DDL-1

DDL (v.1) – Data Definition Language. It is designed to represent linked data of generic type. DDL has a C-style syntax.

Language brief.

Here is a «language brief».

basic elements

type	`T`	
constant	`C`	
literal	`L`	
expression	`E`	expression is evaluated in the context of the resulting type

basic types

integral

`sint8`	`uint8`
`sint16`	`uint16`
`sint32`	`uint32`
`sint64`	`uint64`
`sint`	`uint`
	`ulen`

```
( sint, uint ) = ( sint32, uint32 ) or ( sint64, uint64 )
```

```
int = sint
```

ulen = uint32 or uint64 ≥ uint

special

text	strings
ip	IP addresses

derives types

T *	pointer
T[E]	array (ulen E)
T[]	array with an inferred dimension

struct T

{

T_1 name_1[=E_1];

T_2 name_2[=E_2];

…

T_n name_n[=E_n];

};

structure

type aliases

```
type T = T' ;
```

constants

```
T C = E ;
```

expressions

compound

$\{ E_1 , E_2 , \ldots , E_n \}$

$[E] \{ .name_1 = E_1 , .name_2 = E_2 , \ldots , .name_n = E_n \}$

simple

L

C

operator's

(E)

T(E)   T is an integral type, operations are performed in the ring T

$E_1\ op_2\ E_2$ op_2 = + − * / %

$op_1\ E$ op_1 = + − * &

E.name

E−>name

E[E']   slen E' – special internal type, ±ulen with overflow check

literals

1234567890	decimal
1234567890ABCDEFabcdefH	hexadecimal (H|h)
1000001B	binary (B|b)
192.168.1.10	IP address
null	universal null
"abcdef\b\t\n\v\f\r"	strings with \-symbols
'abcdef'	simple strings

scopes

```
scope name { <content> }
```

$\mathtt{name}_1\#\mathtt{name}_2\#...\#\mathtt{name}_n$

$\#\mathtt{name}_1\#\mathtt{name}_2\#...\#\mathtt{name}_n$

$.\#\mathtt{name}_1\#\mathtt{name}_2\#...\#\mathtt{name}_n$

$..\#\mathtt{name}_1\#\mathtt{name}_2\#...\#\mathtt{name}_n$

$...\#\mathtt{name}_1\#\mathtt{name}_2\#...\#\mathtt{name}_n$

file inclusion

```
include <file-name>
```

file content – scope-level, unresolved names are permitted

Tokens.

The source file is parsed on tokens. On each step the prefix is cut up from the the rest of the file. Prefixes are cut up according ether the maximal prefix rule or the minimal prefix rule. The file is divided on text lines to designate a character position. An end of line is determined by the one of the following character combinations (the longest is selected) : “\r” “\n” “\r\n” .

symbol classes

L	*a..z A..Z _*
D	*0..9*
B	*0 1*
H	*0..9 a..f A..F*
C	*[] {} () ; , # = & + − ∗ / % .*
S	*space \t \v \f \r \n*
P	*printable symbols*
$\mathbf{P}_{>}$	**P** \ { > }
$\mathbf{P}_{'}$	**P** \ { ' }
$\mathbf{P}_{"}$	**P** \ { " , \ }

token classes

Comments are cut up according the minimal prefix rule.

ShortComment	*/ / … end-of-line or end-of-file*
LongComment	*/ ∗ … ∗ /*

All othe tokens are cut up according the maximal prefix rule.

Space	$\mathbf{S}^{1..}$
PunctSym	**C** \ { . }
PunctArrow	− >
PunctDots	$.^{1..}$

Word	$\mathbf{L}(\mathbf{L}\vert\mathbf{D})^{*}$
Dec	$\mathbf{D}^{1..}$
Bin	$\mathbf{B}^{1..}(B\vert b)$
Hex	$\mathbf{H}^{1..}(H\vert h)$
Number	Dec Bin Hex
BString	$<\ \mathbf{P}_{>}^{*}\ >$
SString	$'\ \mathbf{P}_{'}^{*}\ '$
DString	$''\ (\ \mathbf{P}_{''}\ \vert\ \backslash\ \mathbf{P}\ )^{*}\ ''$

Two consecutive tokens Number and Word are diagnosed as a error. Tokens from the set Number ∩ Word are also diagnosed as a error .

first letter → token class

S	Space
C	Punct PunctArrow PunctDots
L	Word
D	Number
/	/ ShortComment LongComment
'	SString
"	DString
<	BString

Atoms.

Atoms	Tokens	Token values
Number	Dec Bin Hex	
String	SString DString	
FileName	BString	
Name	Word	
int	Word	“int”
sint	Word	“sint”
uint	Word	“uint”
ulen	Word	“ulen”
sint8	Word	“sint8”
sint16	Word	“sint16”
sint32	Word	“sint32”
sint64	Word	“sint64”
uint8	Word	“uint8”
uint16	Word	“uint16”
uint32	Word	“uint32”
uint64	Word	“uint64”
text	Word	“text”
ip	Word	“ip”
struct	Word	“struct”
type	Word	“type”
null	Word	“null”
scope	Word	“scope”
include	Word	“include”
const	Word	“const”

→	PunctArrow	“->”
.	PunctDots	“.”
...	PunctDots	“..” “...” ets.
*	PunctSym	“*”
,	PunctSym	“,”
;	PunctSym	“;”
=	PunctSym	“=”
+	PunctSym	“+”
-	PunctSym	“-”
&	PunctSym	“&”
#	PunctSym	“#”
/	PunctSym	“/”
%	PunctSym	“%”
(	PunctSym	“(”
)	PunctSym	“)”
[	PunctSym	“[”
]	PunctSym	“]”
{	PunctSym	“{”
}	PunctSym	“}”

Formal definition of the language.

BODY	empty BODY SCOPE BODY INCLUDE BODY TYPE BODY CONST BODY STRUCT ;
SCOPE	**scope Name {** BODY **}**
INCLUDE	**include FileName**
TYPE	**type Name =** TYPEDEF **;**
CONST	TYPEDEF **Name =** EXPR **;**
RNAME	**Name** RNAME **#** **Name**
NAME	RNAME **#** RNAME . **#** RNAME ... **#** RNAME
INAME	**int** **sint** **uint** **ulen** **sint8** **uint8** **sint16** **uint16** **sint32** **uint32** **sint64** **uint64**
TNAME	INAME **text** **ip**

TYPEDEF	NAME TNAME TYPEDEF * TYPEDEF [ ] TYPEDEF [EXPR] STRUCT
STRUCT	**struct Name** { SBODY }
SBODY	empty SBODY TYPE SBODY **const** CONST SBODY STRUCT ; SBODY TYPEDEF **Name** ; SBODY TYPEDEF **Name** = EXPR ;

EXPR	{ }
	{ ELIST }
	{ NELIST }
	EXPR { }
	EXPR { NELIST }
	EXPR_ADD

EXPR_ADD	EXPR_MUL EXPR_ADD + EXPR_MUL EXPR_ADD - EXPR_MUL
EXPR_MUL	EXPR_UN EXPR_MUL * EXPR_UN EXPR_MUL / EXPR_UN EXPR_MUL % EXPR_UN
EXPR_UN	EXPR_POST * EXPR_UN & EXPR_UN + EXPR_UN - EXPR_UN
EXPR_POST	EXPR_NNPOST **Number**
EXPR_NNPOST	EXPR_NNPRIM EXPR_POST [EXPR] EXPR_NNPOST . **Name** EXPR_NNPOST → **Name**
EXPR_NNPRIM	(EXPR)

	ITYPE (EXPR) NAME NNLIT
ELIST	EXPR ELIST , EXPR
NEXPR	**. Name =** EXPR
NELIST	NEXPR NELIST , NEXPR
ITYPE	NAME INAME
NNLIT	**null** **String** **Number . Number . Number . Number**

Data model.

<u>Body</u>

list<Alias>

list<Const>

list<Struct>

list<Len>

list<Scope> // level 1

<u>Alias</u>

Name

●→ Scope

depth

●→ Type

<u>Const</u>

Name

●→ Scope

depth

●→ Type

●→ Expr

<u>Field</u>

Name

●→ Type

●→ Expr

```
Struct
Name
●→ Scope
depth
Scope
olist<Field>

Len
●→ Expr

Scope
Name
●→ Scope
●→ Body

NameRef
    rel
    abs
    this
    dots
olist<Name>

Len
●→ Expr

Type
●→ Struct

    Type_suint<SUInt>
    Type_text
    Type_ip
    Type_ptr
```

●→ Type

Type_array

●→ Type

Type_array_len

●→ Type

Len

Type_struct

Type_ref (name link)

●→ NameRef

●→ Alias^{opt}

●→ Struct^{opt}

DomainType (name link^{opt})

●→ Type^{opt}

●→ NameRef^{opt}

●→ Alias^{opt}

Expr

PosName

Expr_add (_sub _mul _div _rem _ind)

●→ Expr

●→ Expr

Expr_deref (_address _plus _minus)

●→ Expr

Expr_number

Expr_ptr_select

●→ Expr

Name

Expr_select

●→ Expr

Name

Expr_domain

●→ Expr

●→ DomainType

Expr_var (name_link)

●→ NameRef

●→ Const

Expr_null

Expr_string

Expr_ip

Number

Number

Number

Number

Expr_noname_list

olist<●→ Expr>

Expr_named_list

list<●→ Name,●→ Expr>

Expr_apply_named_list

●→ Expr

list<●→ Name,●→ Expr>

File names.

extname – non-empty file name, without '/' '\' ':' characters, may be special (“.” “..”).

name – regular file name.

General file name:

(dev:)^{opt}(/)^{opt}(extname/)*name .

Normalized file name:

(dev:)^{opt}(/)^{opt}(name/)*name

(dev:)^{opt}(../)$^{1..}$(name/)*name .

Absolute names:

(dev:)^{opt}/(name/)*name

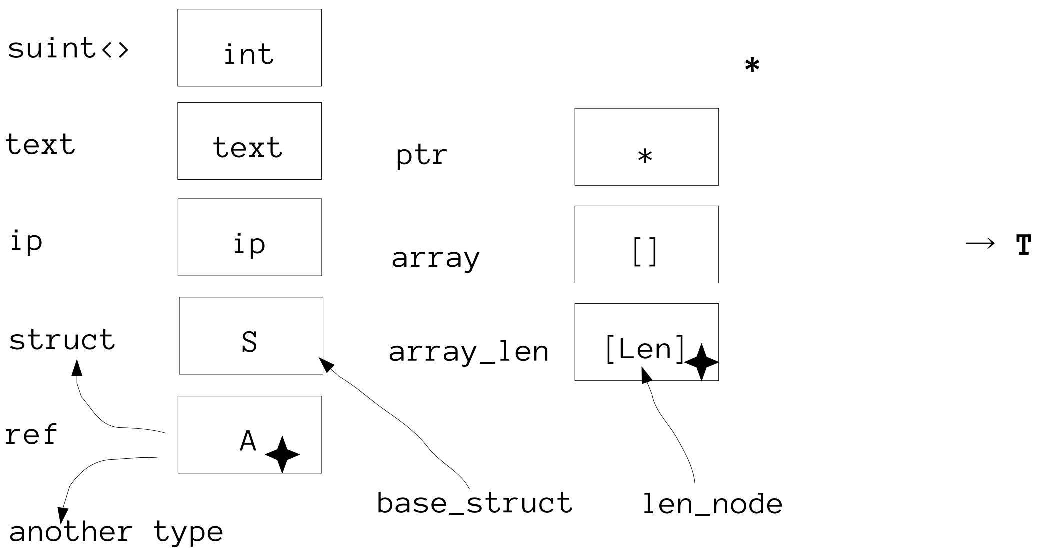
dev:(name/)*name .

Relative names:

(name/)*name .

Evaluation model.

types



forbidden type definition loops

T[] → T

T[Len] → T

T * → T

struct { T_1, ..., T_n } → T_1, ..., T_n

T[] → T

T[Len] → T

type genres and special types

int {**slen**, sint8, ..., uint64}

text

ip

struct {}

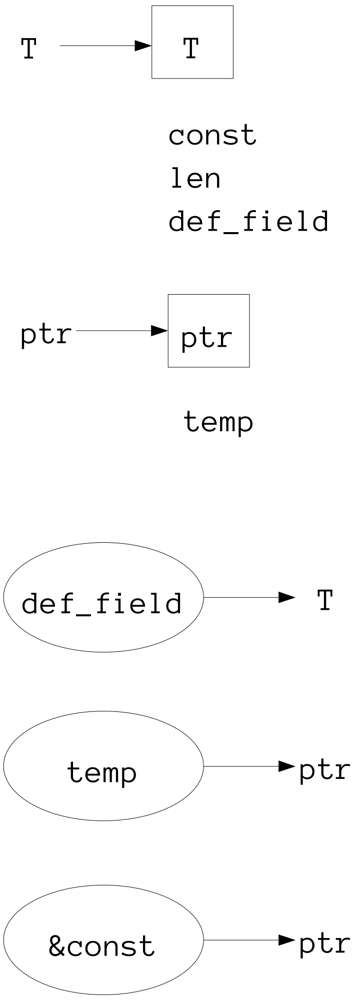
array {}

ptr {}

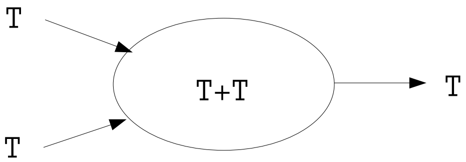
ptr*

LVptr

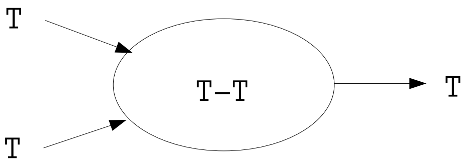
evaluation graph



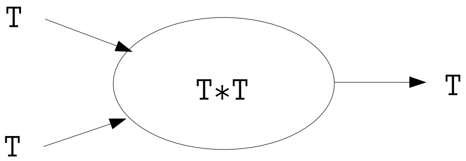
T : int



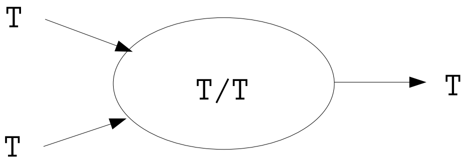
a+b



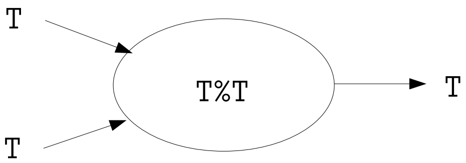
a−b



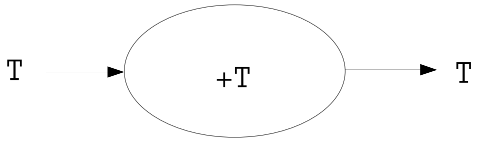
a∗b



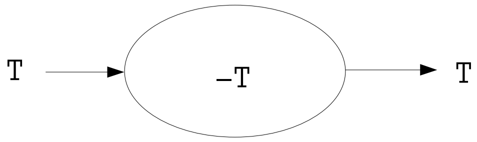
a/b



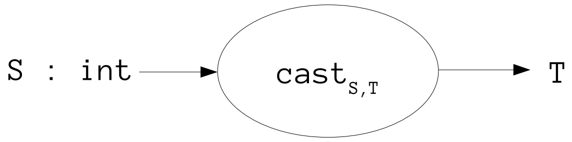
a%b

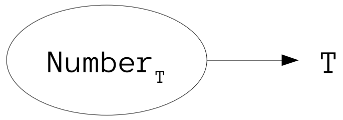


+a

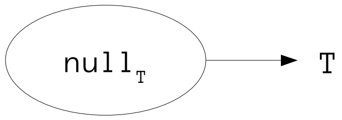


−a



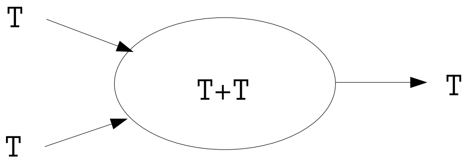


Number

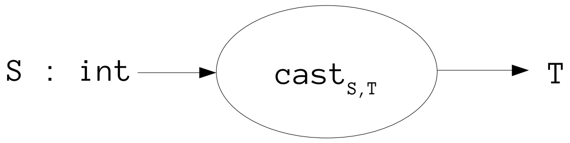


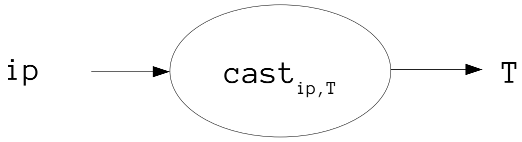
null, {}

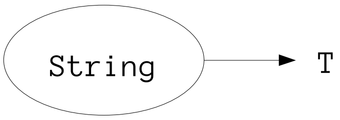
T = text



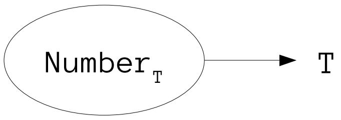
a+b



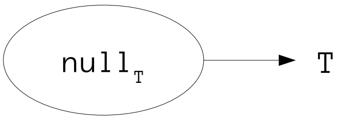




String

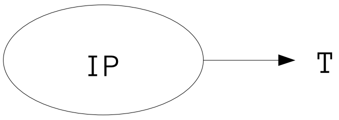


Number

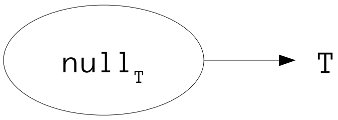


null, {}

T = ip

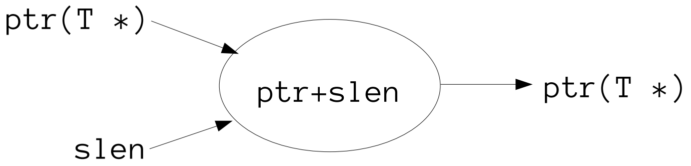


IP

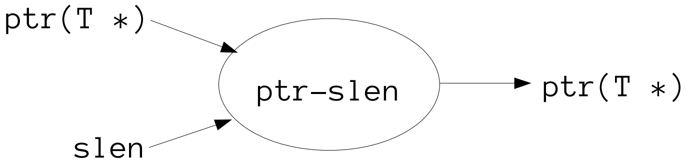


null, {}

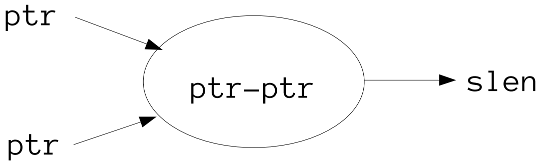
ptr



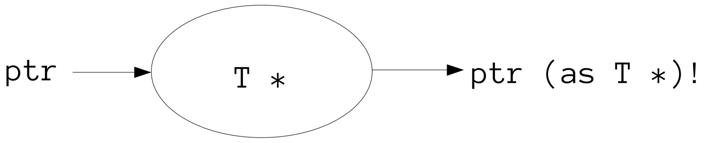
a+b (as ptr,T *)

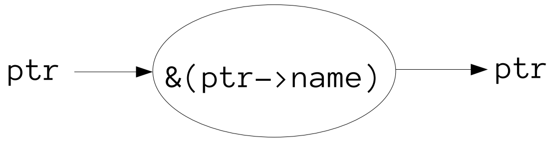


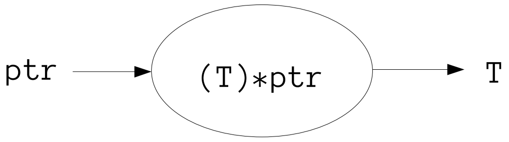
a-b (as ptr,T *)

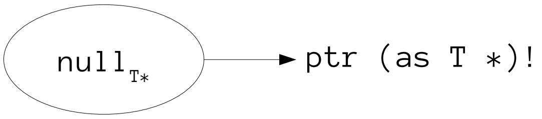


a-b (as int)



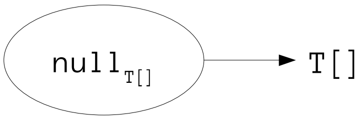




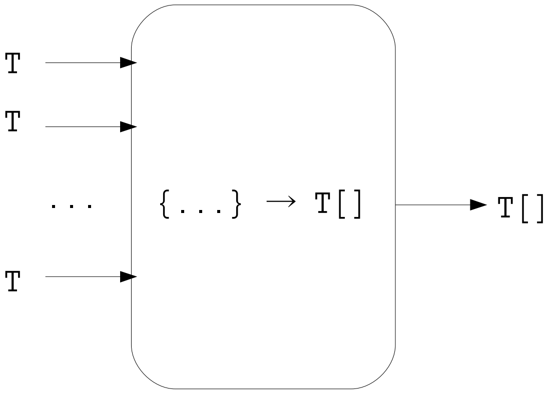


null, {}

T[]

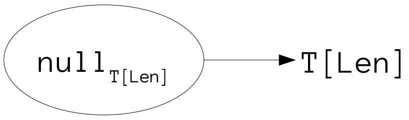


null

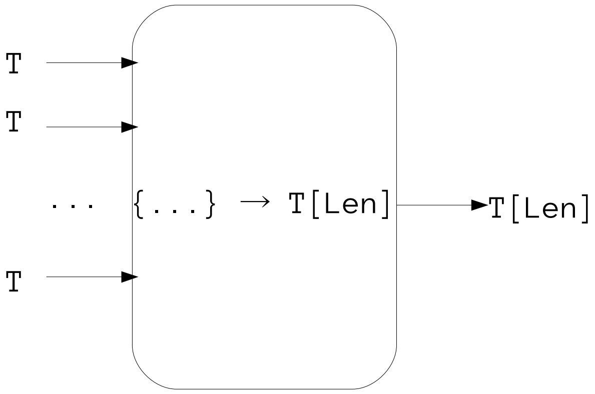


$\{a_1, \ldots, a_n\}$

T[Len]

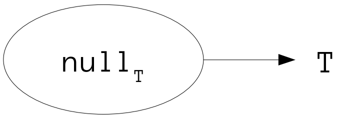


null

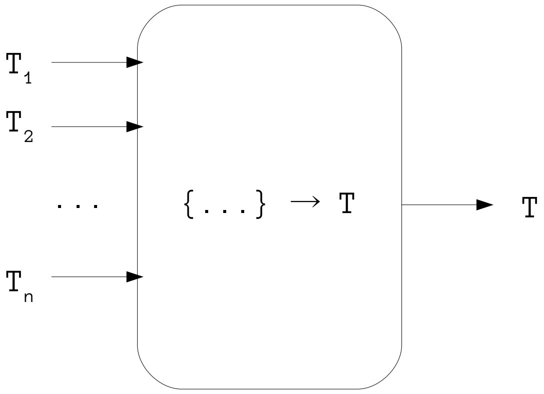


$\{a_1, \ldots, a_n\}$

T = struct { T_1, ..., T_n }



null



$\{a_1, \ldots, a_k\}$

simple expressions

```
null
Number
String
IP
const          LV
```

compound expressions

```
a + b
a - b
a * b
a / b
a % b
a [ b ]        // *(a+b)

+ a
- a
* a            LV
& a

domain( a )

a . name       // &a->name
a → name       LV
```

{ a_1, ..., a_n }

{ .name_1=a_1, ..., .name_n=a_n }

b { .name_1=a_1, ..., .name_n=a_n }

expression evaluation

(T, IP)

T = ip	ip ← IP
T = text	text ← ip ← IP

(T, String)

T = text	text ← String

(T, Number)

T : int	T ← Number
T = text	text ← Number // copy source string

(T, a + b)

T : int	(T,a) $+_{T}$ (T,b)
T = text	(T,a) $+_{text}$ (T,b)
T : ptr	(T,a) $+_{ptr*}$ (slen,b) or (T,b) $+_{ptr*}$ (slen,a)
T = ptr*	(ptr*,a) $+_{ptr*}$ (slen,b) or (ptr*,b) $+_{ptr*}$ (slen,a)

(T, a - b)

T : int	(T,a) $-_{T}$ (T,b) or T ← (ptr*,a) $-_{ptr*}$ (ptr*,b)
T : ptr	(T,a) $-_{ptr*}$ (slen,b)
T = ptr*	(ptr*,a) $-_{ptr*}$ (slen,b)

(T, a * b)

T : int	(T,a) $*_T$ (T,b)

(T, a / b)

T : int	(T,a) $/_T$ (T,b)

(T, a % b)

T : int	(T,a) $\%_T$ (T,b)

(T, + a)

T : int	$+_T$ (T,a)

(T, - a)

T : int	$-_T$ (T,a)

(T, domain(a))

T : int	T ← (domain,a)
T = text	T ← (domain,a) // default decimal integer format

```
(T, null )
(T, {} )

T : int        0_T
T = text       “”
T = ip         0.0.0.0_ip
T : ptr        nothing_T

(T, null )

T : struct
    T
     {              {
      T_1 f_1;       (T_1,null),
      ...            ...
      T_n f_n;       (T_n,null)
     }              }

T : array
    T = T´[L]       { ( (T´,null), )^L }
    T = T´[]        {}
```

$(T, \{ a_1, ..., a_n \})$

```
T : struct
     T
      {                 {
       T_1 f_1;          (T_1,a_1),
       ...               ...
       T_n f_n;          (T_n,a_n),
       ...               ...
       T_p f_p;          (T_p,{}),
       ...               ...
       T_q f_q = b_q ;   (T_q,b_q),
       ...               ...
      }                 }

T : array
     T = T´[L]          { (T´,a_1), ..., (T´,a_n), ( (T´,{}), )^{L-n} }
     T = T´[]           { (T´,a_1), ..., (T´,a_n) }
```

$(T, \{ .name_1=a_1, ..., .name_n=a_n \})$

```
T : struct
     T
      {                 {
       ...               ...
       T_n f_n;          (T_n,a_k), //  f_n == name_k
       ...               ...
       T_p f_p;          (T_p,{}),
       ...               ...
       T_q f_q = b_q ;   (T_q,b_q),
       ...               ...
      }                 }
```

(T, b { .name1=a1, ..., .namen=an })

```
T : struct
    T
     {              {
      ...            ...
      T_n f_n;       (T_n,a_k), //  f_n == name_k
      ...            ...
      T_p f_p;       (T,b).f_p ,
      ...            ...
     }              }
```

(T, &a)

```
T : ptr        T ← (LVPtr,a)
T = ptr*       (LVPtr,a)
```

(T, v)

```
T = LVPtr      &v
T != LVPtr     T ← ∗  &v
```

(T, ∗a)

```
T = LVPtr      (ptr*,a)
T != LVPtr     T ← ∗  (ptr*,a)
```

(T, a → name)

```
T = LVPtr      &(ptr*,a)→name
T != LVPtr     T ← ∗  &(ptr*,a)→name
```

(T, a.name)

T = LVPtr	&(LVPtr,a)→name
T != LVPtr	T ← ∗ &(LVPtr,a)→name

(T, a[b])

T = LVPtr	(ptr^*,a+b)
T != LVPtr	T ← ∗ (ptr^*,a+b)

type casts

int ← int′

text ← int ip

struct ← struct′

ptr^* ← array