qual) in
7      if is-Name0(nm) then
8        make-as1-concaxioms(qualset \ {qual})(dict)
9       else
10       (let mk-String0(str) = nm in
11        if len str ≤ 1 then
12          make-as1-concaxioms(qualset \ {qual})(dict)
13         else
14          (let ch = str[len str] in
15           let rest = mk-String0(⟨str[i] | 1 ≤ i ≤ len str − 1⟩) in
16           if sort ⌢ ⟨(LITERAL, rest)⟩ ∉ dom dict then
17             make-as1-concaxioms(qualset \ {qual})(dict)
18            else
19             (let concop = mk-Qualop0(sort, mk-Quotedop0(CONC)) in
20              let strexpr1 = mk-Stringterm0(⟨⟩, rest),
21                  strexpr2 = mk-Stringterm0(⟨⟩, mk-String0(⟨ch⟩)),
22                  strexpr3 = mk-Stringterm0(⟨⟩, nm) in
23              let equation =
24                  mk-Equation0(mk-Operatorterm0(concop, ⟨strexpr1, strexpr2⟩), strexpr3) in
25              let equation1 =
26                  (trap exit with {} in
27                   {transform-axiom(equation, AXIOMS)(dict)}) in
28              equation1 ∪ make-as1-concaxioms(qualset \ {qual})(dict))))))
```

type: *Qual*-**set** → *Dict* → *Equations*$_1$

目的

构造附在字符串字面值上的隐含的拼接公理的 AS$_1$ 表示法，例如'abc'具有的隐含公理：

```
'ab'//'c' == 'abc';
'a'//'b'  == 'ab';
```

假设等式是静态良形式的。

参数

在一个部分数据类型定义中定义的字面值的 Qual 的集合

结果

构造的 AS$_1$ 等式

算法

第 1 行　当全部处理完时，返回等式的空集（本函数是递归的）。
第 4 行　考虑下一字面值。
第 5 行　令 *nm* 表示字面值的名字。
第 6 行　令 *sort* 表示其类别。
第 7—8 行　如果此字面值由一个正规名字表示，则继续处理其余的字面值，否则
第 10 行　令 *str* 表示字符串中的字符。
第 11—12 行　只要字符数不大于 1，就继续处理其余的字面值。
第 14 行　令 *ch* 表示该字符串中的最后一个字符。
第 15 行　令 *rest* 表示把最后一个字符除外的字符串。

第 16—17 行　　如果删除了最后一个字符的字符串对于该类别而言是未定义的，则继续处理其余的字面值。

第 19 行　　构造“//”运算符。

第 20 行　　构造一个包含其余字符的字符串项，但不包含限定符（*strexpr*1）。

第 21 行　　构造一个包含最后字符的字符串项，但不包含限定符（*strexpr*2）。

第 22 行　　构造一个包含该字符串字面值的字符串项（*strexpr*3）。

第 23 行　　构造等式：**strexpr**1 // **strexpr**2 = = **strexpr**3

第 25—27 行　　试把等式变换到 AS_1。如果失败，则等式不是良形式的，并因此不能隐含此等式。

第 28 行　　把此等式和隐含于其余字面值的等式汇合到一起。

$as_0\text{-}id(qual) \triangleq$ **(3.8.9)**

```
1  if qual = ENV then
2    ENV
3   else
4    (let (,nm) = qual[len qual] in
5     let qualifier = get-sur(qual) in
6     mk-Id0(qualifier, nm))
```

type: $(Qual \mid \text{ENV}) \rightarrow (Id_0 \mid \text{ENV})$

目的　　从一个 *Qual* 中取出一个 AS_0 标识符。该函数用于扩展速记符。

参数

qual　　要转换到 AS_0 标识符的限定表 *Qual*。因为此函数也应用于信道和信号路由端点，此函数可以用 **ENV** 作为变元。

结果　　构造的 AS_0 标识符

算法

第 1 行　　如果它是 ENV，则不要改变它。否则

第 4 行　　抽取名字部分。

第 5 行　　抽取限定符，该限定符是外围作用域单位的 *Qual*。

第 6 行　　返回组合的标识符。

make-implicit-vardecl(*dict*) $\triangleq$ **(3.8.10)**

```
1  (let (,f1nm, f2nm) = dict(GLOBALNAMES) in
2   let as0id = as0-id(get-predef-sort("INTEGER")(dict)) in
3   let init = mk-Id0(⟨⟩, mk-Name0("0", nil)) in
4   let formudecl = mk-Vardef0(nil, nil, ⟨mk-Vardefelem0(⟨f1nm, f2nm⟩, as0id, init)⟩) in
5   let closureset = dict(IMPLIED) in
6   let as0defs = formudecl ⌢ ⟨make-implicit-import-vardef(closure) | closure ∈ closureset⟩ in
7   transform-vardef(as0defs)(dict))
```

type: *Dict* → *Variable-definition*$_1$ **-set** *Dict*

目的　　为用于允许条件和连续信号的两个隐式变量构造 AS_1 变量的定义，并且为进程中的隐式进口变量构造 AS_1 变量的定义

算法

第 1 行　　令 *f1nm* 和 *f2nm* 表示用于允许条件和连续信号中的两个隐式变量的 AS_0 名字。

第 2 行　　令 as_0id 表示 INTEGER 类别的 AS_0 标识符。

第 3 行　　构造初始化表达式。这些变量初始化为 0。

第 4 行　　为这两个变量构造 AS_0 变量定义。

第 5 行　　抽取（变量名和变量类别的）偶对的集合，它包含用于进口表达式变换中的隐式变量。

第 6 行　　令 as_0defs 表示所有的隐式定义。

第 7 行　　把定义变换到 AS_1。

$$make\text{-}implicit\text{-}import\text{-}vardef((t, nm)) \triangleq \tag{3.8.11}$$

```
1  (let as0tid = as0-id(t) in
2   mk-Vardef0(nil, nil, ⟨mk-Vardefelem0(⟨nm⟩, as0tid, nil)⟩))
```

type: $NameclosureD \rightarrow Vardef_0$

目的　　构造隐式进口变量的 AS_0 变量定义

参数　　由变量类别 t 和变量名（nm）组成的偶对

算法

第 1 行　　构造变量类别的 AS_0 标识符。

第 2 行　　构造 AS_0 变量定义。此变量不用初始化，因为它的第一次使用总是在一个输入节点中。

transform-import(*tnm*, *originalnode*$_1$, *actl*, *term*)(*dict*) $\triangleq$ (3.8.12)

```
(let importlist = dict(IMPORTLIST) in
 let (impnm, qual, expr) = hd importlist,
     statenm = create-unique-name() in
 let (, (nm, )) = qual[len qual] in
 let mk-ImportD(tq) = dict(qual) in
 let mk-SignalnamesD(, xq, xr) = exportmap((tq, nm)) in
 let lev = dict(SCOPEUNIT) in
 let osig = mk-Outputsig0(mk-Id0(⟨⟩, xq), ⟨⟩) in
 let outputnode = mk-Actstmt0(nil, mk-Output0(⟨osig⟩, expr, ⟨⟩)) in
 let sigset = all-input-signals(lev)(dict) \ {lev ⌢ ⟨(SIGNAL, xr)⟩} in
 let savenode = mk-Savespec0(⟨as0-id(sig) | sig ∈ sigset⟩),
     statedict = dict(STATEDICT) in
 let termstmt' = mk-Termstmt0(nil, mk-Nextstate0(mk-Id0(⟨⟩, statenm))) in
 let outputtrans = mk-Transition0(outputnode, termstmt') in
 let impdict = [IMPLIED      ↦ (dict(IMPLIED) ∪ {(tq, impnm)}),
                IMPORTLIST ↦ tl importlist] in
 if tl importlist = ⟨⟩ then
   (let inputtrans = mk-Transition0(actl, term) in
    let inpvars = ⟨mk-Inputvars0(mk-Id0(⟨⟩, xr), ⟨mk-Id0(⟨⟩, impnm)⟩)⟩ in
    let inputnode = mk-Inputspec0(inpvars, nil, inputtrans) in
    let stated = mk-StateD(⟨(lev, inputnode), (lev, savenode)⟩, (tnm, originalnode1)) in
    let statedict' = statedict + [statenm ↦ stated] in
    let dict' = dict + [STATEDICT ↦ statedict'] in
    (outputtrans, dict' + impdict))
  else
   (let (termstmt'', dict'') = transform-import(tnm, originalnode1, actl, term)(dict + impdict) in
    let inpvars = ⟨mk-Inputvars0(mk-Id0(⟨⟩, xr), ⟨mk-Id0(⟨⟩, impnm)⟩)⟩ in
    let inputnode = mk-Inputspec0(inpvars, nil, termstmt'') in
    let statedict' =
        statedict + [statenm ↦ mk-StateD(⟨(lev, inputnode), (lev, savenode)⟩, (tnm, nil))] in
    let dict''' = dict'' + [STATEDICT ↦ statedict'] in
    (outputtrans, dict''')))
```

type: *Statename*$_0$ (*Graph-node*$_1$ | *Decision-node*$_1$) *Actstmt*$_0^+$ [*Termstmt*$_0$] $\rightarrow$ *Dict* $\rightarrow$ *Transition*$_0$ *Dict*

目的

更新字典 *Dict* 中的状态字典 *Statedict* 以包括隐含于表达式中的进口状态，并且构造导向该隐式状态的跃迁

参数

tnm

是包含表达式的动作所属的状态的名字。该名字保存在进口状态描述符 *StateD* 中，使得在变换跟随此进口状态的跃迁的过程中，用划线表示的下一状态可以被适当地置换。

originalnode$_1$

是包含进口表达式的动作的 AS$_1$ 形式。此节点也被保存在进口状态中(这两个实物构成在引用文字典 *Quotdict* 中定义的 *Importstateinf*)，因为它必须被包括在跟随进口状态的 AS$_1$ 跃迁中。

actl, *term*

是 AS$_0$ 动作和终端符，它们跟随包含进口表达式的动作。跟随 xtREPLY 输入的跃迁包含这两个实物。当进口状态已被变换时，*originalnode*$_1$ 被增加到跃迁中(参见插入进口动作函数 *insert-importact*)。如果此动作包含不止一个进口表达式，则跟随其它进口状态的 xtREPLY 输入的跃迁由一个 xtQUERY 信号的输出和一个下一状态（引向下

一进口状态）组成。

结果

是一个 AS_0 跃迁，它包含与第一进口状态相连系的 xtQUERY 信号的一个输出，还有一个更新过的 *Dict*，以包括隐式进口状态的描述符。

算法

第 1 行　抽取有关出现在表达式中的进口表达式的信息。

第 2 行　抽取有关第一进口表达式的信息。*impnm* 表示替换进口表达式的隐式变量的名字，*qual* 表示在进口表达式中使用的进口变量的限定表 *Qual*，而 *expr* 表示来自进口表达式的 AS_0 pid 表达式。

第 3 行　为隐式状态构造一个名字。

第 4 行　抽取进口变量名。

第 5 行　抽取进口表达式的结果类别的限定表 *Qual*。

第 6 行　分解与结果类别和进口变量名组成的偶对相关的信号名描述符 *SignalnameD*。*xq* 是 xtQUERY 信号的名字，*xr* 是 xtREPLY 信号的名字。

第 7 行　令 *lev* 表示当前的作用域单位，也就是进程、服务或过程。

第 8—9 行　构造包含该 xtQUERY 信号和 PiD 表达式的输出动作。

第 10 行　令 *sigset* 表示必须保存在隐式状态中的信号的限定表 *Qual*。

第 11 行　构造 AS_0 保存节点。

第 13—14 行　构造包含该 xtQUERY 信号的输出的跃迁，该跃迁终止在隐式状态。

第 15 行　把新的进口变量加入到隐式变量集合中，该集合包含在 *Dict* 中的 **IMPLIED** 项目中，并且从包含在 **IMPORTLIST** 项目的表中删除有关进口表达式的信息。

第 17 行　如果这个进口表达式是表达式中最后一个（或唯一的一个），则

第 18 行　构造包含这些动作和终端符的跃迁，该终端符跟随包含此表达式的动作。

第 20 行　构造 xtREPLY 信号的输入。

第 22—23 行　把隐式状态加到此状态字典 *Statedict* 中。*lev* 表明输入节点和保存节点在变换时所在的范围和出现进口表达式的节点的范围相同（请回忆在服务已合并之后，才进行状态变换）。

第 24 行　返回包含输出的跃迁并返回已更新的 *Dict*。

第 26 行　如果还有进口表达式，则构造导向下一进口状态的输出跃迁。并更新字典 *Dict* 以包括其余的那些进口状态且包括新的隐式变量。

第 28 行　构造一个输入，它包含导向下一进口状态的跃迁。

第 28—31 行　更新状态字典 *Statedict* 以包括新的隐式状态。状态描述符 *StateD* 中的 *Import-stateinf* 描述符不包含 AS_1 动作（$originalnode_1$），因为这个状态不是在该动作中由进口表达式隐含的最后一个状态。

第 32 行　返回包含输出的跃迁并返回已更新的 *Dict*。

3.9 服务的扩展

build-service-descriptor(**mk-***Servicedef*$_0$(*id*, *sigset*, *dcll*, *body*, *tid*))(*dict*) ≙ **(3.9.1)**

```
1  (let mk-Id0(q, snm) = id in
2   let squal = dict(SCOPEUNIT) ⌢ ⟨(SERVICE, snm)⟩ in
3   let dict' = dict + [SCOPEUNIT ↦ squal] in
4   if q ≠ ⟨⟩ then
5     exit("§2.4.1: 定义的名字仅仅可以在间接定义中限定")
6    else
7     if tid ∈ {id, nil} then
8       (let sigqualset = transform-validinputset(sigset)(dict') in
9        let (as1 dclset, dict'') = transform-decllist(dcll)(dict') in
10       let (starttrans, labeldict, statedict, priinput) =
11           build-service-statedict(body)(dict) in
12       (as1 dclset, dict'' + [squal ↦ mk-ServiceD(starttrans, labeldict, statedict, sigqualset, priinput)]))
13     else
14      exit("§2.2.2: 在服务定义中的结尾名不同于定义名"))
```

type: *Servicedef*$_0$ → *Dict* → *Decl*$_1$-**set** *Dict*

目的

把在服务定义中包含的定义变换到 AS$_1$，并且构造一个服务描述符，该描述符包含以状态字典 *Statedict* 和标号字典 *Labeldict* 为形式的服务图。

参数

服务定义包含

id	服务的标识符
sigset	服务的有效输入信号集
dcll	局部定义表
body	服务图
tid	结束服务定义的标识符

结果

这些局部定义的 AS$_1$ 定义和一个 *Dict*，它包含服务描述符和局部定义的描述符

算法

第 2—3 行	更新字典 *Dict* 中的作用域单位 **SCOPEUNIT** 以表示此服务。
第 4—7 行	此服务标识符必须是未被限定的（第 4 行），并且如果规定了结束标识符，则它必须与服务标识符相同（第 7 行）。
第 8 行	可以在服务的输入或优先级输入中接收的信号是在有效输入信号集中规定的信号或是在服务信号路由中规定的信号，这些信号与在该服务中定义的定时器汇合。
第 9 行	变换局部定义。
第 10 行	抽取初始跃迁（*starttrans*）；从服务图构造状态字典 *Statedict*（*statedict*）和标号字典 *Labeldict*（*labeldict*）；并抽取有高优先级的信号。
第 12 行	返回局部定义的 AS$_1$ 形式和已更新用以包括它们的描述符与服务描述符的 *Dict*。

build-service-statedict(**mk-***Body*$_0$(*trans*, *statelist*))(*dict*) ≙ (3.9.2)

```
1  (let trans' = insert-trans-term(trans) in
2   let statelist' = ⟨insert-state-term(statelist[i]) | 1 ≤ i ≤ len statelist⟩ in
3   let labeldict = build-trans-labeldict(trans')([]) in
4   let labeldict' = build-state-labeldict(statelist')(labeldict) in
5   let statedict = build-statedict(statelist', dict(SCOPEUNIT))([]) in
6   let statedict' = remove-asterisk-input-and-save(statedict)(dict) in
7   let priinput = extract-priinput(statedict')(dict) in
8   (trans', labeldict', statedict', priinput))
```

type: *Body*$_0$ → *Dict* → *Transition*$_0$ *Labeldict Statedict Signalqual*-**set**

目的 为服务构造状态字典 *Statedict* 和标号字典 *Labeldict*

参数 服务的体包含

trans 初始跃迁

statelist 服务的状态

结果 初始跃迁（其中终端符已插入到判定的回答中）、构造的标号字典 *Labeldict*、构造的状态字典 *Statedict* 和高优先级信号的限定表 *Qual*

算法

第 1 行 在初始跃迁中插入终端符。

第 2 行 在服务的状态中插入终端符。

第 3 行 构造初始跃迁的标号字典 *Labeldict*。

第 4 行 扩展标号字典 *Labeldict* 以包括整个服务体。

第 5 行 构造状态字典 *Statedict*。

第 6 行 从状态字典 *Statedict* 中的状态里删除星号输入和星号保存。

第 7 行 抽取高优先级信号的限定表 *Qual*。

第 8 行 返回初始跃迁、标号字典 *Labeldict*、状态字典 *Statedict* 和高优先级信号。

extract-priinput(*statedict*)(*dict*) ≙ (3.9.3)

```
1  (if statedict = [] then
2     {}
3   else
4    (let stnm ∈ dom statedict in
5     let mk-StateD(speclist, ) = statedict(stnm) in
6     let inputset = {inp | (, inp) ∈ elems speclist ∧ is-Priinput0(inp)} in
7     let inpvars = {inpv | (∃mk-Priinput0(inpl, ) ∈ inputset)(inpv ∈ elems inpl)} in
8     let priset = {signal-qual(id)(dict) |
9                   (∃mk-Inputvars0(id', ) ∈ inpvars)(id' = id)} in
10    priset ∪ extract-priinput(statedict \ {stnm})(dict)))
```

type: *Statedict* → *Dict* → *Signalqual*-**set**

目的 通过检查输入节点从服务的状态抽取高优先级信号

参数 包含服务状态的 *Statedict*

结果 服务的高优先级信号

算法

第 1 行 当处理完 *statedict* 时，返回空集（本函数是递归的）。

第 4 行 在状态字典 *Statedict* 中取出某一状态。

第 5 行 抽取此状态的输入。

第 6 行 从此状态抽取优先级输入节点。

第 7—8 行 从优先级输入中抽取高优先级信号。

第 10 行 返回此高优先级信号，连同来自其余状态的高优先级信号一起返回。

transform-decomposition-body(*dict*) $\triangleq$ (3.9.4)

```
1  (let servicelist = ⟨squal ∈ dom dict | is-local(squal)(dict) ∧
2                                          is-ServiceD(dict(squal))⟩ in
3   if servicelist = ⟨⟩ then
4     exit("§4.10: 一个分解必须至少包含一个服务定义")
5    else
6     (let statetuplemap = merge-states(servicelist, ⟨⟩, ⟨⟩, 1)(dict) in
7      let statetuplenamemap = [statetuple ↦ create-unique-name() |
8                                 statetuple ∈ dom statetuplemap] in
9      let statedict = [statetuplenamemap(tuple) ↦ (statetuplemap(tuple), nil, nil) |
10                      tuple ∈ dom statetuplemap] in
11     let statedict' = double-states(statedict, servicelist)(dict) in
12     let statedict'' = remove-cont-enable-from-statelist(dom statedict')(dict, statedict') in
13     let dict' = dict + [SERVICES   ↦ (statetuplenamemap, servicelist),
14                         STATEDICT ↦ statedict''] in
15     let (firststate, firstservice) = extract-firststate-or-service(servicelist, ⟨⟩, nil)(dict) in
16     if firstservice = nil then
17       (let term1 = mk-Nextstate-node1(name-to-name1(firststate)) in
18        let trans1 = add-service-varinit(mk-Transition1(⟨⟩, term1), elems servicelist)(dict) in
19        let (statebody1, dict'''') = transform-statelist({})(dict') in
20        (mk-Process-graph1(mk-Process-start-node1(trans1), statebody1), dict''''))
21      else
22       (let mk-ServiceD(trans, labeldict, , , ) = dict(firstservice) in
23        let dict'' = dict' + [LABELDICT ↦ labeldict,
24                              SCOPEUNIT ↦ firstservice] in
25        let (trans1, dict''') = transform-transition(firststate, trans, ⟨⟩)(dict'') in
26        let trans1' = add-service-varinit(trans1, elems servicelist)(dict) in
27        let (statebody1, dict'''') = transform-statelist({})(dict''') in
28        (mk-Process-graph1(mk-Process-start-node1(trans1'), statebody1), dict'''')))
```

type: *Dict* → *Process-graph*$_1$ *Dict*

目的 把一个服务分解变换为一个 AS$_1$ 进程图

参数 包含该服务描述符的字典 *Dict*

结果 是 AS$_1$ 进程图和一个字典 *Dict*。该 *Dict* 包含有关用在服务中的输出信号的信息和有关隐式变量的信息（与变换体函数 *transform-body* 相同）。

算法

第 1—2 行 创建由服务的限定表 *Qual* 组成的表。在变换服务时该表是重要的。表中的位置对应

于状态名元组中的位置，该对应关系在 Z.100 的 § 4.10.2 中的变换模型中也提到了。

第 3 行　在一个分解中必须至少存在一个服务。

第 6 行　合并这些服务状态。结果是一特殊的状态字典 *Statedict*，其中各项目是状态名元组而不是状态名。

第 7—8 行　构造一个映射表，它为该特殊状态字典 *Statedict* 中的每个状态名元组定义一个唯一的状态名。

第 9 行　构造一正规的状态字典 *Statedict*，其中的名字元组由相关的唯一名字代替。

第 11 行　构造一新状态字典 *Statedict*，其中每个状态一分为二：接收优先级输入的状态和接收所有其它输入的状态。给第二个状态一个唯一的名字。

第 12 行　从状态字典 *Statedict* 中删除允许条件和连续信号，方法与对正规进程图的做法相同。

第 13 行　把服务表 *Servicelist* 和状态名元组与各个不同的状态名之间的关系放到 *Dict* 中，为状态变换函数所利用。而且也把状态字典 *Statedict* 放到 *Dict* 中，所用方法和对正规进程图的做法一样。

第 15 行　抽取包含初始动作的服务的 *Qual*，或者在所有的服务都不包含初始跃迁的情况下，抽取第一个状态。也就是说，要么第一状态 *firststate* 是 **nil**，要么第一服务 *firstservice* 是 **nil**。

第 16 行　如果没有包含初始跃迁的服务，则

第 17 行　构造导向第一个状态的 AS_1 下一状态。

第 18 行　构造 AS_1 动作表，它把这些服务的变量初始化，并且它的终端就是导向那个第一状态的下一状态。

第 19 行　变换这些状态。

第 20 行　返回进程图（其中初始变迁包含对变量的初始化）和构造的下一状态节点。

第 22 行　如果有一服务包含初始跃迁，则抽取之（*trans*）。另外也抽取标号字典 *Labeldict*，准备在变换初始跃迁时使用。

第 23 行　把标号字典 *Labeldict* 和初始跃迁在其中变换的范围包括到 *Dict* 中，使得变换函数可以利用它们。

第 25 行　变换初始跃迁。

第 26 行　把对服务的变量进行初始化的 AS_1 动作表加入到初始跃迁中。

第 27 行　变换这些状态。

第 28 行　返回进程图。

double-states(*statedict*, *servicelist*)(*dict*) ≜ (3.9.5)

```
 1  (if statedict = [] then
 2     []
 3   else
 4    (let stnm ∈ dom statedict in
 5     let mk-StateD(speclist, ) = statedict(stnm) in
 6     let (priinput, norminput) be s.t. priinput ∪ norminput = elems speclist ∧
 7         (∀(,inp) ∈ priinput)(is-Priinput0(inp)) ∧
 8         (∀(,inp) ∈ norminput)(¬is-Priinput0(inp)) in
 9     if priinput = {} then
10       [stnm ↦ statedict(stnm)] + double-states(statedict \ {stnm}, servicelist)(dict)
11      else
12       (let saveset =
13           {(servicelist[i], mk-Savespec0(siglist)) | i ∈ ind servicelist ∧
14                                                 (let sigset =
15                                                     all-input-signals(servicelist[i])(dict) in
16                                                  siglist = ⟨as0-id(sig) | sig ∈ sigset⟩)} in
17       let qual ∈ elems servicelist in
18       let newstnm = create-unique-name() in
19       let continput = (qual, mk-Contspec0(mk-Id0(⟨⟩, mk-Name0("TRUE", nil)), nil,
20                                           mk-Termstmt0(nil, mk-Nextstate0(newstnm)))) in
21       let firstspec = ⟨spec | spec ∈ (priinput ∪ saveset)⟩ ⌢ ⟨(continput, nil)⟩,
22           secondspec = ⟨spec | spec ∈ norminput⟩ in
23       let firststate = [stnm ↦ mk-StateD(firstspec, nil)],
24           secondstate = [newstnm ↦ mk-StateD(secondspec, (stnm, nil))] in
25       firststate + secondstate ⨪ double-states(statedict \ {stnm}, servicelist)(dict))))
```

type: *Statedict Servicequal*$^+$ → *Dict* → *Statedict*

目的

用两个状态去替换状态字典 *Statedict* 中的每一个状态。其中第一个状态以优先级信号作为输入，第二个状态则以其它信号作为输入。

参数

Statedict	旧状态字典 *Statedict*（本函数是递归的）（其余部分）
servicelist	服务的 *Qual* 的表（参见变换分解函数 *transform-decomposition*）

结果

扩展的状态字典 *statedict*

算法

第 1 行	当全部处理完时，返回空映射表。
第 4 行	取出要处理的下一状态。
第 5 行	令 *speclist* 表示输入表。
第 6—8 行	把输入分为优先级输入和其它输入。
第 9—10 行	如果没有优先级输入，则不修改此状态。
第 12—16 行	构造 *Spec*，其中每一个 *Spec* 包含所有正常信号的保存。每个服务有一个 *Spec*。
第 17 行	从服务表中随意取出某个服务的 *Qual*。该服务用于构造的 *Spec* 中（第 9 行），它包含送往第二个状态的连续信号，即，构造的连续信号是在一个任意的服务的范围中变换的，这是因为连续信号不使用任何与范围有关的标识符。
第 18 行	为第二个状态创建一个唯一的名字。

第 19 行　　构造导向第二个状态的连续信号。在连续信号中的条件为“true”。请注意，在 Z.100 的服务模型中还没有显式地描述连续信号。但是，SAMETOKEN（参见 Z.100 的 §4.11）对应于隐式变量（在 §4.12 中称为“n”），XCONT 对应于隐式信号 emptyq。

第 21 行　　为第一个状态构造 *Speclist*，它包括连续信号。

第 22 行　　为第二个状态构造 *Speclist*。

第 23 行　　构造第一个状态对状态字典 *Statedict* 的贡献。

第 24 行　　构造第二个状态对状态字典 *Statedict* 的贡献。

第 25 行　　继续对其余的状态进行处理。

extract-firststate-or-service(*servicelist*, *statelist*, *firstservice*)(*dict*) ≙ **(3.9.6)**

```
1  (if servicelist = ⟨⟩ then
2     if firstservice ≠ nil then
3       (nil, firstservice)
4      else
5       (let (statetuplenamemap, ) = dict(SERVICES) in
6        (statetuplenamemap(statelist), firstservice))
7    else
8     (let mk-ServiceD(mk-Transition0(actl, term), , stdict, , ) = dict(hd servicelist) in
9      if actl = ⟨⟩ ∧ is-Nextstate0(s-Terminator0(term)) then
10       (let mk-Termstmt0(, mk-Nextstate0(stnm)) = term in
11        if stnm ∈ dom stdict then
12          extract-firststate-or-service(tl servicelist, statelist ⌢ ⟨stnm⟩, firstservice)(dict)
13         else
14          exit("§2.6.7.2.1: 在下一状态中的名字必须表示一个已定义的状态"))
15      else
16       if firstservice ≠ nil then
17         exit("§4.10.2: 不止一个服务包含一个初始跃迁串")
18        else
19         extract-firststate-or-service(tl servicelist, nil, hd servicelist)(dict)))
```

type: *Servicequal** [*Statename*$_0$*] [*Servicequal*] → *Dict* → [*Statename*$_0$] [*Servicequal*]

目的　　抽取状态字典 *Statedict* 中的组合状态的初始状态或抽取含有初始跃迁的服务

参数

servicelist　　由多个服务组成的表（参见变换分解函数 *transform-decomposition*）

statelist　　是在递归调用本函数期间构造的状态名字元组。当 *servicelist* 中全部的服务都已考虑到，该名字元组定义了要返回的唯一状态名（经过状态元组的映射表，参见变换分解函数 *transform-decomposition*）。

firstservice　　如果已经发现一个包含初始跃迁的服务，则 *firstservice* 包含该服务的限定表 *Qual*。只要已经发现这样一个服务，递归就停止，且返回该服务。但是为了检查是否只有一个这样的服务，该服务作为参数传给下一次递归调用。

结果　　如果存在一个含有初始跃迁的服务，则结果为该服务的 *Qual*（*Servicequal*），否则结果为唯一的起始状态名字。

算法

第 1—2 行　　当已经考虑所有服务之后，则如果已经发现一个包含初始跃迁的服务，返回该服务（*firstservice*）的 *Qual*，否则

第 5—6 行　　启动状态就是对应于这些服务中初始的下一状态的那一个状态（*statelist* 包含那些初

始服务状态）。

第 8 行　　分解要考虑的下一服务的描述符。

第 9 行　　如果起始跃迁仅包含一个下一状态，则

第 10 行　　令 *stnm* 表示下一个状态中的状态名。

第 11 行　　如果此状态是在该服务中定义的（即，如果它可在此服务的局部状态字典 *Statedict* 中发现），则

第 12 行　　继续考虑下一个服务。把状态名加到状态名表中，如果没有服务包含一初始跃迁串，就使得新的唯一状态最终会被发现（第 6 行）。

第 16—19 行　　如果还没有发现包含一初始跃迁串的一个服务，则考虑下一个服务。由于包含一初始跃迁的服务已被发现，状态名字表就没有用了（即它是 **nil**）。

add-service-varinit(*trans*, *squalset*)(*dict*) ≙ **(3.9.7)**

```
1  (if squalset = {} then
2     add-varinit(trans, dict)
3   else
4     (let squal ∈ squalset in
5      let dict' = dict + [SCOPEUNIT ↦ squal] in
6      let trans' = add-varinit(trans, dict') in
7      add-service-varinit(trans', squalset \ {squal})(dict)))
```

type: *Transition*$_1$ *Servicequal-set* → *Dict* → *Transition*$_1$

目的　　把 AS$_1$ 的服务变量的初始化加到初始跃迁中

参数

trans　　初始跃迁

squalset　　一个服务分解中各服务的 *Qual* 组成的集合

结果　　更新过的跃迁

算法

第 1 行　　当处理完所有的服务时，则把在进程级上定义的变量的初始化加到跃迁中。

第 4 行　　令 *squal* 表示余下的服务中的一个。

第 5 行　　令字典 *Dict* 中的作用域单位 **SCOPEUNIT** 表示那个服务。

第 6 行　　令 *trans'* 表示一个跃迁，它更新那个服务的变量初始值。当构造附加给进程（或过程）的初始跃迁时，也使用函数 *add-varinit*。

第 7 行　　为其余的服务增加初始化。

merge-states(*servicelist*, *statenmtuple*, *statelist*, *index*)(*dict*) ≙ **(3.9.8)**

```
1  (if index > len servicelist then
2     [statenmtuple ↦ mk-StateD(statelist, nil)]
3   else
4     (let mk-ServiceD(, , statedict, , ) = dict(servicelist[index]) in
5      merge {merge-states(servicelist,
6                          statenmtuple ⌢ ⟨statenm⟩,
7                          statelist ⌢ s-Speclist(statedict(statenm)),
8                          index + 1)(dict) | statenm ∈ dom statedict}))
```

type: *Servicequal*$^+$ *Statename*$_0{}^*$ *Speclist* N_1 → *Dict* → (*Statename*$_0{}^+$ $\overrightarrow{m}$ *StateD*)

目的

把这些服务的状态字典 *Statedict*（"STATE EXPLODE"）合并到一个 *Statedict* 类的映射表中，其中各项是状态名元组（与 Z.100 的 § 4.10.2 模型部分中定义的状态名元组是同一种类的）。在函数 *transform-decomposition-body* 中应用了本函数。前者为每一个元组引入一个唯一的状态名字，从而把该映射表转换到一个普通的状态字典 *Statedict*。

参数

servicelist 这些服务的 *Servicequal* 表，一个服务在该表中的位置等于它的状态在状态名元组中的位置

statenmtuple，
statelist 状态名元组（*statenmtuple*）和结果状态的输入（*statelist*）被递归地创建。当最初应用函数 *mergestates* 时，它们是空的。

index 对 *servicelist* 的标引号，它指明哪一个服务要在本递归调用层上处理

算法

第 1 行 当已经构造好了一个（完整的）状态名元组（*statenmtuple*）时，则返回这一个特别元组的贡献。

第 4 行 令 *statedict* 表示对应于 *index* 的服务的状态字典 *Statedict*。

第 5—9 行 对于该服务，构造一个映射表，它包括所有的状态元组，它以 *statenmtuple* 作为第一个元素（从已被处理过的服务中构造），并且以此服务的一个状态（*statenm*）作为下一个元素。借助合并运算符，把来自服务的各种状态的贡献统一到一个单一的映射表中。

$$is\text{-}wf\text{-}servicesignals(dict) \triangleq \tag{3.9.9}$$

```
1  (let processsignals =
2      {squal | (∃mk-SignalrouteD(p1, p2, s1, s2) ∈ rng dict)
3               ((p1 = dict(SCOPEUNIT) ∧ squal ∈ s2) ∨ (p2 = dict(SCOPEUNIT) ∧ squal ∈ s1))} in
4   (∃d1, d2 ∈ dom dict)(is-ServiceD(dict(d1)) ∧ is-ServiceD(dict(d2)) ∧
5                        get-sur(d1) = dict(SCOPEUNIT) ∧ get-sur(d2) = dict(SCOPEUNIT) ∧
6                        s-Validinputset(dict(d1)) ∩ s-Validinputset(dict(d2)) ≠ {} ⊃
7                        (d1 = d2 ∧ s-Priinputset(dict(d1)) ∩ processsignals = {})))
```

type： *Dict* → *Bool*

目的

检验服务的输入信号应是不相交的，并检验没有优先级信号在包围的进程的外部

结果

如果信号是良形式的，则结果为真。

算法

第 1—3 行 令 *processsignals* 表示包含在信号路由中的信号（隐式的或显式的）。

第 4 行 对于在进程中定义的每两个实体的 *Qual d1* 和 *d2*，下述情况必须成立：如果它们都表示服务（第 4 行），如果它们是在外围进程中定义的（第 5 行），又如果它们具有公共的输入信号（第 6 行），则它们必须表示同一个服务，而且该服务必须没有高优先级信号导向外围进程的环境中（第 7 行）。

transform-servicesigroutedef(**mk-***Sigroutedef*$_0$(*nm*, *path*, *opath*))(*dict*) ≙ **(3.9.10)**

```
 1  (let mk-Sigroutepath0(endpoint1, endpoint2, siglist) = path in
 2   let sigroutequal = dict(SCOPEUNIT) ↷ ⟨(SIGNALROUTE, nm)⟩ in
 3   let sigset = transform-signallist(siglist)(dict) in
 4   let s1 = get-visible-qual(endpoint1, SERVICE)(dict),
 5       s2 = get-visible-qual(endpoint2. SERVICE)(dict) in
 6   let osigset =
 7       if opath = nil then
 8         {}
 9        else
10         (let mk-Sigroutepath0(orig', dest', osiglist) = opath in
11          let s1' = get-visible-qual(orig', SERVICE)(dict),
12              s2' = get-visible-qual(dest', SERVICE)(dict) in
13          if s1 = s2' ∧ s2 = s1' then
14            transform-signallist(osiglist)(dict)
15           else
16            exit("§4.10.1: 在双向信号路由中两个方向的端点必须相匹配 ")) in
17   if s1 ≠ s2 then
18     (let descr = mk-SignalrouteD(s1, s2, sigset, osigset) in
19      ({}, [sigroutequal ↦ descr]))
20    else
21     exit("§4.10.1: 服务的信号路由的两个端点必须不同 "))
```

type: *Sigroutedef*$_0$ → *Dict* → *Decl*$_1$-**set** *Dict*

目的　构造一服务信号路由的 *Dict* 描述符

参数　此服务信号路由包含

nm	信号路由的名字
path	第一条路径
opath	第二条任选的路径

结果　一个 AS_1 定义的空集(因为所有变换 AS_1 定义的函数返回 AS_1 的定义)和包含描述符的对 *Dict* 的贡献

算法

第 1 行	分解在服务信号路由中的第一条路径。
第 2 行	构造表示此服务信号路由的限定表 *Qual*。
第 3 行	构造第一条路径中信号的限定表 *Qual*。
第 4—5 行	构造端点的限定表 *Qual*。
第 6—16 行	如果第二条路径被省略，则 *osigset* 是空集。否则 *osigset* 表示第二条路径中的信号（与 *sigset* 的方向相反）。
第 10 行	分解第二条路径。
第 11—12 行	构造第二条路径中端点的限定表 *Qual*。
第 12—13 行	第一条路径中的第一端点必须等于第二条路径的第二个端点。第一条路径的第二个端点必须等于第二条路径的第一个端点。
第 7 行	在两个端点中必须不指定相同的服务。
第 8 行	构造服务信号路由描述符。

3.10 隐式信道和信号路由的创建

本节包括的函数为一个功能块创建：

- 连接到功能块的隐式信道，条件是此功能块含有出口进程（对于进口进程，则是由包含出口进程的另一端点功能块来创建信道）。
- 连接到该功能块的隐式子信道，条件是此功能块含有进口或出口进程。
- 连接定义，条件是该功能块包含子结构，而此子结构从功能块的环境进口或向环境出口。
- 隐式信号路由和与它们相关的连接定义，这些内容隐含于进/出口和有效输入信号集合中（在 AS_0 中没有指定信号路由的情况下）。
- 要包含在*功能块描述符 BlockD* 中的*进出口信道描述符 ExpimpchanD*，这些描述符包含的信息有：功能块应怎样接口，才可以为另一端点功能块创建隐式子信道。

应用在*变换功能块函数 transform-block* 中的入口函数是 *implicit-channels-and-signalroutes*。

implicit-channels-and-signal-routes(*decllist*)(*dict*) $\triangleq$ (3.10.1)

```
1  (let (exportset, importset) = imp-exp-in-processes(dict) in
2   let sigroutes = implicit-signalroutes(decllist)(dict),
3       blockimport = all-importeurs(dict),
4       blockexporteurs = all-exporteurs(dict),
5       (fatherimpm, fatherexpm) = father-block-import-export(dict) in
6   let exporttoenv = {closure | closure ∈ (exportset ∩ dom fatherexpm)},
7       importfromblock = {closure | closure ∈ (importset ∩ dom blockexporteurs)},
8       importfromenv = {closure | closure ∈ (importset ∩ dom fatherimpm)} in
9   let importblockmap = [closure ↦ blockexporteurs(closure) | closure ∈ importfromblock],
10      importenvmap = [cl ↦ ⟨(ENV, create-unique-name()) | 1 ≤ i ≤ len fatherimpm⟩ |
11                       cl ∈ importfromenv],
12      exportblockmap = [cl ↦ ⟨(bqual, create-unique-name()) | cl ∈ (exportset ∩ blockimport(bqual))⟩ |
13                        bqual ∈ dom blockimport],
14      exportenvmap = [cl ↦ ⟨(ENV, create-unique-name()) | 1 ≤ i ≤ len fatherexpm⟩ |
15                      cl ∈ exporttoenv] in
16  let importenvchannels = as0-channels(importenvmap, false)(dict),
17      exportblockchannels = as0-channels(exportblockmap, true)(dict),
18      exportenvchannels = as0-channels(exportenvmap, true)(dict) in
19  let connects = implicit-connects(importblockmap, exportblockmap)(dict) in
20  let decll = importenvchannels ↷ exportblockchannels ↷ exportenvchannels in
21  (decll, sigroutes, connects, importblockmap, exportblockmap))
```

type: $Decl_0^*$ → *Dict* → $Decl_0^*$ $Decl_0^*$ $Decl_0^*$ *ExpimpchanD* *ExpimpchanD*

目的

为一功能块构造隐式信道、连接定义和信号路由

参数

decllist

此功能块的定义表，它用于测试在此功能块中是否存在显式信号路由

结果

结果包括：要在和功能块相同的层次上定义的信道、要在功能块定义的信号路由和信号路由的连接、要在子结构（如果有）定义的功能块子结构的连接、隐式信道名与由它传递的进口信号之间的关系（信道名是从为出口功能块定义的*进出口信道描述符 ExpimpchanD* 中推断出来的）和隐式出口信号之间的关系。

算法

第 1 行　令 *exportset* 表示功能块中各进程的类别 *Qual* 和出口名字（*NameclosureD*）组成的偶对的集合，令 *importset* 表示与该功能块的出口对应的名字闭包描述符 *NameclosureD*。

第 2 行　令 *sigroutes* 表示为此功能块创建的隐式信号路由。

第 3 行　令 *blockimport* 表示功能块的 *Qual* 和用于功能块进口名字的名字闭包描述符 *NameclosureD* 之间的关系。

第 4 行　令 *blockexporteurs* 表示进出口信道描述符 *ExpimpchanD*，它包含所有其它功能块的名字和它们的出口名字。

第 5 行　令 *fatherimpm* 表示包含本功能块的功能块的进口 *ExpimpchanD* 描述符。令 *fatherexpm* 表示包含本功能块的功能块的出口 *ExpimpchanD* 描述符。

第 6 行　从此功能块出口到此作用域单位的环境的名字的 *NameclosureD* 是从该功能块出口的名字的 *NameclosureD*（*exportset*）和在外围功能块出口的名字的 *NameclosureD* 之间的交集。

第 7 行　从另一个功能块进口到本功能块的名字的 *NameclosureD* 是从本功能块进口的名字的 *NameclosureD*（*importset*）和在另一功能块出口的名字的 *NameclosureD* 之间的交集。

第 8 行　从此作用域单位的环境进口到此功能块的名字的 *NamecloureD* 是从本功能块进口的名字的 *NameclosureD*（*importset*）和在外围功能块出口的名字的 *NameclosureD* 之间的交集。

第 9 行　构造 *ExpimpchanD* 描述符，该描述符包含有关传递进口信号名字的信道名字的信息。这些信道名字是从包含在出口功能块的 *BlockD* 中的描述符推断出来的。

第 10—11 行　构造进出口信道描述符 *ExpimpchanD*，该描述符包含了有关子信道名字的信息，该子信道名字把进口信号名传递到本功能块的环境。在这种情况下，要创建信道名，因为另一端点是 ENV。

第 12—13 行　构造进出口信道描述符 *ExpimpchanD*，该描述符包含了有关信道名字的信息，该信道名字把出口信号名传递到其它功能块（在同一作用域单位中）。

第 14—15 行　构造进出口信道描述符 *ExpimpchanD*，该描述符包含有关子信道名字的信息，该子信道名字把出口信号名传递到本作用域单位的环境。

第 16 行　为功能块中的进口名字构造 AS_0 子信道。

第 17 行　为出口到另一功能块的名字构造 AS_0 信道。

第 18 行　为出口到外围作用域单位的名字构造 AS_0 信道。

第 19 行　为功能块的子结构中所有隐式子信道构造连接定义。

第 20 行　令 *decll* 表示要被定义在定义功能块的同一层上的那些定义，即全部所构造的信道。

第 21 行　返回这些信道的定义、信号路由定义和这两个 *ExpimpchanD* 映射表。

3.10.1 隐式信道的创建

$as_0\text{-}channels(map, isexport)(dict) \triangleq$ (3.10.1.1)

```
1  (if map = [] then
2     ⟨⟩
3   else
4    (let clos ∈ dom map in
5     ⟨as0-channeldefs(map(clos), isexport, dict(clos))(dict)⟩ ⌢
6     as0-channels(map \ {clos}, isexport)(dict)))
```

type: $ExpimpchanD\ Bool \rightarrow Dict \rightarrow Chandef_0^*$

目的

构造隐式信道，这些信道或者从一个个功能块的环境进口，或者向一功能块的环境出口，或者从一功能块向另一功能块出口

参数

map 是一个进出口信道描述符 *ExpimpchanD*，它包含有关信道名和一个端点名字的信息。第二个端点就是当前的功能块，即根据 *dict*（**SCOPEUNIT**）来构造。

isexport 为真，如果要构造当前功能块中出口名字的信道的话。

结果

构造的 AS_0 信道定义表

算法

第 1 行 当处理完所有进口名字或出口名字时，则返回空表。

第 4 行 为一个进口名字或出口名字取出下一个名字闭包描述符 *NameclosureD*。

第 5 行 为该进口或出口名字构造信道定义。

第 6 行 为其余的名字闭包描述符 *NameclosureD* 构造信道定义。

$as_0\text{-}channeldefs(list, isexport, signalnames)(dict) \triangleq$ (3.10.1.2)

```
1  (if list = ⟨⟩ then
2     ⟨⟩
3   else
4    (let level = dict(SCOPEUNIT),
5         blkid = as0-id(level),
6         mk-SignalnamesD(, xq, xr) = signalnames in
7     let signallist1 = ⟨mk-Id0(⟨⟩, xr)⟩,
8         signallist2 = ⟨mk-Id0(⟨⟩, xq)⟩,
9         (bqual, chan) = hd list in
10    let blkid' = as0-id(bqual) in
11    let (fromblock, toblock) = if isexport then (blkid, blkid') else (blkid', blkid) in
12    let direction1 = mk-Chanpath0(fromblock, toblock, signallist1),
13        direction2 = mk-Chanpath0(toblock, fromblock, signallist2) in
14    ⟨mk-Chandef0(chan, direction1, direction2, nil, chan)⟩ ⌢
15    as0-channeldefs(tl list, isexport, signalnames)(dict)))
```

type: $(Otherend\ Channame_0)^*\ Bool\ SignalnamesD \rightarrow Dict \rightarrow Chandef_0^*$

目的

对于一个给定的信号名描述符 *signalnamesD*，构造信道的两个端点、信道名字以及传递 xtREPLY 信号和 xtQUERY 信号的信道定义

参数

list 是偶对表，每个偶对由另一个端点（不是当前的端点）和信道名组成。偶对表定义了要创建的信道。

isexport 如果当前功能块就是出口的功能块，则 *isexport* 为真。

signalnames 是信号名描述符 *SignalnamesD*，它包含对应于某个名字闭包描述符 *NameclosureD* 的 xtREPLY 和 xtQUERY 信号。

结果 AS_0 信道定义表

算法

第 1 行 当全部处理完时，返回空表（本函数是递归的）。

第 4—14 行 为表中的下一个项构造一信道定义。

第 4 行 令 *level* 表示当前功能块的 *Qual*。

第 5 行 令 *blkid* 表示当前功能块的标识符。

第 6 行 令 *xq* 和 *xr* 分别表示要由信道传递的 xtQUERY 信号和 xtREPLY 信号。

第 7—8 行 构造包含这两个信号的两个信号表。

第 9 行 令 *bqual* 表示另一端功能块的 *Qual*，并令 *chan* 表示此信道名。

第 10 行 构造另一端的标识符。

第 11 行 如果当前功能块是出口者，则 *fromblock* 表示该功能块，否则 *fromblock* 表示另一端点功能块（在子信道情况下）。

第 12—13 行 为这两个信号表构造路径。

第 14 行 构造此信道定义。

第 15 行 为此表（*list*）的其余部分构造信道定义。

father-block-import-export(*dict*) $\triangleq$ (3.10.1.3)

```
1  (let level = dict(SCOPEUNIT) in
2   if len level > 2 then
3     (let fatherlevel = get-sur(get-sur(level)) in
4      let mk-BlockD(exp, imp, , ) = dict(fatherlevel) in
5      (exp, imp))
6    else
7     ([], []))
```

type: *Dict* → *ExpimpchanD ExpimpchanD*

目的 为包含当前功能块的功能块抽取两个进出口信道描述符 *ExpimpchanD*

结果 用于出口信道的 *ExpimpchanD* 和用于进口信道的 *ExpimpchanD*

算法

第 1 行 令 *level* 表示当前功能块的限定表 *Qual*。

第 2 行 如果此功能块包含在一个子结构中，则

第 3 行 令 *fatherlevel* 表示包含此子结构的父辈功能块的限定表 *Qual*。

第 4—5 行 返回该父辈功能块的进出口信道描述符 *ExpimpchanD*。

第 7 行 如果当前功能块被定义在系统层，则这两个映射表是空的，因为从（向）系统的环

境进口（出口）是不可能的。

imp-exp-in-processes(*dict*) ≜ **(3.10.1.4)**

```
 1  (let level = dict(SCOPEUNIT) in
 2   let pquals = {qual | qual ∈ dom dict ∧
 3                        is-ProcessD(dict(qual)) ∧
 4                        get-sur(qual) = level} in
 5   if pquals = {} then
 6     (let subblockquals = {qual | qual ∈ dom dict ∧
 7                                  is-BlockD(dict(qual)) ∧
 8                                  get-sur(get-sur(qual)) = level} in
 9      let eiset = {imp-exp-in-processes(dict + [SCOPEUNIT ↦ qual]) | qual ∈ subblockquals} in
10      (union {imp | (imp, ) ∈ eiset}, union {exp | (, exp) ∈ eiset}))
11    else
12     (let eset = {vqual | vqual ∈ dom dict ∧
13                          is-VarD(dict(vqual)) ∧
14                          get-sur(vqual) ∈ pquals},
15          iset = {vqual | vqual ∈ dom dict ∧ is-ImportD(dict(vqual)) ∧
16                          get-sur(vqual) ∈ pquals} in
17      let expset =
18          {closure | (∃q ∈ eset)
19                     ((let mk-VarD(t, , exp, , ) = dict(q),
20                           (, vnm) = q[len q] in
21                       exp = EXPORTED ∧ closure = (t, vnm)))},
22          impset =
23          {closure | (∃q ∈ iset)
24                     ((let mk-ImportD(t) = dict(q),
25                           (, vnm) = q[len q] in
26                       closure = (t, vnm)))} in
27      (expset, impset)))
```

type: *Dict* → *NameclosureD*-**set** *NameclosureD*-**set**

目的 为出口变量和为进口变量抽取由类别和变量名组成的偶对（*NameclosureD*）。这些变量是在一个功能块里的进程中定义的，或者如果此功能块不含进程，则这些变量是在功能块的子结构中的进程中定义的。

结果 出口变量的名字闭包描述符 *NameclosureD* 和进口变量的名字闭包描述符 *NameclosureD*

算法

第 1 行 令 *level* 表示当前功能块的限定表 *Qual*。

第 2—4 行 令 *pqual* 表示在功能块中定义的进程的限定表 *Qual*。

第 5 行 如果功能块中没有进程，则

第 6—8 行 令 *subblockqual* 表示包含在子结构中的功能块的限定表 *Qual*。

第 9 行 为子功能块的出口变量和进口变量构造名字闭包描述符 *NameclosureD* 偶对的集合。

第 10 行 把集合中的所有出口 *NameclosureD* 汇合，又把集合中所有进口 *NameclosureD* 汇合。

第 12—14 行 如果功能块中包含进程，则令 *eset* 表示在进程内定义的变量的 *Qual* 集合。

第 15—16 行 令 *iset* 表示在进程中定义的进口变量的 *Qual* 集合。

第 17—21 行 令 *expset* 表示出口变量的名字闭包描述符 *NameclosureD* 集合，即，类别（*t*）和出口的（*exp*=**EXPORTED**）变量名（*vnm*）的偶对的集合。

第 22—26 行 令 *impset* 表示进口变量的名字闭包描述符 *NameclosureD* 集合。

第 27 行　　返回这两个集合。

all-importeurs(*dict*) ≙ **(3.10.1.5)**

```
1  (let level = dict(SCOPEUNIT) in
2   let sur = get-sur(level) in
3   let blockqset = {qual | qual ∈ dom dict ∧
4                          is-BlockD(dict(qual)) ∧
5                          get-sur(qual) = sur} \ {level} in
6   let expimpmap = [q ↦ imp-exp-in-processes(dict + [SCOPEUNIT ↦ q]) | q ∈ blockqset] in
7   [q ↦ closset | q ∈ dom expimpmap ∧ (, closset) = expimpmap(q)])
```

type: *Dict* → (*Blockqual* ~~m~~→ *NameclosureD*-**set**)

目的　　抽取有关进口功能块和用于进口的 *NameclosureD* 的关系的信息。当前功能块不包括在这个关系（映射表）中。

算法

第 1 行　　令 *level* 表示当前功能块。

第 2 行　　令 *sur* 表示外围功能块子结构的 *Qual*。

第 3—5 行　　构造由子结构中所有功能块（当前功能块除外）组成的集合。

第 6 行　　构造一个映射表，从进口功能块映射到功能块的 *NameclosureD* 偶对的集合。

第 7 行　　返回一个映射表，从进口功能块映射到功能块中进口变量的 *NameclosureD*。

all-exporteurs(*dict*) ≙ **(3.10.1.6)**

```
1  (let level = dict(SCOPEUNIT) in
2   let sur = get-sur(level) in
3   let blockqset = {qual | qual ∈ dom dict ∧
4                          is-BlockD(dict(qual)) ∧
5                          get-sur(qual) = sur} \ {level} in
6   get-exporteurs(blockqset)(dict))
```

type: *Dict* → *ExpimpchanD*

目的　　构造（组合的）进出口信道描述符 *ExpimpchanD*，它包括所有其它出口功能块（当前功能块除外）的出口变量

算法

第 1 行　　令 *level* 表示当前功能块的限定表 *Qual*。

第 2 行　　令 *sur* 表示外围功能块子结构的限定表 *Qual*。

第 3—5 行　　令 *blockqset* 表示该子结构中所有其它功能块的 *Qual* 的集合。

第 6 行　　递归地遍历该 *Qual* 集合，在这个过程中汇合这些功能块的 *ExpimpchanD*。

get-exporteurs(*blockqset*)(*dict*) ≜ (3.10.1.7)

```
1  (if blockqset = {} then
2     []
3   else
4    (let q ∈ blockqset in
5     let blkqual = dict(SCOPEUNIT) in
6     let mk-BlockD(expm,,,) = dict(q) in
7     let expm' = [closure ↦ tup | closure ∈ dom expm ∧
8                                  (let tup' = expm(closure) in
9                                   if (∃i ∈ ind tup')(s-Otherend(tup[i]) = blkqual) then
10                                    (let i ∈ ind tup' be s.t. s-Otherend(tup[i]) = blkqual in
11                                     tup = ⟨tup[i]⟩)
12                                   else
13                                    false)] in
14    let rest = get-exporteurs(blockqset \ {q})(dict) in
15    [clos ↦ (rest + expm')(clos) | clos ∈ (dom rest − dom expm')] +
16    [clos ↦ newlist | clos ∈ (dom rest ∩ dom expm') ∧ newlist = rest(clos) ⌢ expm'(clos)]))
```

type: *Blockqual*-set → *Dict* → *ExpimpchanD*

目的

构造组合的进出口信道描述符 *ExpimpchanD*，它包括所有功能块中出口的信息

参数

blockgset 要处理的（余下的）功能块的集合（本函数是递归的）

结果

构造出来的组合的进出口信道描述符 *ExpimpchanD*

算法

第 1 行 当已处理了所有功能块后，返回空的映射表。

第 4 行 令 *q* 表示集合中下一功能块的限定表 *Qual*。

第 5 行 令 *blkqual* 表示当前功能块的限定表 *Qual*。

第 6 行 令 *expm* 表示对应于功能块出口的 *ExpimpchanD*。

第 7—13 行 限制 *ExpimpchanD* 映射表，使得它仅包括那些元素，其另一端点是当前的功能块。

第 14 行 为其余功能块构造组合的 *ExpimpchanD*。

第 15—16 行 返回进出口信道描述符 *ExpimpchanD*，其中两个映射表中的表（*expm'* 和 *rest*）已被拼接。第 15 行定义没有改变的项目，而第 16 行定义由拼接映射表中两个项的表构造的项目。

3.10.2 隐式信号路由的创建

implicit-signalroutes(*decll*)(*dict*) ≙ (3.10.2.1)

```
1  (let existexplicit = (∃d ∈ decll)(is-Sigroutedef0(d)) in
2   if ¬existexplicit ∧ (∃mk-Prdef0(, inst, , , , , ) ∈ elems decll)(inst = nil) then
3     exit("§2.5.2: 当没有指定信号路由时，有效输入信号集必须加以指定")
4    else
5     (let bqual = dict(SCOPEUNIT) in
6      let cqualset =
7          {cqual ∈ dom dict | is-ChannelD(dict(cqual)) ∧
8                              (let mk-ChannelD(endp1, endp2, , ) = dict(cqual) in
9                               bqual ∈ {endp1, endp2})} in
10     let pathset = {p ∈ dom dict | is-ProcessD(dict(p)) ∧ is-local(p)(dict)} in
11     let pathpairset ∈ (Processqual-set)-set be s.t. (∀e ∈ pathpairset)(card e = 2) ∧
12         union pathpairset = pathset ∧
13         (∀p1, p2 ∈ pathset)(p1 ≠ p2 ⊃ {p1, p2} ∈ pathpairset) in
14      make-as0-local-signalroutes(pathpairset, existexplicit)(dict) ⌒
15      make-as0-env-signalroutes(cqualset, existexplicit)(dict)))
```

type: $Decl_0^* \rightarrow Dict \rightarrow Decl_0^*$

目的　　为功能块构造 AS_0 隐式信号路由

算法

第 1 行　　如果在功能块中规定了信号路由，则令 *existexplicit* 为真。

第 2—3 行　　如果在功能块中没有规定显式信号路由，则全部所包含的进程定义必须包含一个有效输入信号集。

第 5 行　　抽取功能块的限定表 *Qual*。

第 6—9 行　　抽取那些以该功能块作为一个端点的信道的描述符。

第 10 行　　构造一个集合，它包含功能块中所有进程的 *Qual*。

第 11—13 行　　构造一 *Processqual* 集合的集合，其中每个所包含的集合包括两个元素（第 11 行）。这些集合仅包含在该功能块中定义的那些进程的 *Qual*（第 12 行），而这个集合包含这些集合的所有可能的组合。

第 14 行　　构造功能块中进程之间的信号路由，并且

第 15 行　　构造与该功能块的环境相连的信号路由。

make-as₀-local-signalroutes(*pathset*, *existexplicit*)(*dict*) ≙ **(3.10.2.2)**

```
1  (if pathset = {} then
2     ⟨⟩
3   else
4    (let {p1, p2} ∈ pathset in
5     let mk-ProcessD(, insig1, outsig1, ) = dict(p1),
6         mk-ProcessD(, insig2, outsig2, ) = dict(p2) in
7     let (insig1', outsig1') = if existexplicit then
8                                 import-export-signals(dict)
9                               else
10                                (insig1, outsig1),
11        (insig2', outsig2') = if existexplicit then
12                                import-export-signals(dict)
13                              else
14                                (insig2, outsig2) in
15    let as₀p1id = as₀-id(p1),
16        as₀p2id = as₀-id(p2) in
17    let (com1sig, com2sig) = (insig1' ∩ outsig2', insig2' ∩ outsig1') in
18    let as₀sigset1 = {as₀-id(id) | id ∈ com1sig},
19        as₀sigset2 = {as₀-id(id) | id ∈ com2sig} in
20    let com1 = if com1sig = {}
21                 then nil
22                 else mk-Sigroutepath₀(as₀p1id, as₀p2id, as₀sigset1),
23        com2 = if com2sig = {}
24                 then nil
25                 else mk-Sigroutepath₀(as₀p2id, as₀p1id, as₀sigset2) in
26    let (com', com'') = if com1 = nil then (com2, com1) else (com1, com2) in
27    let route₀ =
28        if com' = nil ∧ com'' = nil
29          then ⟨⟩
30          else (let nm₀ = create-unique-name() in
31                ⟨mk-Sigroutepath₀(nm₀, com', com'')⟩) in
32    route₀ ⌢ make-as₀-local-signalroutes(pathset \ {{p1, p2}}, existexplicit)(dict)))
```

type: (*Processqual*-**set**)-**set** *Bool* → *Dict* → *Sigroutedef*₀*

目的　　构造把一功能块中的进程连接起来的 AS_0 信号路由

参数

pathset	进程的偶对（包含两个元素的集合）的集合
existexplicit	如果为此功能块规定了 AS_0 信号路由，则 *existexplicit* 为真。

结果　　一组 AS_0 信号路由组成的表

算法

第 1 行	当全部处理完时，返回空表（本函数是递归的）。
第 4 行	令 *p*1 和 *p*2 表示要处理的进程的一个组合。
第 5—6 行	分解这两个进程的描述符。
第 7—11 行	如果为进程规定了 AS_0 信号路由，则仅为进（出）口信号构造信号路由。
第 15—16 行	为这两个进程构造 AS_0 标识符。
第 17 行	令 *com*1*sig* 表示从 *p*2 导向 *p*1 的信号，令 *com*2*sig* 表示从 *p*1 导向 *p*2 的信号。
第 18 行	构造对应于 *com*1*sig* 的 AS_0 标识符集合。
第 19 行	构造对应于 *com*2*sig* 的 AS_0 标识符集合。

第 20—25 行　　令 *com*1 和 *com*2 分别表示从 *p*2 到 *p*1 和从 *p*1 到 *p*2 的 AS_0 通信路径（如果一条路径中不含信号，则其值为 **nil**）。

第 26—31 行　　令 $route_0$ 表示一个信号路由，它包含两进程之间的信号。

第 32 行　　把此信号路由定义和其余的进程偶对的那些信号路由定义汇合到一起。

make-as$_0$*-env-signalroutes*(*cqualset*, *existexplicit*)(*dict*) ≙ **(3.10.2.3)**

```
 1  (if cqualset = {} then
 2     ⟨⟩
 3    else
 4     (let bqual = dict(SCOPEUNIT) in
 5      let cqual ∈ cqualset in
 6      let pqualset = {pqual | pqual ∈ dom dict ∧ is-local(pqual)(dict)} in
 7      let sigroutes = make-as0-channel-signalroutes(cqual, pqualset, existexplicit)(dict) in
 8      if sigroutes = {} then
 9        make-as0-env-signalroutes(cqualset \ {cqual}, existexplicit)(dict)
10       else
11        (let as0chanid = as0-id(cqual),
12             routenameset = {nm | (∃mk-Sigroutedef0(nm',,) ∈ sigroutes)(nm' = nm)} in
13         let as0list = ⟨as0-id(bqual ⌢ ⟨(SIGNALROUTE, nm)⟩) | nm ∈ routenameset⟩ in
14         ⟨sigroute | sigroute ∈ sigroutes⟩ ⌢
15         ⟨mk-Connect0(as0chanid, as0list)⟩ ⌢
16         make-as0-env-signalroutes(cqualset \ {cqual}, existexplicit)(dict))))
```

type: *Channelqual*-**set** *Bool* → *Dict* → $Decl_0^*$

目的　　构造与功能块环境相连的隐式 AS_0 信号路由

参数

cqualset　　连接到此功能块的信道集合

existexplicit　　其值为真，条件是在功能块的具体语法中规定了信号路由。

结果　　构造的 AS_0 信号路由表和适合的连接定义

算法

第 1 行　　当全部处理完时，返回空表（本函数是递归的）。

第 4 行　　令 *bqual* 表示功能块的限定表 *Qual*。

第 5 行　　取出要处理的下一个信道的限定表 *Qual*。

第 6 行　　令 *pqualset* 表示此功能块中各进程的 *Qual* 的集合。

第 7 行　　构造连接到此信道的信号路由。

第 8 行　　如果没有信号路由连接到此信道（这是一种错误，但是当把信号路由变换到 AS_1 时才得到检查）或该信道已存在信号路由，则继续考虑其余的信道。

第 11 行　　构造此信道的 AS_0 标识符。

第 12 行　　抽取构造的信号路由的名字。

第 13 行　　把这些名字变为标识符。

第 14 行　　返回信号路由和

第 15 行　　它们相关的连接定义和

第 16 行　　对应于其余信道的信号路由和连接。

make-as$_0$-channel-signalroutes(*cqual*, *pqualset*, *existexplicit*)(*dict*) ≙ (3.10.2.4)

```
1  (if pqualset = {} then
2     ⟨⟩
3    else
4     (let mk-ChannelD(end1, , sigset1, sigset2, ) = dict(cqual) in
5      let pqual ∈ pqualset in
6      let mk-ProcessD(, insig, outsig, ) = dict(pqual) in
7      let (insig', outsig') = if existexplicit then
8                                 import-export-signals(dict)
9                              else
10                             (insig, outsig) in
11     let bqual = dict(SCOPEUNIT) in
12     let (fromprocesssignals, toprocesssignals) = if bqual = end1 then
13                                                    (sigset1 ∩ outsig', sigset2 ∩ insig')
14                                                  else
15                                                    (sigset2 ∩ outsig', sigset1 ∩ insig') in
16     if fromprocesssignals ∪ toprocesssignals = {} then
17       make-as0-channel-signalroutes(cqual, pqualset \ {pqual}, existexplicit)(dict)
18      else
19       (let routenm = create-unique-name() in
20        let as0pid = as0-id(pqual) in
21        let as0fromsignals = {as0-id(id) | id ∈ fromprocesssignals},
22            as0tosignals = {as0-id(id) | id ∈ toprocesssignals} in
23        let path1 = if fromprocesssignals = {}
24                    then nil
25                    else mk-Sigroutepath0(as0pid, ENV, as0fromsignals),
26            path2 = if toprocesssignals = {}
27                    then nil
28                    else mk-Sigroutepath0(ENV, as0pid, as0tosignals) in
29        let (path', path'') = if path1 = nil then (path2, path1) else (path1, path2) in
30        ⟨mk-Sigroutedef0(routenm, path', path'')⟩ ⌢
31        make-as0-channel-signalroutes(cqual, pqualset \ {pqual}, existexplicit)(dict))))
```

type: *Channelqual Processqual*-**set** *Bool* → *Dict* → *Sigroutedef*$_0$*

目的　　构造对应于给定信道的隐式信号路由

参数

cqual	信道的限定表 *Qual*
pqualset	功能块中各进程的 *Qual* 组成的集合
existexplicit	为真，条件是在功能块中规定了信号路由。

结果　　构造的 AS_0 信号路由表

算法

第 1 行	当全部处理完时，返回空集（本函数是递归的）。
第 4 行	令 *end*1 表示发送信号 *sigset*1 和接收信号 *sigset*2 的信道的端点。
第 5 行	令 *pqual* 表示要考虑的下一进程。
第 6 行	令 *insig* 表示进程的输入信号，令 *outsig* 表示进程的输出信号。
第 7 行	如果具体语法中规定了信号路由，则隐式信号路由仅为隐含于进口（出口）中的那些信号路由。
第 11 行	令 *bqual* 表示外围功能块的限定表 *Qual*。
第 12—15 行	令 *fromprocesssignals* 表示从进程经由信道发送的信号，令 *toprocesssignals* 表示经过此信道接收的信号。

第 16—17 行　　如果没有通过信道发送或接收的信号，则继续考虑其余的进程。
第 19 行　　为要构造的信号路由创建一隐式名字。
第 20 行　　构造进程的 AS_0 标识符。
第 21 行　　把输出信号变换为 AS_0 信号标识符。
第 22 行　　把输出信号变换为 AS_0 信号标识符。
第 23—25 行　　如果从进程通过信道有发送的信号，则构造适当的 $Sigroutepath_0$。
第 26—28 行　　如果从进程通过信道有接收的信号，则构造适当的 $Sigroutepath_0$。
第 29 行　　令 *path'* 表示包含信号的路径（第一条路径），令 *path''* 表示第二条任选的路径。
第 30 行　　构造并返回与信道连接的一条信号路由，并为其余的进程构造与该信道相连的信号路由。

3.10.3 隐式连接语句的创建

implicit-connects(*exp*, *imp*)(*dict*) ≙ **(3.10.3.1)**

```
1  (let level = dict(SCOPEUNIT) in
2   let expset =
3       {expmap | (∃qual ∈ dom dict)
4                 (is-BlockD(dict(qual)) ∧
5                  get-sur(qual) = level ∧
6                  dict(qual) = mk-BlockD(expmap,,,))} in
7   let impset =
8       {impmap | (∃qual ∈ dom dict)
9                 (is-BlockD(dict(qual)) ∧
10                 get-sur(qual) = level ∧
11                 dict(qual) = mk-BlockD(,impmap,,))} in
12  let outconnect =
13      if exp = [] then
14        ⟨⟩
15       else
16        (let closure ∈ dom exp in
17         gen-connect(expset, closure, exp(closure), exp \ {closure})),
18      inconnect =
19      if imp = [] then
20        ⟨⟩
21       else
22        (let closure ∈ dom imp in
23         gen-connect(impset, closure, imp(closure), imp \ {closure})) in
24  outconnect ↷ inconnect)
```

type: *ExpimpchanD ExpimpchanD* → *Dict* → $Connect_0^*$

目的　　为由进口（出口）隐含的隐式信道构造 AS_0 信道连接

参数

exp　　功能块的 *ExpimpchanD* 的出口映射表，该功能块包含与隐式子信道关联的功能块
imp　　功能块的 *ExpimpchanD* 的进口映射表，该功能块包含与隐式子信道关联的功能块

结果　　AS_0 信道连接表

算法

第 1 行　　令 *level* 表示当前功能块子结构的限定表 *Qual*。

第 2—6 行　　构造由出口 *ExpimpchanD* 映射表组成的集合。子结构中的每个子功能块都有一个映射表。

第 7—11 行　　构造由进口 *ExpimpchanD* 映射表组成的集合。子结构中的每个子功能块都有一个映射表。

第 12—17 行　　为从子结构出口中隐含的信道构造信道连接。*closure* 表示要处理的第一个 *NameclosureD*。

第 18—23 行　　为向子结构进口中隐含的信道构造信道连接。*closure* 表示要处理的第一个 *NameclosureD*。

gen-connect(*mapset*, *closure*, *list*, *map*) ≜ **(3.10.3.2)**

```
1  (if list = ⟨⟩ then
2     if map = [] then
3        ⟨⟩
4      else
5       (let closure' ∈ dom map in
6         gen-connect(mapset, closure', map(closure'), map \ {closure'}))
7    else
8     (let (mapset', chanlist) = gen-connect-elem(mapset, closure, ⟨⟩) in
9      let (,channame) = hd list in
10     let connect = mk-Connect0(channame, chanlist) in
11     ⟨connect⟩ ⌢ gen-connect(mapset', closure, tl list, map)))
```

type: *ExpimpchanD*-**set** *NameclosureD* (*Otherend Channame*$_0$)* *ExpimpchanD* → *Connect*$_0^+$

目的　　构造信道连接。这些信道连接是向（从）一个功能块子结构进口（出口）一个变量时隐含的。

参数

expimpset　　功能块子结构中的功能块的出口 *ExpimpchanD* 或进口 *Expimp chanD* 的集合，即关于在子结构中出口或进口的信息

closure　　要处理的名字闭包描述符 *NameclosureD*

list　　是一个表，包含一连串与子结构相连的隐式信道，这些信道传递对应于名字闭包描述符 *NameclosureD* (*closure*) 的隐式信号。

map　　该子结构的其余的要被处理的名字闭包描述符 *namecloureD* 的 *ExpimpchanD*

结果　　AS_0 信道连接表

算法

第 1 行　　当处理完与 *closure* 对应的连接到功能块子结构的信道时，则

第 2 行　　如果已没有 *NameclosureD* 要处理，则返回。否则

第 5 行　　令 *closure* 表示下一个要处理的 *NameclosureD*。

第 6 行　　为对应于 *closure* 的隐式子信道创建信道连接。并且为其余的 *NamecloureD* 创建信道连接。

第 8 行　　为子功能块更新 *ExpimpchanD* 映射表的集合 (*mapset'*)，以便排除一些项目，这些项目对应于导向子结构环境的 *closure*。还要构造将在信道连接中使用的 AS_0 子信道标识符。

第 9 行　　令 *channame* 表示与子结构连接的隐式信道的名字。
第 10 行　　构造信道连接。
第 11 行　　把此信道与其余隐式子信道的信道连接汇合到一起。

gen-connect-elem(*mapset*, *closure*, *chanlist*) ≙ **(3.10.3.3)**

```
1  (if mapset = {} then
2     ({},chanlist)
3    else
4     (let map ∈ mapset in
5      let (mapset', chanlist') = gen-connect-elem(mapset \ {map}, closure, chanlist) in
6      let clist = map(closure) in
7      if (∃index ∈ ind clist)(s-Otherend(clist[index]) = ENV) then
8        (let index be s.t. s-Otherend(clist[index]) = ENV in
9         let clist' = ⟨clist[i] | 1 ≤ i ≤ len clist ∧ i ≠ index⟩ in
10        let map' = map + [closure ↦ clist'] in
11        let (,chan) = clist[index] in
12        (mapset' ∪ {map'}, chanlist' ⌢ ⟨mk-Id0(⟨⟩, chan)⟩))
13       else
14        (mapset', chanlist')))
```

type: *ExpimpchanD*-**set** *NameclosureD* Id_0^* → *ExpimpchanD*-**set** Id_0^*

目的　　抽取用于信道连接的信道标识符，该信道连接是由某个进口或出口变量隐含的。此信道标识符出现在各种子功能块的 *ExpimpchanD* 描述符中。在已经抽取一个信道标识符后，信息就从 *ExpimpchanD* 描述符中删除了。此函数还返回修改了的 *ExpimpchanD* 描述符。

参数

mapset　　是 *ExpimpchanD* 映射表的集合。每个子功能块都有一个映射表。
closure　　隐式子信道附属的名字闭包描述符 *NameclosureD*
chanlist　　递归创建的信道描述符表

结果　　修改过的 *ExpimpchanD* 映射表的集合和构造的子信道标识符表

算法

第 1 行　　当处理完所有的 *ExpimpchanD* 描述符时，则返回构造的信道标识符表。
第 4 行　　令 *map* 表示要考虑的下一个 *ExpimpchanD* 描述符。
第 5 行　　修改对应于其余子功能块的 *ExpimpchanD* 映射表的集合，并构造导向其余子功能块的适当的子信道。
第 6 行　　*clist* 包含有关与子功能块相连的所有子信道的信息，这些子信道传递对应于名字闭包描述符 *NameclosureD* (*closure*) 的隐式信号。
第 7 行　　如果一个子信道导向该子结构的环境，则
第 8 行　　令 *index* 是一元素的标引号，该元素包含导向环境的信道名。
第 9 行　　从表中排除那个元素。
第 10 行　　用此新表修改该映射表 *ExpimpchanD*。
第 11 行　　令 *chan* 表示子信道名。
第 12 行　　返回修改过的 *ExpimpchanD* 映射表的集合和信道标识符表。

第 14 行　　如果没有导向该子结构环境的子信道，则返回 *ExpimpchanD* 映射表和未改变的信道标识符表。

3.11 实用函数

get-predef-sort(*quot*)(*dict*) ≜ (3.11.1)

```
1  (let nm = mk-Name0(quot, nil),
2       level = dict(SCOPEUNIT) in
3   let sys = level[1] in
4   sys ⌢ ⟨(TYPE, nm)⟩)
```

type: *Char** → *Dict* → *Sortqual*

目的　　抽取一预定义类别的限定表 *Qual*

参数

quot　　表示此类别名的拼字的字符串序列

算法

第 1 行　　从拼字构造一 AS_0 名字。

第 2—3 行　　令 *sys* 表示系统层的限定表 *Qual*。

第 4 行　　返回预定义类别的限定表 *Qual*（定义在系统层）。

get-parent(*qual*)(*dict*) ≜ (3.11.2)

```
1  (if qual ∈ dom dict then
2     (if is-SortD(dict(qual)) then
3        qual
4        else
5        (let mk-SyntypeD(q,,,) = dict(qual) in
6         get-parent(q)(dict \ {qual})))
7     else
8     exit、"§5.4.1.9: 同义类型用自己定义自己")
```

type: *Sortqual* → *Dict* → *Sortqual*

目的　　从表示类别或同义类型的一个 *Qual*，抽取父辈类别（在同义类型情况下）

算法

第 1 行　　如果此限定表 *Qual* 在字典 *Dict* 中，则

第 2—3 行　　如果此限定表 *Qual* 表示一个类别，则返回此限定表 *Qual*，否则

第 5—6 行　　如果此限定表 *Qual* 表示同义类型，则抽取其父辈类别，但从字典 *Dict* 中排除此同义类型的限定表 *Qual*，这样使得我们可以检测到此同义类型被递归地定义的情况。

is-local(*qual*)(*dict*) ≜ (3.11.3)

```
1  qual = ENV ∨
2  dict(SCOPEUNIT) = get-sur(qual)
```

type: (*Qual* | ENV) → *Dict* → *Bool*

目的　　如果一个限定表 *Qual* 的定义对一个作用域单位是局部的，则返回真。

算法

第 1 行　　如果 *qual* 是引用文 **ENV**（用于有关信道和信号路由的场合）或者

第 2 行　　它们是在相同作用域单位中定义的，则返回真。

process-level(*qual*) ≙ **(3.11.4)**

```
1  (let (q, ) = qual[len qual] in
2   if q ∈ {PROCEDURE, SERVICE} then
3     process-level(get-sur(qual))
4    else
5     qual)
```

type:　*Qual* → *Qual*

目的　　从表示一个进程、过程、或服务的一个限定表 *Qual* 抽取外围进程的限定表 *Qual*

算法

第 1—5 行　　如果此限定表 *Qual* 表示一个过程或一个服务，则把此函数应用于外围作用域单位的限定表 *Qual* 上。

process-or-service-level(*qual*) ≙ **(3.11.5)**

```
1  (let (q, ) = qual[len qual] in
2   if q = PROCEDURE then
3     process-or-service-level(get-sur(qual))
4    else
5     qual)
```

type:　*Qual* → *Qual*

目的　　从表示一个进程、一个过程或服务的限定表 *Qual* 中抽取外围进程或服务的限定表 *Qual*

算法

第 1—5 行　　如果此限定表 *Qual* 表示一个过程，则把此函数应用于外围作用域单位的限定表 *Qual* 上。

get-sur(*qual*) ≙ **(3.11.6)**

```
1  ⟨qual[i] | 1 ≤ i ≤ len qual − 1⟩
```

type:　*Qual* → *Qual*

目的　　从一个限定表 *Qual*，通过删除最后一个元素，返回外围作用域单位的限定表 *Qual*。

get-inherited-parent(*qual*)(*dict*) ≙ (3.11.7)

```
1  (if qual ∈ dom dict then
2      cases dict(qual):
3       (mk-SortD(, pqual, , )
4          → if pqual = nil then qual else get-inherited-parent(pqual)(dict \ {qual}),
5        mk-SyntypeD(pqual, , , )
6          → get-inherited-parent(pqual)(dict \ {qual}))
7    else
8     exit("§5.4.1.9: 同义类型用自己定义自己")
```

type: *Sortqual* → *Dict* → *Sortqual*

目的 从一个表示类别或同义类型的限定表 *Qual*，抽取一个继承类别的父辈类别

算法

第 1 行 如果此限定表 *Qual* 是在字典 *Dict* 中，则

第 3 行 如果此限定表 *Qual* 表示一非继承类别，则返回此限定表 *Qual*，否则如果此 *Qual* 表示一继承类别，则抽取其父辈类别，但要从字典 *Dict* 中排除此类别的 *Qual*，使得我们可以检测到此类别继承自身的情况。

第 5 行 如果此限定表 *Qual* 表示一同义类型，则抽取其父辈类别，但从字典 *Dict* 中排除此同义类型的 *Qual*，使得我们可以检测到此类别继承自身的情况。

3.12 辅助信息的生成

本节包含一些函数，它们用于生成辅助信息提供给基础系统使用。入口函数是 *generate-auxiliary-information*。

generate-auxiliary-information(*extparms*)(*dict*) ≙ (3.12.1)

```
1  (let (starttime, timeunit) = derive-time-inf(extparms)(dict) in
2   let tick = make-tick-function(timeunit)(dict) in
3   let terminf = make-term-inf(dict) in
4   let expired = make-expired-function(dict) in
5   let delayf = derive-delay-inf(extparms)(dict) in
6   ((tick, starttime), terminf, expired, delayf))
```

type: *External-Information* → *Dict* → *Auxiliary-Information*

目的 导出辅助信息提供给基础系统使用（附件 F. 3）

参数

extparms 一些外部信息，从中抽取所需信息

结果 辅助信息 *Auxiliary-Information*

算法

第 1 行 导出表示开始时间的 TIME 字面值和表示最小时间单位的 DURATION 字面值。

第 2 行 构造用于更新当前时间的函数（即，它把时间单位 *timeunit* 加到当前时间上）。

第 3 行 构造 PID 类别的标识符，和三个字面值 NULL、TRUE 和 FALSE 的标识符。

第 4 行　　构造一个函数用于测试一个给定的定时器是否到时（即，它比较两个 TIME 值）。
第 5 行　　导出用于信道中不确定延迟的模型里的函数。
第 6 行　　构造并返回辅助信息 *Auxiliary-Information*。

make-tick-function(*timeunit*)(*dict*) ≙ (3.12.2)

```
1  (let durationq = get-predef-sort("DURATION")(dict),
2       timeq = get-predef-sort("TIME")(dict) in
3   let mk-OperatorD(, , nq, ) = dict(timeq ↷ ⟨(OPERATOR, (PLUS, ⟨timeq, durationq⟩, timeq))⟩) in
4   let as1 id = make-as1-identifier(nq)(dict) in
5   let f(value) = mk-Ground-term1((as1 id, ⟨value, timeunit⟩)) in
6   f)
```

type: *Literal-operator-identifier*$_1$ → *Dict* → (*Ground-term*$_1$ → *Ground-term*$_1$)

目的　　构造一个函数，它用一个给定的最小时间单位来更新时间。本函数用于抽象 SDL 自动机中。应用 NOW 的当前值并且返回这个新值。

参数

timeunit　　用一个 AS$_1$ 字面值表示的时间单位

算法

第 1 行　　构造 DURATION 类别的限定表 *Qual*。
第 2 行　　构造 TIME 类别的限定表 *Qual*。
第 3 行　　分解为 TIME 类别定义的“+”运算符的描述符。此运算符的限定表 *Qual* 是用包含实体类（**OPERATOR**）和此运算符的名字（**PLUS**）、变元的类别（*timeq* 和 *durationq*）以及结果的类别（*timeq*）等成分的元素与 TIME 的限定表 *Qual* 拼接而构成的。参照 *Operatorqualelem* 的定义。
第 4 行　　构造此运算符的 AS$_1$ 标识符。
第 5—6 行　　构造并返回一个 Meta-IV 函数。每当给定一 SDL 值时，把一个时间单位加到此值上。

make-term-inf(*dict*) ≙ (3.12.3)

```
1  (let bqual = get-predef-sort("BOOLEAN")(dict) in
2   let pidqual = get-predef-sort("PID")(dict) in
3   let pid1 = make-as1-identifier(pidqual)(dict),
4       null1 = make-as1-identifier(pidqual ↷ ⟨(LITERAL, mk-Name0("NULL", nil))⟩)(dict),
5       true1 = make-as1-identifier(bqual ↷ ⟨(LITERAL, mk-Name0("TRUE", nil))⟩)(dict),
6       false1 = make-as1-identifier(bqual ↷ ⟨(LITERAL, mk-Name0("FALSE", nil))⟩)(dict) in
7   (pid1, null1, true1, false1))
```

type: *Dict* → *Term-Information*

目的　　构造用于动态语义中的项信息 *Term-Information*（参见 *Term-Information* 的域定义）

结果　　由 PID 类别、NULL 字面值、TRUE 字面值和 FALSE 字面值的标识符组成的项信息 *Term-Information*

make-expired-function(*dict*) ≜ (3.12.4)

```
1  (let bqual = get-predef-sort("BOOLEAN")(dict) in
2   let geop = mk-Qualop0(bqual, mk-Quotedop0(GE)) in
3   let expiredf(currenttime, expiredtime) =
4       eval-simple-expr(mk-Operatorapp0(geop, ⟨currenttime, expiredtime⟩), "BOOLEAN")(dict) in
5   expiredf)
```

type: *Dict* → *Is-expiredF*

目的 构造一个函数，用于测试一个给定时间值是否超过一个到期时间值，即产生一个函数，它返回 Meta-IV 值 true，如果

currenttime >= **expiredtime**

此处 **currenttime** 和 **expiredtime** 是形式参数。

此函数用于动态语义学中。

算法

第 1 行 构造 BOOLEAN 类别的限定表 *Qual*。

第 2 行 构造“>=”运算符的 AS_0 标识符。

第 3—5 行 构造并返回 *Is-Expired* 函数 (*expiredf*)。

3.13 全局常数的映射表

下面定义两个全局映射表 *name-to-name*$_1$ 和 *exportmap*。

3.13.1 AS_0 名字和 AS_1 名字之间的关系

因为名字的拼字等对于解释是没有什么关系的，所以 AS_1 中的名字和 AS_0 中的表示法不同。因此，AS_0 名字 (*Name*$_0$)，字符串 (*String*$_0$) 和引用文运算符 (*Quotedop*$_0$) 是借助于一全局映射表 (*name-to-name*$_1$) 变换到 AS_1 名字 *Name*$_1$ 的。

```
let name-to-name1 be s.t. (∀nm1 ∈ rng name-to-name1)(is-Name1(nm1)) ∧
   (∀(nm1, nm2) ∈ (Name0 | String0 | Quotedop0))
    (({nm1, nm2} ⊂ dom name-to-name1) ∧
     (nm1 ≠ nm2 ⊃ name-to-name1(nm1) ≠ name-to-name1(nm2))) in
```

目的 构造一个映射表，它把 AS_0 的名字、字符串和引用文运算符映射到 AS_1 的名字。

算法 构造映射表，使得每个可能的 AS_0 名字、字符串和引用文运算符都在映射表的域中，而每两个实体映射为不同的 AS_1 名字。

3.13.2 进口（出口）名字和隐式信号名之间的关系

在 SDL 中，某些隐式信号被附加到每个出口变量上。如果两个出口变量具有相同的名字（拼字）和具有相同的数据类别，则它们就附有相同的隐式信号。为了管理信号的这种“共享”，使用了一个专门的常数映射表 *exportmap*，此映射表把由出口变量的类别和出口变量名组成的偶对映射到附加信号名的一个描述符：

let *exportmap* = [*closure* ↦ **mk**-*SignalnamesD*(*create-unique-name*(),
create-unique-name(),
create-unique-name()) | *closure* ∈ *NameclosureD*] **in**

出口映射表 *exportmap* 的域定义如下：

1	*Exportmap*	=	*NameclosureD* $\overset{m}{\rightarrow}$ *SignalnamesD*
2	*NameclosureD*	=	*Sortqual* $Name_0$
3	*SignalnamesD*	::	*Impliednm Xquerynm Xreplynm*
4	*Impliednm*	=	$Name_0$
5	*Xquerynm*	=	$Name_0$
6	*Xreplynm*	=	$Name_0$

信号描述符（*SignalnamesD*）的组成如下：

Impliednm	附加在每个出口变量上的隐式变量
Xquerynm	由一进口者发送的 xtQUERY 信号
Xreplynm	由出口者发送的 xtREPLY 信号，并带有类别 *Sortqual* 的一个值

3.14 非形式函数

apply-generic-parameters(*genericsystem*, *extparameters*) ≙ **(3.14.1)**

1 /* 把一个类属系统定义变换为一具体的系统定义 */

type: Sys_0 *External-Information* → Sys_0

SDL 没有定义怎样应用类属参数。这个非形式函数需要一个 AS_0 系统定义和包括某些类属实在参数的外部信息，并返回一个不包含类属参数的 AS_0 系统定义，即，一个系统定义，它不含外部同义词，并在任选动作的回答中没有非形式正文。

derive-time-inf(*extparms*)(*dict*) ≙ **(3.14.2)**

1 /* 从外部信息导出开始时间和时间单位 */

type: *External-Information* → *Dict* → $Literal\text{-}operator\text{-}identifier_1$ $Literal\text{-}operator\text{-}identifier_1$

这个非形式函数返回表示系统起始时间的类别 TIME 的字面值运算符标识符 $Literal\text{-}operator\text{-}identifier_1$ 和表示绝对时间增量值的类别 DURATION 的字面值运算符标识符 $Literal\text{-}operator\text{-}identifier_1$（两者都用于 NOW 表达式的模型中，参见附件 F. 3）。

derive-delay-inf(*extparms*)(*dict*) ≙ **(3.14.3)**

1 /* 从外部信息导出延迟函数 */

type: *External-Information* → *Dict* → *DelayF*

这个非形式函数根据外部信息导出。命令函数（*DelayF*）用于附件 F. 3，以模拟信道中的非确定性延迟。

eval-expr(*expr*, *sort*)(*dict*) ≙ **(3.14.4)**

1 /* 计算表达式的值 */

type: $Ground\text{-}expression_1$ $Char^+$ → *Dict* → (*Intg* | *Bool*)

目的 计算一个 AS_1 简单表达式

参数

expr AS_1 简单表达式

sort 表达式类别的拼字

结果 如果 SDL 类别为“Integer”则结果为 Meta-IV 类型 *Intg*；如果 SDL 类别是“Boolean”则结果为 Meta-IV 域 *Bool*。

算法 为节省篇幅起见，没有把此函数包括在形式定义中。请参考 Z.100 的 § 5.6 中关于预定义类别的形式规格说明，并参考附件 F.3 关于 SDL 数据模型的形式定义。

select-consistent-subset(*systemdefinition*, *genericparameters*) ≜ **(3.14.5)**

```
1  /*  借助于AS1和子集实在参数，选择一个一致性子集  */
```

type: *System-definition*$_1$ *External-Information* → *Block-identifier*$_1$-**set**

该非形式函数返回表示一致性子集的功能标识符的集合。它在附件 F.3 中用于选择将要解释的功能块。在附件 F.3 中也要检查一致性子集是否与在 Z.100 的 § 3.2.1 中定义的性质相符合。

create-unique-name() ≜ **(3.14.6)**

```
1  /*  创建一个新的 mk-Name0(Token,nil)  */
```

type: () → *Name*$_0$

该非形式函数返回一个先前未用过的名字 AS_0 *Name*$_0$。请注意，此函数自然是强制性的。我们假设此函数使用了一个全局变量，用来记录跟踪所有用过的名字。

check-name-syntax(*string*) ≜ **(3.14.7)**

```
1  /*  把一个字符串转换为大写字母串  */
```

type: *Char** → *Char**

首先，删除后随空格的任何下划线字符的序列；其次，用一个下划线字符替换每一个空格序列；再其次，把拼字转换为大写字母；最后，检查最终的拼字是否和在 Z.100 的 § 2.2.1 中定义的名字的词法规则一致。

4 与 Z.100 不一致的地方

从 SG X (Marts 1988) 最终通过 Z.100 建议文本以后，在建议的正文中又发现了几处不一致或遗漏。为了形式定义的完整性，对这些情况作了如下决定：

1. Z.100 的 § 2.7.4 中的语句：

   “然而，如果这两个信号是由连接起始进程 *Qriginating-process* 和目的进程 *Destination-process* 的同一路径传递的，则保留它们的次序。”

   应解释为：

   “然而，如果这两个信号是由连接起始进程 *Qriginating-process* 和目的进程 *Destination-process* 的同一信

道传递的，或如果它们是在同一功能块中定义的，则保留它们的次序。”

2. Z.100 的 § 2.7.5 中的语句：

“各个判定回答 *Decision-answer* 必须互相排斥。”

应解释为

“各个判定回答 *Decision-answer* 必须互相排斥，并且它们必须是相同类别的。如果判定问题 *Decision-question* 包含一个表达式 *Expression*，则它必须和判定回答 *Decision-answer* 的类别相同。”

3. 在 Z.100 的 § 2.7 中为了避免循环定义而在模型 *Model* 的前面增加一条新语句：

“可以确定判定的上下文（即类别）而不需要考虑由＜字符串＞构成的＜回答＞。

4. Z.100 的 § 3.3 中的语句：

“如果一致性划分的子集包含一用子信号通信的功能块定义，则与此功能块通信的那些功能块必须使用相同的子信号。”

应解释为：

“当选择一致性子集时，连接于信道一个端点的信号路由上的信号集合一定不包括所包含的子信号的父辈信号，并且除非另一端点是系统的环境 ENVIRONMENT，则第一个端点的信号集合必须等于连接到另一端点的信号路由上的信号集合。”

5. Z.100 的 § 4.3.3 中的语句：

“＜布尔简单表达式＞必须不依赖于＜选择定义＞中的任何定义。”

修改如下（为了保证唯一的选择和唯一的同义词约束）：

直接或间接（经过其它同义词）用在＜选择定义＞的＜布尔简单表达式＞中的同义词必须满足以下条件：

- 它必须符合 § 2.2.2 中所陈述的可见性规则，即使此同义词定义没有被选择也必须这样。
- 在＜同义词定义＞中的类别必须限定在系统层上，条件是对那个（预定义）类别存在一个可见的再定义，甚至于如果该再定义是局部的而且也没有被选择也是如此。
- 必须存在一种选择次序，使得所有的＜简单表达式＞都可以求值。

6. Z.100 的 § 4.10.2 中的语句：

“信号是高优先级信号，当且仅当在一个＜服务定义＞的＜优先级输入＞中对它进行了描述。”

解释为

“在一进程中，信号是高优先级信号，当且仅当在该进程中一个＜服务定义＞的＜优先级输入＞中对它进行了描述。”

7. Z.100 的 § 4.10.2 中的语句：

“当包围它的＜进程定义＞包含一个＜服务定义＞时，一个＜过程定义＞一定不能带有＜状态＞。”

被扩展为

“当包围它的＜进程定义＞包含一个＜服务定义＞时，一个＜过程定义＞一定不能带有＜状态＞。对一个以上服务是可见的＜过程定义＞一定不能包含一个 VIA 构件。”

8. Z.100 的 § 5.6.7.1 中的等式：

```
r > 1 == True => a * r > a == True;
r > 1 == True => a / r < a == True;
```

应该是

```
r > 1 == True => a * r > a == True;
r > 1 == True => a / r < a == True;
```

以便包括 a 等于 0 的情况。

域索引

函数索引

引用文索引

错误信息

§ 2.5.2　第 2 条路径的方向必须与第一条路径的方向相反 122
§ 2.5.2　有效输入信号必须包含所有的局部信号并且不包含定时器 63
§ 2.5.2　当没有指定信号路由时，有效输入信号集必须加以指定 249
§ 2.5.3　信道到信号路由的连接出现两次 125
§ 2.5.3　信道到信号路由的连接不是良形式的 126
§ 2.5.3　没有用于把信道连接到功能块的连接 125
§ 2.5.3　信号路由应出现在一个且仅出现在一个连接中 125
§ 2.5.3　在多于一个连接中提及同一个信号路由 125
§ 2.5.5　信号表中的信号标识符不能区分 124
§ 2.5　信道的端点被定义在本信道以外的作用域单位中 57
§ 2.5　信道的两个端点必须是不同的 57
§ 2.5　信道定义中的第二条路径必须和第一条路径方向相反 57
§ 2.6.1.2　没有和那个类别唯一对应的被透露变量 120
§ 2.6.3　状态的结尾名必须和起始名相同 172
§ 2.6.3　结尾状态名不允许出现在星号状态中 172
§ 2.6.3　状态中的信号不是分离的 177，178
§ 2.6.3　状态节点中的信号不是良形式的 175
§ 2.6.3　状态中的状态名必须是不同的 173
§ 2.6.4　输入节点中的参数的个数出错 192
§ 2.6.6　标号相同 170
§ 2.6.7.1　在跃迁中丢失终端符 166，168
§ 2.6.7.2.1　下一个状态中的名字必须表示一个已定义的状态 195，196，197，238
§ 2.6.7.2.2　在汇合中的标号未被定义 195
§ 2.7.2　实在参数表的长度必须等于形式参数表的长度 212
§ 2.7.2　创建的进程必须被定义在同一个功能块中 212
§ 2.7.3　IN/OUT 实在参数必须是一个变量标识符 216
§ 2.7.3　过程参数表的长度必须等于形式参数表的长度 214
§ 2.7.3　对于 IN/OUT 形式参数和实在参数必须规定相同的类别 216
§ 2.7.4　在 VIA（经由）集合中的信道未连接到功能块上 211
§ 2.7.4　在输出动作中的标识符必须表示一个信号 208
§ 2.7.4　如果规定了信号路由，则不能在 VIA 中描述信道 211
§ 2.7.4　非法 VIA 集合 210
§ 2.7.4　对输出动作变元个数不正确 208
§ 2.7.4　信号不能经由 VIA 集合中的信道传递 211
§ 2.7.4　输出中的信号没有可以接收的接收者 210
§ 2.7.4　在 VIA 集合中信号路由或信道被指定两次 210
§ 2.8　在复位动作中的标识符不是一个定时器 203
§ 2.8　在设置动作中的标识符不是一个定时器 205
§ 2.8　在定时器活跃表达式中的标识符不表示一个定时器 152
§ 2.8　非法的定时器活跃表达式 152
§ 2.8　在复位动作中参数表的长度必须等于定义中规定的长度 203
§ 2.8　在设置动作中参数表的长度必须等于定义中规定的长度 205
§ 3.2.2　功能块子结构不包含一个功能块定义 54
§ 3.2.2　信道连接出现两次 128
§ 3.2.2　信道连接不是良形式的 129
§ 3.2.2　没有连接用于连接到功能块子结构的信道 128

§ 3.2.2　子信道应出现在一个且仅出现在一个连接中 128
§ 3.2.2　子信道在多于一个连接中被描述 128
§ 3.2.3　在连接中的功能块标识符不表示信道的端点 60
§ 3.2.3　必须正好是两个信道端点的连接
§ 3.3　进程使用相同信号在不同具体化层次上的信号 61
§ 4.10.1　服务信号路由标识符在服务信号路由连接中出现了两次 133
§ 4.10.1　服务信号路由中的信号必须和用于所连接的信号路由中的信号相同 133
§ 4.10.1　在双向信号路由中两个方向的端点必须相匹配 241
§ 4.10.1　服务的信号路由的两个端点必须不同 241
§ 4.10.2　所有服务信号路由必须在一个连接中被描述 132
§ 4.10.2　非法的服务信号路由连接 132
§ 4.10.2　在进程中丢失了信号路由连接 132
§ 4.10.2　不止一个服务包含一个初始跃迁串 238
§ 4.10.2　服务信号路由没连接到 VIA 集合中的信号路由 210
§ 4.10.2　在分解或服务中的过程具有状态或进口表达式 67
§ 4.10.2　进程包含服务定义和定时器定义两类定义 116
§ 4.10.2　服务信号不是良形式的 161
§ 4.10　一个分解必须至少包含一个服务定义 235
§ 4.10　非法使用高优先级信号 208
§ 4.11　如果省略优先级，则只能指定一个连续信号 182
§ 4.11　几个连续信号具有相同的优先级 185
§ 4.2.3　等式中的两项必须是相同的类别 103
§ 4.3.2　简单表达式不是预定义类别的或其中包含未定义的标识符 44
§ 4.3.3　在选择中的定义在那个作用域单位中是不允许的 39
§ 4.3.3　选择的定义在那个作用域单位中是不允许的 48
§ 4.3.4　不止一个任选的回答与该问题匹配 221
§ 4.3.4　没有一个任选的回答与该任选的问题匹配 221
§ 4.4　非法的星号状态 173
§ 4.7　状态具有多于一个星号输入或保存 177，178
§ 4.9　在初始跃迁中的下一状态划线 195
§ 5.2.1　在部分类型定义中的结尾名不同于定义名 71
§ 5.2.1　数据类型定义中的类别没有值 226
§ 5.2.1　不存在一个运算符，它返回那个类别的一个值 71
§ 5.2.2　在一个部分类型定义中字面值定义两次 94。
§ 5.2.2　用显式定义和继承性定义两种方式来定义运算符或字面值 73
§ 5.2.3　不一致地使用隐含量化的值名字 141
§ 5.2.3　值标识符一定不能限定 141
§ 5.2　对一个运算符来说参数表的长度不正确 147
§ 5.4.1.10　两个结构字段具有相同的名字 113
§ 5.4.1.11　在重新命名中两次定义的字面值 76
§ 5.4.1.11　在字面值重新命名中的字面值没有在父辈类别中定义 76
§ 5.4.1.11　字面值被重新命名两次 76
§ 5.4.1.11　在运算符重新命名中的运算符没有在父辈类别中定义 78
§ 5.4.1.11　运算符重新命名两次 78
§ 5.4.1.11　带有感叹号的运算符在一个运算符重新命名中被描述 78
§ 5.4.1.11　类别基于自身 73

§ 5.4.1.12.2 CONSTANT 生成程序实在参数必须是一个项 89
§ 5.4.1.12.2 LITERAL 生成程序实在参数必须是一个字面值标记 89
§ 5.4.1.12.2 在生成程序中实在参数表和形式参数表的长度必须相同 84
§ 5.4.1.12.2 OPERATOR 生成程序的实在参数必须是一个运算符标记 89
§ 5.4.1.12.2 TYPE 生成程序的实在参数必须是一个标识符 89
§ 5.4.1.12 生成程序定义中的结尾名不同于定义名 70
§ 5.4.1.12 用于字面值标记的生成程序的常数参数 87
§ 5.4.1.12 限定了生成程序的形式名字 87
§ 5.4.1.12 名字类不能用于等式中 87
§ 5.4.1.13 同义词类别不能被唯一地确定 93
§ 5.4.1.14 在正规间隔中的字符串的长度不等于 1 98
§ 5.4.1.14 在名字类中的重复名字必须表示一个自然数 98，99，100
§ 5.4.1.15 非法使用 SPELLING（拼字）运算符 148
§ 5.4.1.7 错误项是限制条件的一部分 105
§ 5.4.1.7 错误项被用于一个组合项中 135
§ 5.4.1.9.1 对于范围条件的类别没有定义有序运算符 92
§ 5.4.1.9 在同义类型定义中的结尾名不同于定义名 90
§ 5.4.1.9 同义类型基于自身 90
§ 5.4.1.9 同义类型用自己定义自己 256，258
§ 5.5.3.3 不止一个缺省赋值语句 84
§ 5.5.3 修改的赋值句没有扩展或有多重扩展 201
§ 5.5.4.1 NOW 表达式必须被用于 TIME 值被允许的地方 151
§ 5.5.4.3 PID 表达式用于错误的上下文中 153
§ 5.5.4 命令运算符不能被用于常数表达式中 149

中国印刷 ISBN 92-61-03785-2