软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(8):2285−2308 [doi: 10.13328/j.cnki.jos.005962]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种包解析器硬件配置描述语言及其编译结构*

李璜华[1], 李 凌[1], 赵 宇[2], 王生原[1], 李翔宇[3]

[1](清华大学 计算机科学与技术系,北京 100084)

[2](北京信息科技大学 理学院,北京 100192)

[3](清华大学 微电子学研究所,北京 100084)

通讯作者: 王生原, E-mail: wwssyy@mail.tsinghua.edu.cn

摘 要: 设计了一种用于实现可重构网络数据包解析器的专用硬件配置描述语言 P3.由于要有利于高安全等级网络的实现,侧重于从高可信性角度进行语言设计,包括形式化定义该语言的类型系统和操作语义,以及设计其可信编译结构.基于对可重构硬件基本需求的充分理解,从软硬件协同角度出发,最终明确了 P3 语言的核心特性及其编译器 P3C 的可信编译结构.由于可重构数据包解析器是软件定义网络(SDN)、可编程数据平面的重要一环,因此,实现 P3C 的可信编译结构将对 SDN 的安全性具有重大意义.期待 P3C 项目的开展能够促进网络与形式化领域相关工作的进一步研究.

关键词: 领域专用语言;可重构数据包解析器;形式语义;可信编译;软件定义网络

中图法分类号: TP314

中文引用格式: 李璜华,李凌,赵宇,王生原,李翔宇.一种包解析器硬件配置描述语言及其编译结构.软件学报,2020,31(8):2285−2308. http://www.jos.org.cn/1000-9825/5962.htm

英文引用格式: Li HH, Li L, Zhao Y, Wang SY, Li XY. Specification language for packet parsers and its compiler architecture. Ruan Jian Xue Bao/Journal of Software, 2020,31(8):2285−2308 (in Chinese). http://www.jos.org.cn/1000-9825/5962.htm

Specification Language for Packet Parsers and Its Compiler Architecture

LI Huang-Hua[1], LI Ling[1], ZHAO Yu[2], WANG Sheng-Yuan[1], LI Xiang-Yu[3]

[1](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[2](School of Applied Science, Beijing Information Science and Technology University, Beijing 100192, China)

[3](Institute of Microelectronics, Tsinghua University, Beijing 100084, China)

Abstract: This paper designs a domain-specific language P3 for reconfigurable protocol-independent packet parsers. Due to the requirement to facilitate the implementation of a high-security network, the language is designed from the perspective of high trustworthiness, including the formal definition of type system and operational semantics of the language and its trusted compiler architecture. Based on the full understanding of the basic requirements of the reconfigurable hardware, from the view of hardware-software codesign, the core characteristics of P3 language and its trusted compiler architecture named P3C are finally defined. As the reconfigurable packet parser is an important part of SDN and programmable data plane, implementing the trusted compiler architecture of P3C will be of great significance to the security of SDN. It is expected that the development of P3C project will promote the further research in the field of network and formal method.

Key words: domain-specific language; reconfigurable packet parser; formal semantics; trustworthy compiler; software-defined networking

* 基金项目: 核高基国家科技重大专项(2017ZX01030-301-003)
Foundation item: CHB National Science and Technology Major Project of China (2017ZX01030-301-003)
本文由“面向新兴系统的形式化建模与验证方法”专题特约编辑陈振邦副教授、冯新宇教授、刘志明教授推荐.
收稿时间: 2019-08-31; 修改时间: 2019-11-02; 采用时间: 2019-12-30; jos 在线出版时间: 2020-04-18

互联网的实现,需要相应的网络设备对网络数据包进行处理和转发.考虑到处理速度,目前大多数的交换机芯片都是采用固定功能的专用硬件实现,如 Broadcom 公司的 Trident[1]芯片.这样的交换机一经部署,就只能支持固定的网络协议,它具有很明显的缺陷.

- 首先,近年来,互联网产业发展迅猛,各种新型网络技术层出不穷,出现了大量的新型协议如 MPLS[2], GRE[3]等.为了适应这些新的网络协议,运营商只能反复更新网络的中转设备.然而开发新的硬件并推广一个新的硬件标准是十分花费时间和金钱的,如VxLAN[4]协议在其被提出3年后才被新的硬件标准支持.
- 其次,这种固定功能的硬件设备往往采用“自底向上”的设计模式,在这种设计模式下,设计者需要首先考虑芯片的数据清单,确认硬件的资源能满足设计的要求后,再规划好如何利用这些资源,然后在这之上设计功能的实现.这对于设计者来说是十分不方便的,因为设计者往往更自然地以“自顶向下”的方式来考虑问题,即设计数据包的解析、处理和转发等过程的逻辑,而不愿意考虑硬件的实现细节.

为了提高网络中转设备的通用性和灵活性、简化开发者的开发过程,相关研究人员做出了大量的努力.

软件定义网络(software-defined networking,简称 SDN)[5]将控制平面和数据平面分离,给予操作者对控制平面进行编程的能力,由控制平面通过统一、标准的接口对不同的网络设备进行配置和管理.通过这种方式,SDN 把网络设备逐渐变为可编程的软件平台,操作者可以“自顶向下”地通过编程决定各网络设备的行为.

2013 年,Bosshart 等人提出了可编程交换机的抽象转发模型[6],为实现可编程数据平面提供了可能.在这个模型中,数据包在进行转发时会通过一连串的可重配置的匹配动作表(reconfigurable match-action tables,简称 RMT),通过对这些灵活的匹配动作表的重新配置,就可以使交换机支持不同的对数据包的处理、转发方式.2014 年,Bosshart 等人在抽象转发模型的基础上进一步提出了 P4 语言[7],为可编程交换机提供了一种高级编程语言.P4 语言专注于描述数据包在匹配动作表中被处理的逻辑过程,并通过相应的编译器实现其到具体硬件的配置,符合“自顶向下”的设计模式.P4 具有对数据平面的重配置能力、协议无关性以及目标硬件无关性,极大地提高了对交换机编程的通用性和灵活性,也方便代码的跨平台迁移.自此,可编程数据平面逐渐兴起,SDN 的潜能得到了更大的开发.

软件的安全性始终是软件工作者感兴趣的话题,随着 P4 的兴起,软件工作者也开始尝试用形式化方法来验证 P4 程序的可靠性,从而提高 SDN 的安全性.2018 年,Liu 等人[8]设计并开发了一个用于 P4 程序正确性验证的工具 P4v,该工具首先将 P4 程序转译为 GCL(guarded command language),然后通过添加特定注释并利用符号执行技术来验证 P4 程序的逻辑正确性.作者在他们的工作中指出,相关的开源 P4 编译器是存在 bug 的,因此他们构建了一个独立的编译器前端包括解析器、类型检查以及到 GCL 的翻译器.尽管这极大地提高了 P4 程序的可信度,但 P4 程序和转换的 GCL 之间的一致性仍然是难以保证的,所以 Liu 等人也把设计一个可验证的编译器甚至可验证的硬件作为感兴趣的未来研究工作.Lopes 等人研究实现了一个基于 Z3 求解器的原型工具[9],可用于验证 P4 编译过程(verifying P4 compilation).Kheradmand 等人为 P4 语言定义了一种基于 *K*-框架的形式语义[10],并借助 *K*-框架近期开发的原型工具 KEQ[11],给出一种实现 P4 编译过程的翻译确认[12]解决方案.

交换机在工作时,第 1 步是实现数据包的解析.为了实现解析器的快速设计,Benacek 等人[13]将 P4 语言编写的解析器程序映射到固定的硬件架构中,可以快速生成解析器对数据包进行灵活的解析,但没有实现资源的复用.李翔宇课题组为此提出了一种通过静态配置可以实现不同解析逻辑的数据包解析器基本处理单元(process element,简称 PE)[14],经由编译器(本文工作)生成的配置文件配置后,可用于任何一层协议(可自定义)的解析,并通过流水线式级联的方式来搭建可以支持各种协议集合的可重构数据包解析器,具有更少的资源占用率、更好的性能和更高的灵活性.本文工作的出发点之一就是围绕这一可重构数据包解析器进行软硬件协同设计,提出并实现一种与硬件设计匹配的硬件配置描述语言.

本文工作的大背景是服务于一种高安全交换芯片的设计与实现,需要从高安全可信的角度设计和实现这种硬件配置描述语言,因此提出 P3 语言(specification language for reconfigurable protocol-independent packet parsers).P3 语言比 P4 语言要简单很多,仅面向上述可重构数据包解析器的设计.P4 语言描述的是整个数据平面,

包括数据包的解析、处理和转发过程,对于仅仅数据包解析器来说,使用 P4 语言太过庞大冗余,因此实现 P4 语言的安全可信编译器难度大、周期长,不能满足本文工作的项目要求.为实现 P3 语言,我们设计了具有高安全可信等级的 P3C 编译器结构,可极大地提高网络数据包解析以至于 SDN 的安全性.

本文第 1 节对所针对的可重构数据包解析器硬件设计进行概述.第 2 节将介绍 P3 语言的核心特性.第 3 节介绍对 P3 类型系统进行严格规范以实现静态语义检查的设计工作.第 4 节为 P3 定义一种操作语义风格的形式化动态语义.第 5 节介绍 P3 语言编译器 P3C 的可信编译结构.

1 可重构数据包解析器硬件设计概述

本文工作的大背景是服务于一种高安全交换芯片的设计与实现,该芯片的可重构 Parser 模块用于对输入数据包的头部信息(以太报文的 L2~L4 等头部)加以识别,得到并修改后续业务处理所需要的相关字段.可重构 Parser 的总体结构如图 1 所示,整体是一个流水线结构,由相同的基本处理单元(PE)级联而成,这些 PE 被配置单元(PE_config)写入的配置文件配置成用于不同层的协议解析单元,配置文件由编译器生成.从 L2 到 L4,包括各级子层(L2S,L3S),每层的协议解析对应一级 PE.PE 间通过帧寄存器(frame,也称为 PKT 寄存器)和中间值寄存器堆(intermediates register file,简称 IRF)进行数据交换.输入的数据包包头(PKT_header)会在 frame 数据通路中进行传输,由一系列 PE 单元对包头封装中的协议进行逐层解析,并将提取出的关键字段及处理结果存放到 IRF 中.

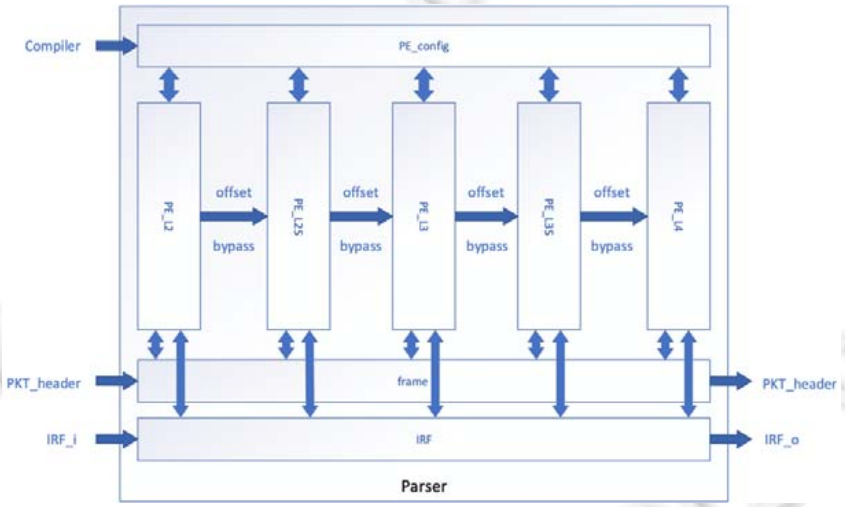

Fig.1 Architecture of a packet parser

图 1 包解析器的整体架构

下面我们以一个简化的业务需求为例,对该 Parser 进行直观说明.假设该解析器的输入为报文的头部,我们需要从该报文头部中解析得到一些描述字段——包括报文类型(IPV4/IPv6/MPLS…)、L2/L3/L4 封装头的偏移量、是否采用 VLAN 等.该报文头部首先进入 frame 数据通路,并通过 frame 传入各个 PE 之中.接着,各个 PE 会依次对该报文头部进行处理:首先,在 PE_L2 层中可以提取出 L2 封装头的信息,包括 L2 封装头的长度和是否采用了 VLAN,并将之存入 IRF 中;然后,根据 L2 封装头的长度给下一层传送偏移量(offset)信息,PE_L2S 则根据该 offset 调整解析包头的区间(即对齐);之后,每层处理时也皆是如此.除了 offset 信息,各 PE 还会给下一层传送一个 bypass 信号,用于实现跨层处理,比如 Ethernet+IPv4+…的数据包,在 PE_L2 层完成了 Ethernet 的解析,而 IPv4 是在 PE_L3 层解析的,因此需要跳过 L2 子层,这时,PE_L2 发出的 bypass 信号就会告诉 PE_L2S 不用处理,直接跳过.按照这种方式,各 PE 依序进行处理,L3 层的报文类型(IPV4/IPv6/MPLS…)和 L3 封装头偏移量将在 PE_L3 处理完后存入 IRF 中,L4 封装头偏移量则在 PE_L4 处理完后得到.

该数据包解析器的核心部分是一系列的可重构 PE 模块,这些 PE 模块通过编译器生成的配置文件配置实

现不同的硬件解析逻辑,从而支持对不同协议包头的解析.

每一层的包头解析在 PE 中可以概括为提取、查找、动作这 3 个过程,图 2 为 PE 的硬件示意图,包含了若干 *Cell* 单元、*PE_bypass* 模块和 *Offset* 模块等.其中,核心模块是 *Cell* 单元,包头解析的 3 个过程都在 *Cell* 单元中完成,它共包含 *PA*,*PB*,*PC* 和 *Action* 几个部分.*PA* 模块完成“提取”过程,用来实现特定的数据域的提取,它根据前一级 *PE* 输出的关键字段偏移量 *offset* 和 *IRF* 中的一些状态信息对本级 *frame* 寄存器中的数据包头部字段进行提取.*PA* 层的输出送入 *PB* 层,*PB* 层的核心是一个特征匹配查找表,它将 *PA* 提取出来的关键字与一系列特征匹配模版进行对比,找到匹配的模版,模版的编号即对应协议的类型.根据协议类型和一些字段值,可以判断下层需要解析的协议,若需要跨层处理,*PB* 就会往 *PE_bypass* 模块发送一个信号表示跳过下一层.*PB* 中的特征模版完全根据 *PE_config* 写入的配置文件配置而成.*PC* 的核心是一组查找表,它的输入是 *PB* 输出的协议类型编号,输出为根据类型编号索引得到的应该执行的操作指令,和下一级解析时所需要设置的 *offset*.同 *PB* 一样,*PC* 的表项内容也是由 *PE_config* 配置而成.*Action* 是一个指令执行模块,根据 *PC* 索引的操作指令执行相应动作,并把结果写入 *IRF* 中.为了提高硬件资源的复用率和通用扩展性,硬件设计者设计了两种不同的 *Cell* 形式:*Cell_A* 和 *Cell_B*,以 1 个 *Cell_A*、0~2 个 *Cell_B* 的形式组成 *PE* 实现并行解析,其中,计算 *offset* 和给 *PE_bypass* 传递信号只在 *Cell A* 中进行,*Cell B* 仅包含其他解析过程.

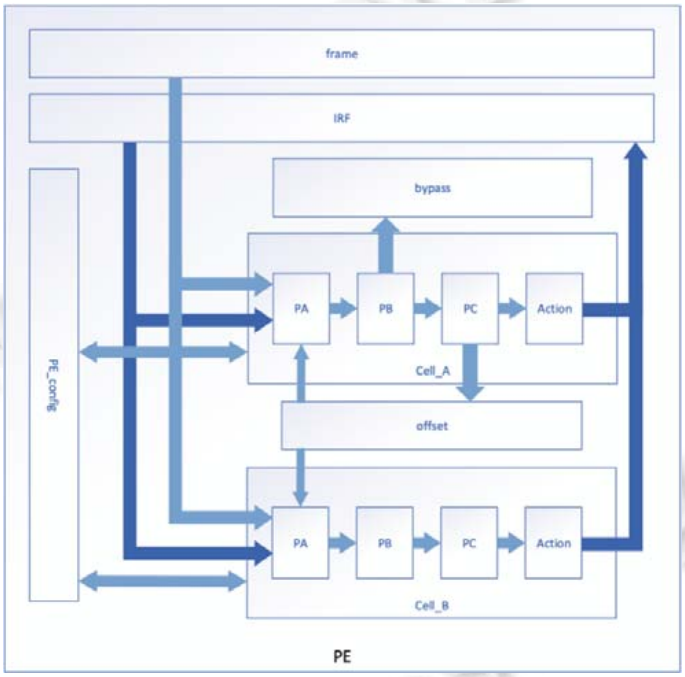

Fig.2 Architecture of a *PE*

图 2 *PE* 结构

2 P3 语言的设计

基于前面一节所述的可重构解析器架构,本文设计了一种可重构数据包解析器的专用硬件配置描述语言 P3,作为较抽象的语言,可用于“自顶向下”地描述一个自定义的数据包解析器.

2.1 P3基本概念

对应于数据包在该解析器中的解析过程,P3 语言主要描述了以下两个抽象概念.

- 层(layer):描述了数据包在解析器中进行逐层解析时,经过每一级 *PE* 时的解析操作.
- 协议(protocol):声明了在解析器中需要进行解析的网络协议模版,描述了协议中应包含的各个协议域

字段信息以及可选的在层中的固定解析操作.

下面,本文通过一个例子用 P3 语言定义一个简单的 IPv4 数据包解析器,它仅包含 L2 和 L3 层(对应图 1 中的 PE_L2 和 PE_L3),以此来说明 P3 语言描述的两个抽象概念.

2.2 层(layer)

P3 描述了数据包解析器对协议包头的逐层解析,每一层对应一个可重构单元 PE.在每一层中,P3 声明了需要匹配的协议模版,并且具体描述了在该 *PE* 中的每个 *Cell* 单元里需要执行的匹配查找条件以及相应动作.P3 抽象了 *Cell* 中的硬件细节,编程者无需关注 *PB*,*PC* 或 *Action* 模块如何与配置文件相对应,只需用人类容易理解的语法声明需要匹配的协议类型以及用条件语句描述匹配条件并给予执行动作,编译器会生成相应的配置文件完成对这些硬件模块的配置.

在图 3 中的代码片段中,P3 描述了数据帧在 L2 层中的解析过程.

```
l2{
  //l2 layer IRFs
  ARegisters {                                  //定义cell A的IRF寄存器访问
    IRF_tag_type_2b=IRF[23:16];                 //用于set
    IRF_pkt_type_3b=IRF[31:24];                 //用于set
    IRF_l2_protocol_flag_type_8b=IRF[39:32];    //用于set
    IRF_outer_vlan_high=IRF[7:0];               //用于mov
    IRF_outer_vlan_low=IRF[15:8];               //用于mov
  }
  B0Registers {                                 //定义cell B0的IRF寄存器访问
    IRF_l2_type=IRF[7:0];
  }
  ethernet eth;                                 //声明协议ethernet包头的一个实例eth
  ieee802-1qTag vlan;                           //声明协议ieee802-1qTag包头的一个实例vlan
  cellA {
    if ((eth.etherType==0x8100) && (vlan.etherType==0x0800))
      length=eth.length+vlan.length;            //计算本层的Length值
      next_header=ipv4;                         //给出本层的NextHader标识(协议名称)
      bypass=1;                                 //设置本层的Bypass值
      action={
        mov IRF_outer_vlan_high++IRF_outer_vlan_low, vlan.pcp++vlan.cfi++vlan.vid;
        set IRF_tag_type_2b, 1;                 //single-tagged(st)
        set IRF_pkt_type_3b, 0;
      }
    elseif (eth.etherType==0x0800)              //ipv4
      length=eth.length;                        //计算本层的Length值
      next_header=ipv4;                         //给出本层的NextHader标识(协议名称)
      bypass=1;                                 //设置本层的Bypass值
      action={
        set IRF_tag_type_2b, 0;                 //untagged(ut)
        set IRF_pkt_type_3b, 0;
    }
    else
      ...
    endif
  }
  cellB0 {
    if (eth.dmac==0xFFFFFFFFFFFF)
      set IRF_l2_type, 3;                       //bc
    elseif (eth.dmac[40]==1)
      set IRF_l2_type, 2;                       //mc
    else
      set IRF_l2_type, 1;                       //sc
    endif
  }
}
```

Fig.3 An example of L2 layer declaration

图 3 L2 层声明示例

首先,*XRegister* 代码块声明了在 *CellX* 中会涉及到的所有 IRF 寄存器访问标识符(*X* 可为 *A*,*B*0 或 *B*1,对应 1 个 *CellA* 和 0~2 个 *CellB*);接着,声明在 L2 层中需要进行匹配的协议包头模版(ethernet,ieee802-1qTag)实例,所有出现的协议包头模版均由用户自定义,定义规则会在第 2.3 节中介绍,相关代码见后文图 5;最后,Cell*X* 代码块描述了数据帧在相应 *CellX* 中的解析过程,具体包括由 if 语句指定的匹配查找条件、设置重要参数和一些基本寄存器操作.设置重要参数指的是 *length*,*next_header* 和 *bypass* 这 3 个参数.

- *length* 表示该层协议的包头长度,用于给 *Offset* 单元设置下一层协议包头匹配的偏移量;
- *next_header* 表示下一层需要解析的协议类型;
- *bypass* 用于设置 *PE_bypass* 单元,从而实现协议的跨层处理(如在 L2 层中,*bypass*=1 表示下次解析 L3 层协议).

基本寄存器操作指的是在 *action* 代码块(*action*{·}也可以省略不写)中的下列 4 类简单寄存器指令(对应下一节图 8 中的抽象语法定义).

- *set Reg*,*e*:将寄存器访问表达式 *Reg* 表示的寄存器区间置为表达式 *e* 的值(常量值);
- *mov RegM*,*e*:将表达式 *e*(可能含有对多个协议域的访问)的值传送到寄存器访问表达式 *RegM*(可能对应多个寄存器访问表达式所描述字段的合并)所描述的寄存器区间;
- *lg Reg*,*a*,*b*:若 *a* 大于 *b*,则将寄存器访问表达式 *Reg* 表示的寄存器区间置为 1;否则置为 0;
- *eq Reg*,*a*,*b*:若 *a* 等于 *b*,则将寄存器访问表达式 *Reg* 表示的寄存器区间置为 1;否则置为 0.

在本例中,P3 对 L3 层的描述与 L2 层相似,主要是对 IPv4 协议包头的匹配解析,相应实现可参考图 4 左侧代码片段.特别地,由于 IPv4 协议包头的解析规则相对固定,所以可以直接把它的匹配查找条件、设置重要参数和一些寄存器操作写在协议声明部分(见第 2.3 节),从而可以在不同的层中重用而使得层的描述部分更加简洁.

```
l3 {
    ARegisters {
      IRF_l3_type=IRF[23:16];          //用于set
      IRF_TOS_8b=IRF[7:0];             //用于mov
      IRF_ttl_8b=IRF[15:8];            //用于mov
      IRF_TTL_EXP=IRF[199:192];        //用于alu
    }
    B0Registers {…}
    B1Registers {…}
    ipv4 v4;
    cellA {
      mov IRF_TOS_8b, v4.diffserv;
      mov IRF_ttl_8b, v4.ttl;
      lg IRF_TTL_EXP, 2, v4.ttl;
    }
    cellB0 {
      if (v4.srcAddr==0x00000000)
        …
      endif
      if (v4.dstAddr==0xffffffff)
        …
      endif
    }
    cellB1 {
      if (v4.flagOffset == 0)
        if (v4.flags == 0)
          …
      endif
    }
    …
}
```

```
protocol ipv4 {
  fields={
    version: 4;                  //1
    ihl: 4;                      //1
    diffserv: 8;                 //2
    totalLen: 16;                //3,4
    identificaiton: 16;          //5,6
    flags: 3;                    //7
    fragOffset: 13;              //7,8
    ttl: 8;                      //9
    theProtocol: 8;              //10
    hdrChecksum: 16;             //11,12
    srcAddr: 32;                 //13,14,15,16
    dstAddr: 32;                 //17,18,19,20
    options: *;
  }
  if (ihl==5)
    length=20;
    set IRF_l3_type[3], 0;
  elseif (ihl==6)
    length=24;
    set IRF_l3_type[3], 1;
  elseif (ihl == 7)
    length = 28;
    set IRF_l3_type[3], 1;
  else
    …
  endif
  if (theProtocol==2)
    …
  elseif (theProtocol==4)
    …
  endif
}
```

Fig.4 Examples of L3 layer declaration and IPv4 protocol declaration

图 4 L3 层声明和 IPv4 协议声明示例

由此可见,P3 在对每层的描述中主要包含 3 个部分.

- 声明在 *CellX* 中可能涉及到的 *IRF* 寄存器访问标识符;
- 按顺序声明需要进行匹配的协议模版实例;
- 描述在 *CellX* 中进行协议匹配的条件以及相应解析操作.

2.3 协议(protocol)

借鉴了 P4 的部分语法[15],P3 通过声明一个字段名以及其位宽的有序集合来定义一个协议包头模版,并用一个 *length* 参数(不同于层中的 *length*)来显式指定包头的长度(*length* 参数的取值不小于所有字段总的字节数).同时,协议声明中还允许加入层解析中的自定义解析条件,用来描述该协议包头相对固定的解析规则,从而可以简化对层部分的描述.

图 5 是标准以太网和虚拟局域网的协议声明部分.

```
protocol ethernet {
    fields={
        dmac: 48;              //48bit
        smac: 48;
        etherType: 16;
    }
    length=14;                 //14Bytes, 14×8bit
    …                          //list of statements
}
```

```
protocol ieee802-1qTag {
    fields={
        pcp: 3;
        cfi: 1;
        vid: 12;
        etherType: 16;
    }
    length=4;                  //4Bytes, 4×8bit
    …                          //list of statements
}
```

Fig.5 Examples of the header's fields in two protocol declarations

图 5 两个协议声明中的包头数据域示例

IPv4 的协议包头也按照这种方式进行声明,不过由于其包头长度不固定,因此加入条件语句进行描述,部分声明如图 4 右侧代码片段所示,代码片段中的条件语句描述了 IPv4 协议包头的变长情况.根据 IPv4 协议的包头结构[16],第 4 个~第 7 个比特(第 0 个开始)对应的协议域字段(样例中的 *ihl*)指定了 IPv4 包头的长度,因此,这里的 *length* 参数是由 *ihl* 字段的值来决定的.另外要注意,协议声明中出现的所有寄存器访问(*IRF_l3_type*)都应在层声明中的 *ARegisters* 部分预先定义.

结合层的描述与协议声明部分,在这个简单的例子中,数据帧在 L2 层完成了对以太网协议包头和虚拟局域网包头的解析,执行逻辑如下:数据帧进入 L2 层后,首先让包头匹配以太网包头(ethernet)模版,进行偏移后,再匹配 IEEE802.1Q 标准的 VLAN 协议模版.此时 *eth.etherType* 提取出当前包头的第 13、第 14 字节,而 *vlan.etherType* 提取出当前包头的第 17、第 18 字节.根据以太网帧和 IEEE802.1Q 包头的协议域字段含义[16],若 *eth.etherType* 为 0*x*8100,则表示这是一个加了 802.1Q 标签的帧,以太网类型在 *vlan.etherType* 字段中;若为 0*x*0800,则为网际协议(IP),所以下一层需要解析 IPv4 协议.若 *eth.etherType* 为 0*x*0800,则说明它是没有加 802.1Q 标签的网际协议,同样指定下一协议包头为 IPv4.在每个条件分支中,最后都会通过 *action* 代码块中的 set 操作将解析结果保存至 IRF 的对应位置中.

在完成了对 L2 层协议的解析后,数据帧进入 L3 层.在 L3 层中完成对 IPv4 协议模版的匹配,并保存解析结果至 IRF 中.

2.4 P3语言特性

在传统网络编程时,对包头协议域字段进行读取或对寄存器进行读写是需要非常小心的事情,因为编程者需要清楚地知道相应的字段应该处于一长串二进制位的哪一段,并准确地用十六进制(或二进制)将其表示出来,一个 0/1 位的偏移就会引发巨大的错误.并且,对一大堆十六进制(或二进制)串进行查错也是十分麻烦的一件事.为了方便网络编程者使用,P3 语言包含了一些特性.

2.4.1 字段合并

在寄存器的操作中,有时编程者需要对位置相邻的寄存器区间或协议域字段进行合并,然后一次性进行读写.为此,P3 加入了一个二元运算符“++”来完成这种操作.

在图 3 的例子中,出现了以下操作:

mov *IRF_outer_vlan_high*++*IRF_outer_vlan_low*, *vlan.pcp*++*vlan.cfi*++*vlan.vid*;

表示的就是将 *vlan.pcp*,*vlan.cfi*,*vlan.vid* 这 3 个连续的协议域字段先进行合并,然后存储到 *IRF_outer_vlan_hign* 与 *IRF_outer_vlan_low* 两个标识符表示的 IRF 寄存器区间合并起来的连续区间中.需要特别注意的是,通过“++”连接起来的两个寄存器区间或协议域字段必须是连续的.

2.4.2 重新编址

在对每一层的协议包头进行解析时,每解析完一层,就要将提取出来的信息保存至公共的 *IRF* 中.为方便起见,可将 *IRF* 看作一个公共的长存储区,若采用图 6 中的固定编址方案,即根据 *IRF* 中的绝对位置进行索引,随着提取到的信息量增加,计算索引值会变得非常麻烦且容易出错.由于目标硬件解析器对每层协议的解析是由相同的 *PE* 模块完成的,而每个 *PE* 模块的内部又是由相似的几个 *Cell* 单元构成,所以我们对 *IRF* 以 *Cell* 单元为基本单位进行分段固定编址,具体如图 7 所示.

Fig.6 Global fixed address of *IRF*

图 6 *IRF* 全局固定编址

Fig.7 Sectional fixed address of *IRF*

图 7 *IRF* 分段固定编址

每一段对应一个 *Cell* 可以访问的 IRF 资源,共 24 个字节(192 个比特).由于一个 *PE* 中最多包含 3 个 *Cell* 单元(1 个 *Cell_A*、0~2 个 *Cell_B*),所以固定给每一个 *PE* 分配 3 段,共 72 个字节.对应图 1 中的需要解析到 L4 层协议的交换机结构,共包含 5 个 *PE* 模块,因此 *IRF* 的总长度为 72×5=360 字节.编址时,编程者只需根据当前位置在当前段中的相对位置进行索引,在寄存器寻址时,编译器会首先根据上下文确定当前应处于哪一段(即哪一个 *PE* 的哪一个 *Cell*),然后再加上相应的固定偏移在 *IRF* 中寻址.

另外,在一些特殊情况中,当前层可能会需要访问前一层提取到 *IRF* 中的协议包头信息,所以 P3 也允许对前一层声明过的寄存器访问标识符进行跨层访问.

3 P3 类型系统

为方便静态语义检查的实现,本文对 P3 类型系统进行了严格规范.本节给出 P3 类型系统形式化定义的核心部分.在此之前,需要先介绍 P3 抽象语法的相关内容,见第 3.1 节.

3.1 抽象语法

图 8 包括了我们所选取的部分 P3 抽象语法,用于介绍本节的 P3 类型系统以及后一节的 P3 操作语义.在抽

象语法的定义中,非终结符采用了正体,终结符和辅助关键字使用了斜体,起始符号为 *parser_spec*.*Lreglen*, *Creglen*,*Pset* 和 *Lset* 等用于声明一个 P3 规范(为方便,本文许多地方也称 P3 程序或代码)的几个参数:层寄存器长度、*cell* 寄存器长度、协议名称集以及层名称集.语法的主体是 *decl* 分支,由多个声明组成,每个声明可以是常量声明(*ConstDcl*)、全局寄存器访问声明(*RegAccSet*)、协议声明(*protocol_decl*)或层声明(*layer_action*).每一层都有局部寄存器访问的声明,包含 3 个部分(*CellARegs*,*CellB*0*Regs* 和 *CellB*1*Regs*),每个部分声明了各个 *cell* 的局部寄存器访问标识符.

每一层需要声明一个或多个协议数据包头的实例(*ProtocolDef*),在该层的各个 *cell* 规范中,可通过域名表达式(*Efield*)访问某个实例的各个域.协议声明内部,只要通过域名标识符即可以访问各个域.

语句(*stmt*)可以出现在各层的每个 *cell* 规范中,也可以出现在协议声明中.后者主要是为了规范描述在多个层中复用,各语句相当于出现在各相关层 *cell A* 中(由于域的访问需用相应实例的域名表达式替换,所以要求这种情况下相关层中最多只能声明该协议包头的一个实例).每条语句可以是描述查表逻辑的分支语句(*if_else_stmt*)、一系列的硬件指令(*action_stmt*)以及重要参数(*NextHeader*,*Length*,*Bypass*)设置(仅在 *cell A* 或在协议声明中设置).

图 8 中,后半部分内容(*expr* 之后)主要描述与寄存器和协议域访问相关的部分,是 P3 规范中最有代表性的内容,本节和下一节的类型系统和操作语义将围绕这一主线进行介绍.

```
parser_spec::=(Lreglen(n),Creglen(n),Pset(id+),Lset(id+),decl+)
decl::=ConstDcl(id,c)|RegAccSet(reg_acc_set)|protocol_decl|layer_action
protocol_decl::=(id,fields,stmt+)
fields::=(Fields((id,c)+),OptionField(id:0)
layer_action::=(id,local_reg_decll_decl+,l_actions)
l_decl::=ProtocolDef(id,id+)
local_reg_decl::=(CellARegs(reg_acc_set),CellB0Regs(reg_acc_set),CellB1Regs(reg_acc_set))
l_actions::=(CellA(stmt+),CellB0(stmt+),CellB1(stmt+))
stmt::=if_else_stmt|NextHeader(id)|Length(n)|Bypass(n)|action_stmt
expr::=c|id|unop(expr)|expr binop expr|Efield(expr,id)
       |ERegBit(expr,expr)|ERegSection(expr,expr,expr)
       |EFieldBit(expr,expr)|EFieldSection(expr,expr,expr)
       |ProtLen(id)
binop::=… |++| …
action_stmt::=instruction*
instruction::=Set(tgt_reg_acc_name,expr)|Mov(mov_reg_acc_name,expr)
             |Lg(tgt_reg_acc_name,expr,expr)|Eq(tgt_reg_acc_name,expr,expr)
reg_acc_set::=IRF(id,expr,expr)|IRF(id,expr)
tgt_reg_acc_name::=TargetRegAccName(id)
                  |TargetRegAccName(tgt_reg_acc_name,expr,expr)
                  |TargetRegAccName(tgt_reg_acc_name,expr)
mov_reg_acc_name::=MovRegAccName(tgt_reg_acc_name)
                  |MovRegAccName( mov_reg_acc_name,tgt_reg_acc_name)
```

Fig.8 Abstract syntax of P3 (partial)

图 8 P3 抽象语法(部分)

3.2 类型表达式

以下是用于定义 P3 类型系统的类型表达式:

⟨*type*⟩::=*Int*	有符号 32 位整数
|*Hexes*(*n*)	*n* 位十六进制数(串)
|*Bits*(*n*)	*n* 位二进制数(串)
|*RegAcc* (*k*,*i*,*j*)	寄存器访问类型,其中 *k* 表示当前上下文中的 *IRF* 寄存器大小,满足 $0\leqslant j\leqslant i<k$
|*FieldAcc*(*id*,*k*,*i*,*j*)	位于 *Cell* 上下文中的协议域访问类型,*id* 为协议实例标识符,*k* 为该协议实例的长度,*i* 和 *j* 为访问区间,满足 $0\leqslant i\leqslant j<k\vee(i=k\wedge j$ is undefined),析取运算的后半部分表示该协议存在 optional 域

$\|FieldAcc(k,i,j)$	位于协议上下文中的协议域选取类型,k 为该协议实例的长度,i 和 j 为访问区间,满足 $0\leqslant i\leqslant j<k\vee(i=k\wedge j$ is undefined)
$\|X$	协议类型,任意名字为 X 的协议的所有实例的类型为 X

对于常数表达式,在很多地方,我们需要计算它的值以用于合法性检查,因此,我们给所有数值类型(*Int*, *Hexes*(*n*),*Bits*(*n*))增加了一个辅助值组成了一种特别地常数类型:

$\langle type\rangle::=(\tau,i)$	组合类型,τ表示某数值类型,i 表示一个有符号 32 位整数值

3.3 类型环境

一个类型环境记录一类特定标识符在某个上下文或作用域中被赋予的类型表达式.我们分别用$\mathcal{C}$,$\mathcal{R}$,$\mathcal{L}$和$\mathcal{P}$分别表示全局常量类型环境、特殊的全局寄存器访问类型环境(见下文解释)、层(layer)的局部类型环境和一个协议(protocol)的局部类型环境.层局部类型环境$\mathcal{L}$内部又嵌套有$\mathcal{L}_A$,$\mathcal{L}_{B0}$ 和$\mathcal{L}_{B1}$ 这 3 个局部类型环境,对应于 *CellA*,*CellB*0 和 *CellB*1.某些时候,会用$\mathcal{L}_{id}$ 和$\mathcal{P}_{id}$ 来表示在 *id* 这个标识符所表示的层或协议上下文中的局部类型环境.

我们引入特殊类型环境$\mathcal{R}$,用来记录当前层前面一层的只读寄存器访问标识符的类型,它会随着层间切换而动态更新.在一开始时,$\mathcal{R}$由全局寄存器访问的声明进行初始化,这些全局寄存器访问标识符在第 1 个层中可见.当新的层声明出现时,$\mathcal{R}$更新为上一层的 *cell* 局部环境$\mathcal{L}_A$,$\mathcal{L}_{B0}$ 和$\mathcal{L}_{B1}$ 的特殊并集(参见第 3.4 节).

最后,为了保证一致性,我们还定义了几个全局参数,包括层寄存器大小、*cell* 寄存器大小、协议集合以及层集合.为此,我们引入了 4 个特殊的全局类型环境$\mathcal{L}reglen$,$\mathcal{C}reglen$,$\mathcal{P}set$ 和$\mathcal{L}set$,以及为了方便,我们用符号$\mathcal{G}$来表示它们的组合,即$\mathcal{G}=(\mathcal{L}reglen,\mathcal{C}reglen,\mathcal{P}set,\mathcal{L}set)$.

3.4 类型规则

P3 包含了一系列类型规则以实现静态类型检查,完整的类型规则可参见 P3 设计文档[17].本文会对其中一些有代表性的规则进行介绍,这些规则主要是围绕着寄存器的读写以及协议域的访问来展开.

在这些类型规则中,除了常规断言,最主要的断言形式包括以下 3 种.

- $\mathcal{E}\vdash e{:}A$;
- $\mathcal{E}\vdash D$;
- $\mathcal{E}\vdash\Diamond$.

其中,$\mathcal{E}$为一个或一系列类型环境,e 为表达式(含各类标识符),A 为类型表达式(见第 3.2 节),D 对应于解析器规范各个部分的定义.$\mathcal{E}\vdash e{:}A$ 表示在环境(集)$\mathcal{E}$下,e 类型良好且具有类型 A.$\mathcal{E}\vdash D$ 表示在环境(集)$\mathcal{E}$下,D 是类型良好的.$\mathcal{E}\vdash\Diamond$表示某个环境$\mathcal{E}$是良好定义的(标识符的类型值唯一,或者说域中无重名标识符).

3.4.1 寄存器和协议域访问类型环境的初始化

规则(1)用来初始化 *cellA* 中定义的寄存器访问.

$$\frac{\mathcal{G}\vdash\mathcal{C}reglen(n)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}'_A\vdash e_1:(Int,n_1)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}'_A\vdash e_2:(Int,n_2)\ 0\leqslant n_2\leqslant n_1<n\ id\notin dom(\mathcal{L}'_A)\\ \forall id'\in dom(\mathcal{L}'_A).(\mathcal{L}'_A\vdash id':RegAcc(n,n'_1,n'_2)\rightarrow n'_1<n_2\vee n_1<n'_2)\ \mathcal{L}_A=\mathcal{L}'_A\cup\{id:RegAcc(n,n_1,n_2)\}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_A\vdash IRF(id,e_1,e_2)} \tag{1}$$

这些定义形如 $id=IRF(e_1,e_2)$,参见图 3 中 L2 层的 *ARegisters* 声明部分;在抽象语法中,用一个 $IRF(id,e_1,e_2)$ 节点来表示这样的定义.将这样的寄存器访问定义加入环境$\mathcal{L}_A$时需要检查 3 点.

- 一是检查访问的寄存器区间$[n_1,n_2]$不能超过当前 *cell* 的 *IRF* 的范围(根据习惯,区间上下界由大到小给出).全局环境$\mathcal{G}$中的抽象语法节点 $\mathcal{C}reglen(n)$指定了当前 *cell* 的寄存器长度为 n,表达式 e_1 和 e_2 的值为 n_1 和 n_2,指定了访问的区间上下界,若 $0\leqslant n_2\leqslant n_1<n$,则满足条件;
- 二是检查当前定义的标识符 *id* 没有被声明过,即 $id\notin dom(\mathcal{L}'_A)$;
- 最后要检查定义的各个寄存器区间是不重叠的,以保证读写寄存器的安全性,即对于所有已声明过的寄存器区间 $[n'_1,n'_2]$ 应该与当前声明的区间$[n_1,n_2]$没有公共部分,可表述为 $n'_1<n_2\vee n_1<n'_2$.

另外,在 *cellB*0 或 *cellB*1 中的寄存器访问定义规则与 *cellA* 中一致,只需将$\mathcal{L}_A$替换为$\mathcal{L}_{B0}$或$\mathcal{L}_{B1}$即可.后面规则中的类似情况,将不再重复解释.

对 *cellA* 中寄存器的某一位的访问定义规则为规则(2),可简单将其看作规则(1)的特例,即 n_2=n_1 的情况.

$$\frac{\mathcal{G}\vdash \mathcal{C}reglen(n)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}'_A\vdash e:(Int,k)\ 0\leqslant k<n\ id\notin dom(\mathcal{L}'_A)\quad \forall id'\in dom(\mathcal{L}'_A).(\mathcal{L}'_A\vdash id':RegAcc(n,n'_1,n'_2)\rightarrow n'_1<k\vee k<n'_2)\ \mathcal{L}_A=\mathcal{L}'_A\cup\{id:RegAcc(n,k,k)\}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_A\vdash IRF(id,e)} \tag{2}$$

P3 语言允许当前层中的 *cell* 单元访问前一层的寄存器空间,可认为是一类全局寄存器访问空间.为此,我们使用了特定的全局类型环境$\mathcal{R}$.规则(3)用来描述层切换时环境$\mathcal{R}$的初始化.

$$\frac{\begin{array}{c}\mathcal{G}\vdash Lreglen(n)\ \mathcal{G}\vdash Creglen(k)\ n=3*k\\ \mathcal{R}=\{id:RegAcc(n,2*k+n_1,2*k+n_2)\mid id:RegAcc(k,n_1,n_2)\in\mathcal{L}_A\}\cup\\ \{id:RegAcc(n,k+n_1,k+n_2)\mid id:RegAcc(k,n_1,n_2)\in\mathcal{L}_{B0}\}\cup\\ \{id:RegAcc(n,n_1,n_2)\mid id:RegAcc(k,n_1,n_2)\in\mathcal{L}_{B1}\}\end{array}}{\mathcal{R}\vdash\Diamond} \tag{3}$$

即在离开某一层的上下文环境时,需要把当前层的$\mathcal{L}_A$,$\mathcal{L}_{B0}$ 和$\mathcal{L}_{B1}$ 环境进行合并,然后更新至环境$\mathcal{R}$.具体步骤如下:首先,全局环境中的 *Lreglen*(*n*)和 *Creglen*(*k*)分别指定了当前层中的层寄存器长度 *n* 和 *Cell* 寄存器长度 *k*,每层的寄存器会平均分为 3 段给环境$\mathcal{L}_A$,$\mathcal{L}_{B0}$ 和$\mathcal{L}_{B1}$,因此 *n*=3*k*;然后,需要将每段 *Cell* 寄存器中的地址转换为层寄存器中的地址,在层寄存器的地址空间[*n*−1,0]中,$\mathcal{L}_{B1}$ 对应[*k*−1,0]部分,$\mathcal{L}_{B0}$ 对应[2*k*−1,*k*]部分,$\mathcal{L}_A$ 对应[3*k*−1,2*k*]部分,因此,在合并$\mathcal{L}_A$,$\mathcal{L}_{B0}$ 和$\mathcal{L}_{B1}$ 时,需要对其中声明的寄存器区间分别加上 2*k*,*k* 和 0 的偏移.另外值得注意的是,在第 1 层开始处理的时刻,环境$\mathcal{R}$由全局寄存器访问定义来初始化,相应的规则类似于规则(1)和规则(2).

规则(4)用于初始化协议层类型环境$\mathcal{P}$.

$$\frac{\begin{array}{c}flds=((fid_1:c_1),...,(fid_k:c_k))\ ofld=(ofid:0)\ \forall i:1\leqslant i\leqslant k.(\phi\vdash c_i:(Int,n_i))\\ n=n_1+n_2+...+n_k\ \ \forall i(1\leqslant i\leqslant k\rightarrow n_i>0)\\ \forall i,j(1\leqslant i<j\leqslant k\rightarrow fid_i\neq fid_j)\ \forall i(1\leqslant i\leqslant k\rightarrow fid_i\neq ofid)\\ \mathcal{G}\vdash\Diamond\ \mathcal{C}\vdash\Diamond\ \mathcal{R}\vdash\Diamond\ \mathcal{L}\vdash\Diamond\ \mathcal{L}_A\vdash\Diamond\ \mathcal{P}'\vdash\Diamond\ \forall i(1\leqslant i\leqslant k\rightarrow fid_i\notin dom(\mathcal{P}'))\ ofid\notin dom(\mathcal{P}')\\ \mathcal{P}=\mathcal{P}'\cup\{fid_i:FieldAcc(n,n_1+...+n_{i-1},n_1+...+n_i-1)\mid 1\leqslant i\leqslant k\}\cup\{ofid:FieldAcc(n,n,null)\}\end{array}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_A,\mathcal{P}\vdash(Fields(flds),OptionField(ofld))} \tag{4}$$

$\mathcal{P}$中域名标识符的类型为 *FieldAcc*(*k*,*i*,*j*),记录了协议包头固定长度数据域(除 option 域外)的长度(*k*)及各个域的起始和结束位置.本文对 option 域标识符的类型作了特殊技术处理,假设其长度为 0.在针对协议域定义时,需要检查所有定长域的长度为正整数,以及所有的协议域名没有重复.

3.4.2 表达式中寄存器和协议域的访问

表达式中寄存器访问标识符的类型已经由规则(1)和规则(2)进行过初始化,可以直接从环境中得到,如在 *cell* 局部类型环境中,可由下式得到.

$$\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash rid:RegAcc(n,n_1,n_2).$$

在进行寄存器访问时,我们可能需要选取已有寄存器访问的一段区间或某一位进行访问,规则(5)和规则(6)分别描述了这两种情况.

$$\frac{\mathcal{G}\vdash\mathcal{C}reglen(n)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash e_1:RegAcc(n,n_1,n_2)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash e_2:(Int,n')\quad \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash e_3:(Int,n'')\ 0\leqslant n_2\leqslant n_1<n\ 0\leqslant n''\leqslant n'\leqslant n_1-n_2\ \mathcal{L}_C\text{ is }\mathcal{L}_A,\mathcal{L}_{B0}\text{ or }\mathcal{L}_{B1}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash ERegSection(e_1,e_2,e_3):RegAcc(n,n_2+n'',n_2+n')} \tag{5}$$

$$\frac{\mathcal{G}\vdash\mathcal{C}reglen(n)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash e_1:RegAcc(n,n_1,n_2)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash e_2:(Int,n')\quad 0\leqslant n_2\leqslant n_1<n\ 0\leqslant n'\leqslant n_1-n_2\ \mathcal{L}_C\text{ is }\mathcal{L}_A,\mathcal{L}_{B0}\text{ or }\mathcal{L}_{B1}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash ERegBit(e_1,e_2):RegAcc(n,n_2+n',n_2+n')} \tag{6}$$

规则(5)中,*ERegSection*(e_1,e_2,e_3)表示对寄存器访问 e_1 的第 e_3 到第 e_2 位子区间进行访问.具体过程如下:设该

Cell 占用的 *IRF* 长度为 n,定义的寄存器访问 e_1 表示当前 *IRF* 的$[n_1,n_2]$区间(满足 $0\leqslant n_2\leqslant n_1<n$),现在需要访问 e_1 的第 n''位至第 n'位这一子区间,那么该规则需要检查要访问的子区间不能超过该字段的长度,即 $0\leqslant n''\leqslant n'<n_1-n_2$.若检查通过,最终访问的结果应为当前 *Cell* 中 *IRF* 的第 n_2+n''位至第 n_2+n'位,因此赋予一个类型表达式 $RegAcc(n,n_2+n'',n_2+n')$.类似的,规则(6)为规则(5)的特例,描述了对寄存器访问的某一位的访问.

在协议声明上下文里,表达式中域标识符的类型已经由规则(4)进行过初始化,可直接从环境中得到:

$$\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_A,\mathcal{P}\vdash fid:FieldAcc(n,n_1,n_2).$$

对指定协议的协议域访问只发生在 *Cell* 局部环境中,如图 3 中的域名表达式 *eth.etherType* 和 *vlan.etherType*,规则(7)对这样的访问进行检查.

$$\frac{\begin{array}{c}\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L}\vdash e:pid\ \mathcal{G},\mathcal{C},\mathcal{R}\vdash(pid,(Fields(flds),OptionField(ofld)),...))\\ flds=((fid_1:c_1),...,(fid_k:c_k))\ ofld=(ofid:0)\ \forall i:1\leqslant i\leqslant k.(\phi\vdash c_i:(Int,n_i))\\ n=n_1+n_2+...+n_k\ \exists i.fid=fid_i\ \mathcal{L}_C\text{ is }\mathcal{L}_A,\mathcal{L}_{B0}\text{ or }\mathcal{L}_{B1}\end{array}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash Efield(e,fid):FieldAcc(id,n,n_1+...+n_{i-1},n_1+...+n_i-1)} \tag{7}$$

抽象语法节点 *Efield*(*e*,*fid*)表示访问协议实例 *e* 的 *fid* 定长协议域.假设 *e* 为 *pid* 这个协议类型的实例名,*pid* 协议类型具有 k 个定长协议域 $fid_i(1\leqslant i\leqslant k)$和一个变长协议域 *ofid*,那么该规则需要检查访问的协议域名字 *fid* 应该为 k 个定长协议域名字 fid_i 中的一个.

若要访问协议域的某一段区间或某一位,可将协议域类比为寄存器访问区间,然后仿照规则(5)或规则(6)便可以得到相应的检查规则,这里就不再赘述.

3.4.3 指令中寄存器的访问

P3 通过一系列基本指令来对寄存器进行操作,在这些指令中对寄存器和协议域的访问也要满足相应的规则.在图 8 的 *instrucion* 非终结符中,可以看到,对于 *Set*,*Eq* 和 *Lg* 指令的目标寄存器对应的非终结符为 *tgt_reg_acc_name*,它表示单个寄存器访问标识符的寄存器空间访问(含区间嵌套).对应此类目标寄存器访问的规则包括规则(8)~规则(10).

$$\frac{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash id:RegAcc(n,n_1,n_2)\ \mathcal{L}_C\ is\ \mathcal{L}_A,\mathcal{L}_{B0}\text{ or }\mathcal{L}_{B1}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash TargetRegAccName(id):RegAcc(n,n_1,n_2)} \tag{8}$$

$$\frac{\begin{array}{c}\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash tran:RegAcc(n,m_1,m_2)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash e_1:(Int,k_1)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash e_2:(Int,k_2)\\ 0\leqslant k_2\leqslant k_1\leqslant m_1-m_2\ n_1=m_2+k_1\ n_2=m_2+k_2\ \mathcal{L}_C\text{ is }\mathcal{L}_A,\mathcal{L}_{B0}\text{ or }\mathcal{L}_{B1}\end{array}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash TargetRegAccName(tran,e_1,e_2):RegAcc(n,n_1,n_2)} \tag{9}$$

$$\frac{\begin{array}{c}\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash tran:RegAcc(n,m_1,m_2)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash e:(Int,k)\\ 0\leqslant k\leqslant m_1-m_2\ m=m_2+k\ \mathcal{L}_C\text{ is }\mathcal{L}_A,\mathcal{L}_{B0}\text{ or }\mathcal{L}_{B1}\end{array}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash TargetRegAccName(tran,e):RegAcc(n,m,m)} \tag{10}$$

规则(8)指通过寄存器访问标识符 *id* 对定义的寄存器区间进行整体访问;规则(9)指对 *tran* 表示的寄存器访问的第 e_2 至第 e_1 位子区间进行访问,此时需要检查所访问的子区间是合法的;规则(10)指对 *tran* 表示的寄存器访问的第 *e* 位进行访问,可视为规则(9)的一个特例.这 3 条是在 *cell* 局部环境($\mathcal{L}_C$)下的类型规则,这里,$\mathcal{L}_C$ 可以是 $\mathcal{L}_A$,$\mathcal{L}_{B0}$ 和 $\mathcal{L}_{B1}$.在 $\mathcal{P}$ 环境下的访问规则也和这 3 条类似,只需对类型环境作相应改变即可.

mov 指令中对寄存器的访问有所不同,它支持对多个定义的寄存器访问通过“++”运算符进行连接,然后再统一赋值,例如图 3 例子中的 *mov IRF_outer_vlan_high++IRF_outer_vlan_low*,*vlan.pcp++vlan.cfi++vlan.vid*.需要特别注意的是,这里连接的寄存器访问指的是 *IRF_outer_vlan_high* 和 *IRF_outer_vlan_low* 这两个标识符定义的寄存器区间,而 *vlan.pcp++vlan.cfi++vlan.vid* 是作为表达式来进行处理的.从图 8 中可以看到,*mov* 指令访问寄存器对应的非终结符为 *mov_reg_acc_name*,本质为通过递归方式定义的一串 *tgt_reg_acc_name*,即一连串寄存器访问的连接,对应的规则为规则(11)和规则(12).

$$\frac{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash tra:RegAcc(n,m_1,m_2)\ \mathcal{L}_C\text{ is }\mathcal{L}_A,\mathcal{L}_{B0}\text{ or }\mathcal{L}_{B1}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C\vdash MovRegAccName(tra):RegAcc(n,m_1,m_2)} \tag{11}$$

$$\frac{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C \vdash mra : RegAcc(n,m_1,m_2)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C \vdash tra : RegAcc(n,n_1,n_2)\quad m_2 = n_1 + 1\ \mathcal{L}_C \text{ is } \mathcal{L}_A,\mathcal{L}_{B0} \text{ or } \mathcal{L}_{B1}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C \vdash MovRegAccName(mra,tra) : RegAcc(n,m_1,n_2)} \tag{12}$$

这里要注意,当有多个寄存器访问通过“++”连接时,需要按照规则(12)中的方式来检查两个寄存器区间是否相邻,即 $m_2=n_1+1$.

3.4.4 位串连接运算

除了传统的位串连接以外,P3 还支持另外两种特殊的位串连接运算,用来连接两个相邻的寄存器访问或协议域,从而能够方便地对合并的位串进行整体的读取.

我们用“++”表示位串运算,遇不同操作数时可重载.在$\mathcal{L}_C$环境下,对寄存器访问和协议域的位串连接规则分别为规则(13)和规则(14).

$$\frac{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C \vdash e_1 : RegAcc(k,n_1,n_2)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C \vdash e_2 : RegAcc(k,m_1,m_2)\quad n_2 = m_1 + 1\ 0 \leqslant m_2 \leqslant m_1 < n_2 \leqslant n_1 < k\ \mathcal{L}_C \text{ is } \mathcal{L}_A,\mathcal{L}_{B0} \text{ or } \mathcal{L}_{B1}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C \vdash e_1 ++ e_2 : RegAcc(k,n_1,m_2)} \tag{13}$$

$$\frac{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C \vdash e_1 : FieldAcc(id,k,n_1,n_2)\ \mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C \vdash e_2 : FieldAcc(id,k,m_1,m_2)\quad m_1 = n_2 + 1\ 0 \leqslant n_1 \leqslant n_2 < m_1 \leqslant m_2 < k\ \mathcal{L}_C \text{ is } \mathcal{L}_A,\mathcal{L}_{B0} \text{ or } \mathcal{L}_{B1}}{\mathcal{G},\mathcal{C},\mathcal{R},\mathcal{L},\mathcal{L}_C \vdash e_1 ++ e_2 : FieldAcc(id,k,n_1,m_2)} \tag{14}$$

在对寄存器访问进行位串连接时,一定要保证连接的两个寄存器访问必须是相邻的,所以在规则(13)中,假设需要连接的两个寄存器访问依次为当前 *IRF* 的$[n_1,n_2]$和$[m_1,m_2]$两个区间,自然需要满足 $n_2=m_1+1$.

在对协议域进行位串连接时,同样要注意保证连接的两个协议域属于同一协议并且是相邻的.

4 P3 操作语义

前面介绍的类型系统刻画了 P3 语言的静态语义,可用于实现静态类型检查.能够通过静态类型检查的 P3 程序是类型良好的(well-typed).对于类型良好的 P3 程序,可遵循其动态语义执行.本节基于图 8 的 P3 抽象语法,为 P3 定义一种操作语义风格的形式化动态语义.

4.1 语义环境

P3 程序可以访问两个存储区:一个用来存放数据包包头信息,称为 *PKT* 寄存器;另一个是通用寄存器 *IRF*.对于 P3 程序所描述的包解析器处理过程来说,可以将 *PKT* 看作是只读寄存器,而 *IRF* 是可以读和写的寄存器.在 P3 语义模型中,对于 *IRF* 寄存器的访问,我们采用的存储模型表示为 *regacc*(*n*,*i*,*j*,*bv*),其中,

- *n* 为当前语义环境下可访问的 *IRF* 寄存器大小(存储 *bit* 数);
- *i* 和 *j*(满足 $0 \leqslant j \leqslant i \leqslant n$)描述一个当前 *IRF* 寄存器的访问区间(*i*,*j*);
- *bv* 代表一个长度为 $\hat{i}j+1$ 的二进制位串值.

对于 *PKT* 寄存器的访问,出于语义定义的需要,不同于访问 *IRF*,我们采用的存储模型可表示为

fdacc(*id*,*n*,*i*,*j*,*bv*),

其中,*id* 标识某类协议数据包头的一个实例,*n* 为该实例被分配的 *PKT* 寄存器大小(存储 *bit* 数),而 *i*,*j*,*bv* 的含义与 *regacc*(*n*,*i*,*j*,*bv*)中是类似的.

基于上述存储模型,我们设计了定义 P3 语言语义所需要用到的语义环境,如图 9 所示.

P3 语言有多个层次的语义环境,其中,全局环境表示为 *ge*,*layer* 局部环境表示为 *le*,*cell* 局部环境用 *ce* 表示,而 *protocol* 局部环境为ξ_ρ.

环境 *ge*,*le* 和 *ce* 分别由各自的子环境构成,即 $ge=(\gamma,\sigma,\delta)$,$le=(\xi_t,nh,len,bp)$以及 $ce=(\delta_A,\delta_{B0},\delta_{B1})$.

图 9 给出了所涉及到的全部语义环境的定义及其基本含义.环境之间的嵌套层次关系从外到内依次为 *Global*,*Layer*,*Cell* 和 *Protocol*,即 *ge*,*le*,*ce* 和ξ_ρ依次嵌套.

Global	ge	::=	(γ,σ,δ)	全局环境分为 3 部分
	γ	::=	(lr,cr,ps,ls,ι,ρ)	P3 规范全局定义中的几要素
	σ	::=	$id\rightarrow val$	全局常量环境
	δ	::=	$raid\rightarrow regacc(n,i,j,bv)$	全局 *IRF* 寄存器(大小为 n,目前为 3 倍 *cell* 寄存器长度)访问环境,访问区间(i,j),内容为 bv
	lr	::=	$lreglen(k)$	设置全局 *IRF* 寄存器大小
	cr	::=	$creglen(k)$	设置 *Cell* 局部 *IRF* 寄存器大小
	ps	::=	$pset(id,\ldots,id)$	Protocol 标识符集合
	ls	::=	$lset(id,\ldots,id)$	Layer 标识符集合(有序)
	ι	::=	$lid\rightarrow ldef$	Layer 定义
	ρ	::=	$pid\rightarrow pdef$	Protocol 定义
Layer	le	::=	(ξ_ι,nh,len,bp)	Layer 局部环境分为 4 部分
	ξ_ι	::=	$id\rightarrow(len,(fid\rightarrow(n,bv)))$	Protoco 实例环境
	nh	::=	$nextheader(pid)$	当前 nextheader 标识符
	len	::=	$length(k)$	当前 *lenth* 大小
	bp	::=	$bypass(k)$	当前 *bypass* 值
Cell	ce	::=	$(\delta_A,\delta_{B0},\delta_{B1})$	
	δ_A	::=	$raid\rightarrow regacc(n,i,j,bv)$	*Cell A* 局部 *IRF* 寄存器(大小为 n,一个 *cell* 的长度)访问环境,访问区间(i,j),内容为 bv
	δ_{B0}	::=	$raid\rightarrow regacc(n,i,j,bv)$	*CellB*0 局部 *IRF* 寄存器(大小为 n,一个 *cell* 的长度)访问环境,访问区间(i,j),内容为 bv
	δ_{B1}	::=	$raid\rightarrow regacc(n,i,j,bv)$	*CellB*1 局部 *IRF* 寄存器(大小为 n,一个 *cell* 的长度)访问环境,访问区间(i,j),内容为 bv
Protocol	ξ_ρ	::=	$fid\rightarrow fdacc(id,n,i,j,bv)$	Protocol 局部环境
Identifier	*raid*,*lid*,*pid*, *fid*	::=	*id*	各种类别的标识符

Fig.9 Semantic environment of the P3 language

图 9 P3 语言语义环境

4.2 语义规则

下面给出所定义的 P3 语言操作语义的部分有代表性的语义规则.完整的语义规则定义可参考 P3 设计文档[17].除了常规断言,语义规则中最主要的断言形式包括如下两种.

- $\Gamma\vdash e\Rightarrow v$;
- $\Gamma\vdash(\pi,s)\Rightarrow\pi'$.

其中,Γ为某些环境的集合,e 为表达式(含各类标识符),s 为各种定义或语句,π和π'为环境.$\Gamma\vdash e\Rightarrow v$ 表示在环境(集)Γ下,e 的计算结果为 v.这里,v 可以是整数和位串等常规类型取值,也可以是第 4.1 节中所述的两类寄存器存储访问取值.$\Gamma\vdash(\pi,s)\Rightarrow\pi'$表示在环境环境(集)$\Gamma$下,由实施定义或执行语句 s 后,环境(集)π将改变为π'.

4.2.1 寄存器访问环境的初始化

规则(15)用来初始化 *cell A* 中定义的寄存器访问.

$$\frac{\begin{array}{c}ge=(\gamma,\sigma,\delta)\ ge,le,\delta_A\vdash e_1\Rightarrow n_1\ ge,le,\delta_A\vdash e_2\Rightarrow n_2\ \gamma=(lr,creglen(n),ps,ls,\iota,\rho)\\ 0\leqslant n_2\leqslant n_1<n\ id\notin dom(\delta_A)\ \forall id'\in dom(\delta_A).(\delta_A\vdash id'\Rightarrow regacc(n,i,j,bv)\rightarrow i<n_2\vee n_1<j)\\ \delta'_A=\delta_A\cup\{id:regacc(n,n_1,n_2,null)\}\end{array}}{ge,le\vdash(\delta_A,IRF(id,e_1,e_2))\Rightarrow\delta'_A}\tag{15}$$

这些定义形如 id=*IRF*(e_1,e_2),参见图 3 中 L2 层的 *ARegisters* 定义部分.当前环境下,寄存器 *IRF* 的大小为 n,由全局环境γ中的 *creglen*(n)保存.e_1 和 e_2 在当前环境下的取值 n_1 和 n_2 决定了变量 id 的寄存器访问区间,初始时,将寄存器该区间的位串值置为 *null*.如同前一节中类型规则(1)的限定,*IRF*(n_1,n_2)中的 n_1 和 n_2 满足 $0\leqslant n_2\leqslant n_1<n$.

对于 *cell B*0 和 *cell B*1 中寄存器访问的初始化也类似于规则(15).硬件实现时可以保证:不同层的局部寄存器访问使用不同的 *IRF* 空间;同一层中不同 *cell* 的局部寄存器访问也分别配备有互不冲突的私有 *IRF* 空间.后面规则中的类似情况,将不再重复解释.

规则(16)也是用来初始化 *cell A* 中定义的寄存器访问.

$$\frac{ge=(\gamma,\sigma,\delta)\ ge,le,\delta_A\vdash e\Rightarrow k\ \gamma=(lr,creglen(n),ps,ls,\iota,\rho)\ 0\leqslant k<n\quad id\notin dom(\delta_A)\ \forall id'\in dom(\delta_A).(\delta_A\vdash id'\Rightarrow regacc(n,i,j,bv)\rightarrow i<k\vee k<j)\quad \delta'_A=\delta_A\cup\{id:regacc(n,k,k,null)\}}{ge,le\vdash(\delta_A,IRF(id,e))\Rightarrow\delta'_A} \tag{16}$$

这些定义形如 *id=IRF*(*e*),是访问寄存器 *IRF* 的某一 *bit* 的空间.

在 P3 语言中,允许当前层中的 *cell* 单元访问前一层的寄存器空间,为此,我们使用了特定的全局环境δ.规则(17)用来描述在层切换的时刻环境δ的初始化.

$$\frac{\begin{array}{c}ge=(\gamma,\sigma,\delta)\ \gamma=(lreglen(n),creglen(k),ps,ls,\iota,\rho)\ n=3*k\\ le=(\xi_\iota,nextheader(pid),length(i),bypass(j))\ ce=(\delta_A,\delta_{B0},\delta_{B1})\\ \delta'=\{(id:regacc(n,2*k+n_1,2*k+n_2,bva))\mid(id:regacc(k,n_1,n_2,bva))\in\delta_A\}\cup\\ \{(id:regacc(n,k+n_1,k+n_2,bvb0))\mid(id:regacc(k,n_1,n_2,bvb0))\in\delta_{B0}\}\cup\\ \{(id:regacc(n,n_1,n_2,bvb1))\mid(id:regacc(k,n_1,n_2,bvb1))\in\delta_{B1}\}\\ ge'=(\gamma,\sigma,\delta')\ le'=(\phi,nextheader(null),length(null),bypass(null))\ ce'=(\phi,\phi,\phi)\end{array}}{\vdash(ge,le,ce,"layer-switch")\Rightarrow(ge',le',ce')} \tag{17}$$

环境δ中,可访问的寄存器 *IRF* 空间大小 *n* 记录在全局环境γ的 *lreglen*(*n*)分量中.注意,另一分量 *creglen*(*k*)表示当前 *cell* 的局部 *IRF* 寄存器空间大小.由于每一层包含 3 个 *cell* 单元,因此 *n*=3*k*.另外值得注意的是,在第 1 层开始处理的时刻,环境δ由全局寄存器访问定义来初始化,相应的规则类似于规则(15)和规则(16),只是寄存器 *IRF* 空间的初始化由硬件来完成(不用置为 *null*),限于篇幅,不在此列出.

由于在 *cell* 的局部寄存器访问环境(如δ_A)下也可以访问全局寄存器访问环境δ,因此按照先本地后全局来处理可能的同名寄存器访问标识符的情形.

4.2.2 表达式中寄存器的访问

表达式中对寄存器的访问,可以直接使用当前环境中已有定义的寄存器访问 *id*,比如环境δ_A下 *id* 的寄存器访问的语义值为 *regacc*(*id*,*n*,n_1,n_2,*bv*),可由下式得到.

$$ge,le,\delta_A\vdash id\Rightarrow regacc(n,n_1,n_2,bv).$$

注意,这里的 *id* 可能未在局部寄存器环境δ_A中有定义,而是可能定义在 *ge* 包含的全局寄存器环境δ中.

规则(18)用于计算寄存器访问中某一位的语义值.

$$\frac{ge,le,\delta_C\vdash e_1\Rightarrow regacc(n,n_1,n_2,bv)\ ge,le,\delta_C\vdash e_2\Rightarrow n'\ 0\leqslant n_2\leqslant n_1<n\quad 0\leqslant n'\leqslant n_1-n_2\ b=get\_binary\_bit(bv,n')\ \delta_C\text{ is }\delta_A,\delta_{B0}\text{ or }\delta_{B1}}{ge,le,\delta_C\vdash ERegBit(e_1,e_2)\Rightarrow regacc(n,n_2+n',n_2+n',b)} \tag{18}$$

例如,对于图 3 中 L2 层 *cell A* 的寄存器访问定义 *IRF_tag_type_2b*=*IRF*[23:16],假设其当前语义值为 *regacc*(*n*,23,16,*bv*),则表达式 *IRF_tag_type_2b*[2]在δ_A环境(及其外层环境)下的语义值为 *regacc*(*n*,18,18,*b*).这里,*bv* 为当前 *IRF*[23,16]的位串值,*b* 为 *bv* 的第 2 位,即 *IRF* 寄存器的第 18 位 *IRF*[18]的 *bit* 值.该规则同样适用于 *cell B*0 和 *cell B*1,因此在规则中用δ_C代替.这里,δ_C为δ_A,δ_{B0}或δ_{B1}.这样,规则(18)实际上相当于 3 条类似的规则.对此,后面不再赘述.

规则(19)用于计算寄存器访问的子区间的语义值.

$$\frac{ge,le,\delta_C\vdash e_1\Rightarrow regacc(n,n_1,n_2,bv)\ ge,le,\delta_C\vdash e_2\Rightarrow n'\ ge,le,\delta_C\vdash e_3\Rightarrow n''\quad 0\leqslant n_2\leqslant n_1<n\ 0\leqslant n''\leqslant n'\leqslant n_1-n_2\ bv'=get\_binary\_bits(bv,n',n'')\ \delta_C\text{ is }\delta_A,\delta_{B0}\text{ or }\delta_{B1}}{ge,le,\delta_C\vdash ERegSection(e_1,e_2,e_3)\Rightarrow regacc(n,n_2+n',n_2+n'',bv')} \tag{19}$$

例如,对于图 3 中 L2 层 *cell A* 的寄存器访问定义 *IRF_tag_type_2b*=*IRF*[23:16],假设其当前语义值为 *regacc*(*n*,23,16,*bv*),则表达式 *IRF_tag_type_2b*[5,2]在δ_A环境(及其外层环境)下的语义值为 *regacc*(*n*,21,18,*bv*′).这里,*bv* 为当前 *IRF*[23,16]的位串值,*bv*′为 *bv* 的第 5 位~第 2 位之间区间的 *bit* 串值,即 *IRF* 寄存器的第 21 位~第 18 位区间 *IRF*[21,18]的 *bit* 串值.

寄存器访问支持任意(合法的)嵌套访问,如 *IRF_tag_type_2b*[5,2][4,2],*IRF_tag_type_2b*[5,2][0],*IRF_tag_type_2b*[5,2][4,2][2],等等.

4.2.3 表达式中协议域的访问

在 *Protocol* 局部环境ξ_ρ下,表达式中可通过域名标识符对协议域进行直接访问.比如,域名为 *fid* 的域的语义值为 *fdacc*(*id*,*n*,n_1,n_2,*bv*),可由下式得到.

$$ge,le,\delta_A,\xi_\rho \vdash fid \Rightarrow fdacc(id,n,n_1,n_2,bv).$$

这里,*id* 为相应协议的一个实例标识符.注意:我们省略了环境ξ_ρ的初始化规则,在初始化ξ_ρ时,首先会置 *bv* 为 *null*,而在上式中的 *bv* 会在δ_A环境中由相应实例(*id*)的初始取值来获取.后者由硬件设置,参考下面的规则(20)及其相应的解释.

$$\frac{ge,le,\delta_C \vdash id \Rightarrow (pid,pins)\ pins=((fid_1,(n_1,bv_1)),(fid_2,(n_2,bv_2)),...,(fid_k,(n_k,bv_k)))\quad n=n_1+n_2+...+n_k\ \exists i.fid=fid_i\ \delta_C \text{ is } \delta_A,\delta_{B0} \text{ or } \delta_{B1}}{ge,le,\delta_C \vdash Efield(id,fid) \Rightarrow fdacc(id,n,n_1+n_2+...+n_{i-1},n_1+n_2+...+n_i-1,bv_i)} \tag{20}$$

域名标识符只能在环境ξ_ρ下直接访问,若是在 *le* 环境的某个 *cell* 子环境δ_C下,需要通过域名表达式来访问一个 *protocol* 实例的域,见规则(20).域名表达式形如图 3 中的 *eth.etherType* 和 *vlan.etherType*.

在规则(20)中,会根据域 *fid* 在包头中的位置和大小来计算语义值.如图 3 中 *eth.etherType* 的语义值形如 *fdacc*(*eth*,112,96,111,*bv*),其中,*bv* 为当前 *PKT* 寄存器中 *eth* 区域中对应于域 *fid* 的 *bit* 空间取值,由硬件进行初始化设置,在 *cell* 单元的处理中仅能只读访问.

规则(21)用于计算协议域访问中某一位的语义值,与规则(18)相似.

$$\frac{ge,le,\delta_A \vdash e_1 \Rightarrow fdacc(id,n,n_1,n_2,bv)\ ge,le,\delta_c \vdash e_2 \Rightarrow n'\ 0 \leqslant n_1 \leqslant n_2 < n\quad 0 \leqslant n' \leqslant n_2-n_1\ b=get\_binary\_bit(bv,n')\ \delta_C \text{ is } \delta_A,\delta_{B0} \text{ or } \delta_{B1}}{ge,le,\delta_C \vdash EFieldBit(e_1,e_2) \Rightarrow fdacc(id,n,n_1+n',n_1+n',b)} \tag{21}$$

例如,在图 3 的 *cell B*0 处理单元中,计算表达式 *eth.dmac*[40]得到的语义值为 *fdacc*(*eth*,112,39,39,*b*),其中,*b* 是硬件初始化的 *bit* 值.

规则(22)用于计算协议域访问的子区间的语义值,与规则(19)相似.

$$\frac{ge,le,\delta_C \vdash e_1 \Rightarrow fdacc(id,n,n_1,n_2,bv)\ ge,le,\delta_c \vdash e_2 \Rightarrow n'\ ge,le,\delta_c \vdash e_3 \Rightarrow n''\quad 0 \leqslant n_1 \leqslant n_2 < n\ 0 \leqslant n'' \leqslant n' \leqslant n_2-n_1\ bv'=get\_binary\_bits(bv,n',n'')\ \delta_C \text{ is } \delta_A,\delta_{B0} \text{ or } \delta_{B1}}{ge,le,\delta_C \vdash EFieldSection(e_1,e_2,e_3) \Rightarrow fdacc(id,n,n_1+n',n_1+n'',bv')} \tag{22}$$

例如,ethernet 实例 *eth* 的访问表达式 *eth.smac*[16,23]进行计算得到语义值为 *fdacc*(*eth*,112,63,70,*bv*′),其中,*bv*′是已被硬件初始化过的 *bit* 串 *bv*(对应于实例 eth 的数据包包头)的子串.

类似于寄存器的访问,协议域的访问也支持任意(合法的)嵌套,如 *eth.smac*[16,23][2,4],*eth.smac*[16,23][0],*eth.smac*[16,23][2,4][2],等等.这里应注意:我们规定寄存器访问的上下界由大到小,而协议域访问区间或子区间的上下界由小到大.

4.2.4 位串连接运算

位串连接(++)是二元运算.规则(23)描述两个常规 *bit* 串之间的连接语义,类似于传统语言中的字符串连接(用函数 *cat* 计算).

$$\frac{ge,le,\delta_C \vdash e_1 \Rightarrow bs_1\ ge,le,\delta_C \vdash e_2 \Rightarrow bs_2\ bs=cat(bs_1,bs_2)\ \delta_C \text{ is } \delta_A,\delta_{B0} \text{ or } \delta_{B1}}{ge,le,\delta_C \vdash (e_1++e_2) \Rightarrow bs} \tag{23}$$

位串连接运算在 *Protocol* 环境中也可以进行,为节省篇幅,我们省略相应的规则(其他地方也有类似的情况,不再赘述).

规则(24)描述两个寄存器访问的位串连接语义.

$$\frac{ge,le,\delta_C \vdash e_1 \Rightarrow regacc(k,n_1,n_2,bs_1)\ |bs_1|=n_1-n_2+1\ ge,le,\delta_C \vdash e_2 \Rightarrow regacc(k,m_1,m_2,bs_2)\ |bs_2|=m_1-m_2+1\ n_2=m_1+1\ 0\leqslant m_2\leqslant m_1<n_2\leqslant n_1<k\ bs=cat(bs_1,bs_2)\ \delta_C \text{ is } \delta_A,\delta_{B0} \text{ or } \delta_{B1}}{ge,le,\delta_C \vdash (e_1++e_2) \Rightarrow regacc(k,n_1,m_2,bs)} \tag{24}$$

两个 *bit* 串(bs_1 和 bs_2)之间的连接也是传统的字符串连接(用函数 *cat* 计算),但要注意的是,两个寄存器访问区间一定要连续(规则中的条件 $n_1=m_2+1$).

规则(25)描述两个协议域访问的位串连接语义.

$$\frac{ge,le,\delta_C \vdash e_1 \Rightarrow fdacc(id,k,n_1,n_2,bs_1)\ |bs_1|=n_2-n_1+1\ ge,le,\delta_C \vdash e_2 \Rightarrow fdacc(id,k,m_1,m_2,bs_2)\ |bs_2|=m_2-m_1+1\ n_1=m_2+1\ 0\leqslant m_1\leqslant m_2<n_1\leqslant n_2<k\ bs=cat(bs_1,bs_2)\ \delta_C \text{ is } \delta_A,\delta_{B0} \text{ or } \delta_{B1}}{ge,le,\delta_C \vdash (e_1++e_2) \Rightarrow fdacc(id,k,m_1,n_2,bs)} \tag{25}$$

同样也要注意的是,两个域访问是属于同一个实例,并且域区间一定要连续.

4.2.5 指令中寄存器的访问

规则(26)描述 *set* 指令的语义:将表达式 *e* 的取值转化为 *bit* 串后,替换当前环境下寄存器访问 *ra* 语义值中对应区间的 *bit* 串($\delta'_C=\delta_C|_{ra\Rightarrow regacc(k,i,j,bv')}$).

$$\frac{ge,le,\delta_C \vdash e \Rightarrow v\ ge,le,\delta_C \vdash ra \Rightarrow regacc(k,i,j,bv)\ bv'=trans\_to\_bits(bv,n)\ n=i-j+1\ \delta'_C=\delta_C|_{ra\Rightarrow regacc(k,i,j,bv')}\ \ \delta_C \text{ is } \delta_A,\delta_{B0} \text{ or } \delta_{B1}}{ge,le \vdash (\delta_C,Set(ra,e)) \Rightarrow \delta'_C} \tag{26}$$

Set 指令中的表达式 *e* 比较简单,其计算结果为普通类型的语义值,相应的 *trans_to_bits* 函数较易实现.

规则(27)描述 mov 指令的语义:将表达式 *e* 的取值转化为 *bit* 串后,替换当前环境下寄存器访问 *ra* 语义值中对应区间的 *bit* 串.

$$\frac{\begin{array}{c}ge,le,\delta_C \vdash e \Rightarrow v\ mra=(ra_1++ra_2++...++ra_m)\ ge,le,\delta_C \vdash ra_1 \Rightarrow regacc(k,i_1,j_1,bv_1)\\ ge,le,\delta_C \vdash ra_2 \Rightarrow regacc(k,i_2,j_2,bv_2)...\ ge,le,\delta_C \vdash ra_m \Rightarrow regacc(k,i_m,j_m,bv_m)\\ j_1=i_2+1\ j_2=i_3+1...\ j_{m-1}=i_m+1\ bv'=trans\_to\_bits(bv,n)\ n=i_1-j_m+1\\ bv'_1=bv'[i_1,j_1]\ bv'_2=bv'[i_2,j_2]...\ bv'_m=bv'[i_m,j_m]\\ \delta'_C=\delta_C|_{ra_1\Rightarrow regacc(k,i_1,j_1,bv'_1),ra_2\Rightarrow regacc(k,i_2,j_2,bv'_2),...,ra_m\Rightarrow regacc(k,i_m,j_m,bv'_m)}\ \ \delta_C \text{ is } \delta_A,\delta_{B0} \text{ or } \delta_{B1}\end{array}}{ge,le \vdash (\delta_C,Mov(mra,e)) \Rightarrow \delta'_C} \tag{27}$$

虽然非形式解释类似于 *set* 指令,但 *mov* 指令的寄存器访问要复杂许多:首先,寄存器访问 *mra* 可以是其他多个空间连续寄存器访问的连接,因此在修改环境(规则中是δ_C)时,要考虑对每一个寄存器访问变量的语义值进行替换;另外,*mov* 指令中的表达式 *e* 也比较复杂,可以包含寄存器和协议域的访问以及它们的连接运算,因此,相应的 *trans_to_bits* 函数实现较复杂一些.

规则(28)描述 *eq* 指令的语义:对寄存器访问表达式 e_1 和 e_2 的取值进行比较(等价于将它们分别转化为整数后在比较),将比较结果(true/false)转换为 *bit* 串后,替换当前环境下寄存器访问 *ra* 语义值中对应区间的 *bit* 串($\delta'_C=\delta_C|_{ra\Rightarrow regacc(k,i,j,bv')}$).

$$\frac{ge,le,\delta_C \vdash e_1 \Rightarrow v_1\ ge,le,\delta_C \vdash e_2 \Rightarrow v_2\ ge,le,\delta_C \vdash ra \Rightarrow regacc(k,i,j,bv)\ b=trans\_to\_int(v_1)==trans\_to\_int(v_2)\ bv'=trans\_to\_bits(b,n)\ n=i-j+1\ \delta'_C=\delta_C|_{ra\Rightarrow regacc(k,i,j,bv')}\ \ \delta_C \text{ is } \delta_A,\delta_{B0} \text{ or } \delta_{B1}}{ge,le \vdash (\delta_C,Eq(ra,e_1,e_2)) \Rightarrow \delta'_C} \tag{28}$$

对应于指令 *lg* 的语义规则与规则(28)相似.

5 P3 语言编译器 P3C 的设计框架

5.1 编译结构

P3 语言的编译器 P3C 正在实现中,其编译结构如图 10 所示.源是人工编写的 P3 语言源代码(P3 source),目

标是编译器生成的二进制硬件配置文件(configure file).整个编译过程分为 4 个步骤:源代码首先经过词法语法分析器生成一棵 P3 抽象语法树(P3 AST);接着,对该抽象语法树进行类型检查(type checking),以保证该抽象语法树是类型良好的,从而得到一棵类型良好的抽象语法树(Well-typed P3 AST);下一步,根据类型良好的抽象语法树生成一种称为 P3 汇编语言(P3 assembly,简称 P3 ASM)的中间语言代码,作为 P3 源代码和目标配置文件之间的“桥梁”;最后,由 P3 汇编语言代码翻译到目标配置文件.

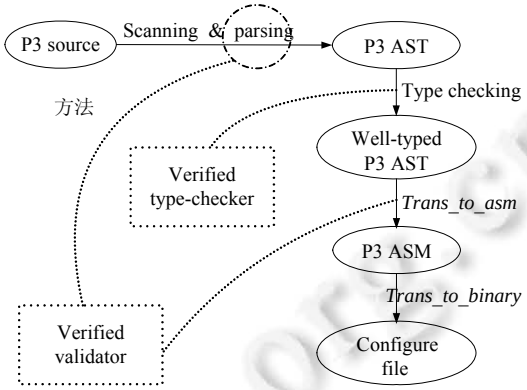

Fig.10 Compiler architecture of P3C

图 10 P3C 编译结构

在整个编译结构中,我们十分注重保证编译过程的可信性.目前,我们对语法分析、类型检查和翻译到 P3 汇编语言这 3 个过程做了可信性设计,从而让 P3C 编译器更加安全可靠,见第 5.2 节.

5.1.1 词法语法分析

词法语法分析是编译过程的第 1 步工作,源代码首先要通过词法分析器的处理,转换为单词序列,然后输入到语法分析器中.语法分析器则根据预先定义好的语法,接受单词序列输入生成抽象语法树为后续步骤提供输入和处理对象.这里主要的设计工作分为两个内容:定义 P3 语言的语法、定义抽象语法树节点.

词法和语法分析实现的基本原理与传统编译器相同,只是在具体实现时采用了 Jourdan 等人提出的方法[18]对语法分析程序进行了形式化验证,见第 5.2.1 节.

5.1.2 静态语义检查

通过以上的词法语法分析,语法分析器构造出了 P3 抽象语法树.为了保证这样的一棵抽象语法树是类型安全的,下一步要对它进行类型检查,若它能够通过类型检查,则为类型良好的 P3 抽象语法树.类型系统的设计工作已在第 3 节中给出.

5.1.3 生成汇编

由于 P3 源语言与需要生成的目标二进制配置文件跨度较大,难以一次完成翻译;同时,软硬件开发人员也需要协同设计.为此,我们定义了一种中间格式的 P3 汇编语言(P3 assembly,简称 P3 asm)作为“桥梁”,用来实现从 P3 源语言到目标二进制配置文件的过渡.P3 asm 更加关注硬件配置的细节,很关键的一点就是,它把 P3 源语言中对层解析过程中的匹配查找条件(if 语句)和寄存器操作(action)的描述转换成了对 *PB*,*PC* 硬件模块的描述.图 11 是针对图 3 中描述 L2 层解析的 P3 规范片段翻译至汇编代码片段的示例.

图 11 中,L2 层对应汇编代码的头两行中,*Pins*(*eth*,112)表明协议数据包头 *eth*(ethernet 的实例)的总 *bit* 数为 122,可由 ethernet 各个 *fields* 的总字节数(14,参见第 2.3 节)得到,即 8×14.类似地,*vlan* 是 ieee802-1qTag(参见第 2.3 节)的实例,总 *bit* 数应为 8×4,因此汇编片段中包含 *Pins*(*vlan*,32).

接下来就是对应于解析器硬件动作的几张配置表.每一层共有 7 张配置表:*cell A* 有 3 张,*cell B*0 和 *cell B*1 各 2 张.图 11 刻画了图 3 所描述的 L2 层示例所对应的 3 张 *cell A* 和 2 张 *cell B*0 硬件配置表.*cell B*1 的两张表和 *cell B*0 的配置表结构上是一致的.我们称 *cell A* 的 3 张表分别为 *cella_pb*,*cella_pc_cur* 和 *cella_pc_nxt*;称 *cell*

*B*0 的 2 张表分别为 *cellb*0_*pb* 和 *cellb*0_*pc_cur*,以及 *cell B*1 的 2 张表分别为 *cellb*1_*pb* 和 *cellb*1_*pc_cur*.

在图 11 中,*Abegin* 和 *Aend* 之间的部分用来描述 *cella_pb* 表,*ACbegin* 和 *ACend* 之间的部分用来描述 *cella_pc_cur* 表,以及 *ANbegin* 和 *ANend* 之间的部分用来描述 *cella_pc_nxt* 表;*B*0*begin* 和 *B*0*end* 之间的部分用来描述 *cellb*0_*pb* 表,而 *B*0*Cbegin* 和 *B*0*Cend* 之间的部分用来描述 *cellb*0_*pc_cur* 表.

```
l2:
Pins(eth,112)                                                    //size:8×14
Pins(vlan,32)                                                    //size:8×4
Abegin                                                           //cella_pb
   0x1,{(eth,96,16)==0x8100,(vlan,16,16)==0x0800},0x1,0x3,1      //eth+vlan+ipv4
   0x1,{(eth,96,16)==0x0800},0x3,0x3,1                           //eth+ipv4
Aend
Acbegin                                                          //cella_pc_cur
   0x1,{(mov(IRF,0,16),(vlan,0,16)),(set(IRF,16,8),1),(set(IRF,24,8),0)},0x12
   0x3,{(set(IRF,16,8),0),(set(IRF,24,8),0)},0xe
ACend
Anbegin                                                          //cella_pc_nxt
   0x3,{(·)+(0 9)},{(·)+(0xc 0xd 0xe 0xf 0x10 0x11 0x12 0x13)},{(·)+(6 7)}   //ipv4
ANend
B0begin                                                          //cellb0_pb
   0x1,{(eth,0,48)==0xFFFFFFFFFFFF},0x1
   0x1,{(eth,40,1)==1},0x2
B0end
B0Cbegin                                                         //cellb0_pc_cur
   0x1,{(set(IRF,0,8),3)}                                        //sub_id:01,set IRF_l2_type=3;
   0x2,{(set(IRF,0,8),2)}                                        //sub_id:02,set IRF_l2_type=2;
B0Cend
```

Fig.11 A segment of P3 assembly code

图 11 P3 汇编代码片段

cella_pb 表中每一项分 5 个部分(以图 11 中 *Abegin* 部分的第 1 项为例):第 1 部分 0*x*1 是 layer 编号,代表 l2 是所声明的第 1 个 layer;第 2 部分({(*eth*,96,16)==0*x*8100,(*vlan*,16,16)==0*x*0800})代表一个逻辑公式(略复杂,下一段单独解释),刻画当前项对应的分支条件;第 3 部分 0*x*1 指向 *cella_pc_cur* 中编号为 0*x*1 的动作集合;第 4 部分对应当前分支对应的 *next_header* 属性,如 0*x*3 代表 ipv4(声明 *protocol* 时,ipv4 处于第 3 位次);第 5 部分对应本层的 *bypass*,可能的取值有 0,1,2.

关于 *cella_pb* 表中每一项的第 2 部分,对应一个分支的条件,即用于查表的条件.比如,{(*eth*,96,16)==0*x*8100,(*vlan*,16,16)==0*x*0800}对应图 3 的 *cellA* 中所描述的分支条件(*eth.etherType*==0*x*8100) && (*vlan.etherType*==0*x*0800).观察一下 *etherType* 在 *eth*(ethernet 的实例)中的偏移位置和位数,可知其偏移量为 96(0~95 是 *dmac* 和 *smac* 占据),位数为 16.类似地,*vlan*(ieee802-1qTag 的实例)中 *etherType* 的偏移量为 16,位数为 16.类似地,*cella_pb* 表第 2 行中的{(*eth*,96,16)==0*x*0800}对应图 3 的 *cellA* 中的分支条件(*eth.etherType*==0*x*0800).各行对应的分支条件之间是互斥的.每个 *cell* 中,各 if 语句以及每条 if 语句内部的所有分支条件是互斥的(在 P3 的设计中,可由静态类型检查来判定).多条平行的 if 语句可组合产生的分支条件全部是互斥的,体现在汇编代码的 *cella_pb*, *cellb*0_*pb* 和 *cellb*1_*pb* 表中也都是互斥的,因此硬件实现查表动作不会发生条件冲突的情形.

表 *cella_pc_cur* 中每一项分 3 个部分(以图 7 中 *ACbegin* 部分的第 1 项为例):第 1 部分对应一个索引号 0*x*1,编号规则不重要,只要表中各项的索引号互不相同即可;第 3 部分对应本层的 *length* 属性值(单位:字节),比如,这里的 0*x*12(对应十进制数 18)由图 3 中的 *length* 语句 *length=eth.length+vlan.length* 得到(*eth* 字节数 14,加上 *vlan* 字节数 4);第 2 部分为所有指令的集合(这些指令在硬件中会并行执行),比如,{(*mov*(*IRF*,0,16),(*vlan*,0,16)),(*set*(*IRF*,16,8),1),(*set*(*IRF*,24,8),0)}对应图 3 中 3 条指令的集合.其中,(*mov*(*IRF*,0,16),(*vlan*,0,16))对应指令"*mov IRF_outer_vlan_high++IRF_outer_vlan_low,vlan.pcp++vlan.cfi++vlan.vid*".试观察,在 *ARegisters* 中有定义:*IRF_outer_vlan_high=IRF*[7:0],*IRF_outer_vlan_low=IRF*[15:8].注意:它们是连续的两个字节,对应 *IRF* 的 0~15 位,共 16 位,所以我们在 asm 中用(*IRF*,0,16)表示.而 *vlan.pcp++vlan.cfi++vlan.vid* 表示 *vlan*(ieee802-1qTag 的实例)中从

开始连续的域 *pcp*,*cfi* 和 *vid* 的连接,长度也是 16 位,对应的偏移量为 0,因此我们在 asm 中用(*vlan*,0,16)表示.另两条(*set*(*IRF*,16,8),1)和(*set*(*IRF*,24,8),0)分别对应两条 *set* 指令:"*set IRF_tag_type_2b*,1","*set IRF_pkt_type_3b*,0".从 *ARegisters* 中 *IRF_tag_type_2b* 和 *IRF_pkt_type_3b* 的定义容易看出,二者分别对应(*IRF*,16,8)和(*IRF*,24,8).

表 *cella_pc_nxt* 中每一项分 2 个部分(以图 7 中 *ANbegin* 部分的第 1 项也是唯一的一项为例):第 1 部分对应本层 *cella_pb* 中某些项的 *next_header* 所指向的协议编号,0*x*3 代表 ipv4(声明 *protocol* 时 ipv4 处于第 3 位次);第 2 部分略微麻烦,包含 3 个子部分,分别是对应于 *cellA*,*cellB*0 和 *cellB*1 这 3 个 *cell* 信息.每个子部分均含两个分量:第 1 个分量指明当前通用寄存器(这里指 *IRF*)中哪些内容需要传到后面层中某个 *cell* 的查表分支中使用,由于目前硬件设计方案尚未定型,所以暂时将其置为空集(空表);第 2 个分量对应于在后面层中某个 *cell* 的分支条件中需要用到该协议(ipv4)的所有字段所在的偏移位置(以字节为单位).图 11 表 *cella_pc_nxt* 中的{(·)+(0 9)}对应于 *cellA*,{(·)+(0*xc* 0*xd* 0*xe* 0*xf* 0*x*10 0*x*11 0*x*12 0*x*13}对应于 *cellB*0,{(·)+(6 7)}对应于 *cellB*1.它们的第 1 个部分对应于 *IRF* 位的偏移,均暂时置为空表(·).我们先来看 *cellB*0 和 *cellB*1 这两部分.从图 4 左边可知,后面的 L3 层中有对 ipv4 的处理,见实例 *v*4.在 L3 的 *cellB*0 中,*v*4 的 *srcAddr* 和 *dstAddr* 字段被用在分支条件中,而在 ipv4 协议包头(如图 4 右边所示)中,这两个字段的偏移位置处于第 13,14,...,20 字节,若从 0 开始编号,则对应的字节偏移位置分别为 12,13,14,15,16,17,18 和 19,换作 hexadecimal 数字可用表(0*xc* 0*xd* 0*xe* 0*xf* 0*x*10 0*x*11 0*x*12 0*x*13)来描述.同样,在 L3 的 *cellB*1 中,*v*4 的 *flagOffset* 和 *flags* 字段被用在分支条件中,而在 ipv4 协议包头中,这两个字段的偏移位置处于第 7 和 8 字节,由于从 0 开始编号,因此用表/集合(6 7)表示.理解了这两个之后,再来看 *cellA* 对应的部分.虽然在 L3 的 *cellA* 描述中没有直接的分支条件,但在 ipv4 协议中包含了分支,本文的设计中默认在协议中的语句都将划归相应的 *cellA*,同时在这种情形(协议中包含了分支条件)下当前层中仅允许包含最多一个 ipv4 的实例,这就不难理解(0 9)这个表/集合了.在 ipv4 协议中包含的分支条件中用到字段 *ihl* 和 *theProtocol*,可对应到协议包头中的第 1 和第 10 个字节,以 0 作为初始编号的话,就对应了(0 9).

表 *cellb*0*_pb* 的每一项包含 3 个部分,其含义与表 *cella_pb* 的前 3 部分相同,只是内容对应于 *cell B*0 而非 *cell A* 的定义.类似地,表 *cellb*0*_pc_cur* 的每一项只有 2 个部分,分别与表 *cella_pc_cur* 每一项的前两部分对应.

如果表 *cellb*1*_pb* 和表 *cellb*1*_pc_cur* 有定义的话,其结构与 *cellb*0*_pb* 和 *cellb*0*_pc_cur* 完全对应.

5.1.4 生成包解析器硬件配置文件

P3C 编译器的最后一步,是由 P3 汇编代码生成二进制硬件配置文件.具体来说,就是对每一层生成与上一节中提到的 7 张表所对应的二进制配置表.如 *cell A* 的 3 张表(*cella_pb*,*cella_pc_cur* 和 *cella_pc_nxt*)中,第 1 张表对应 *PB* 模块的配置,第 2 和第 3 张对应 *PC* 模块的配置(如图 2 所示).根据图 11 中的 asm 片段,P3C 会将 L2 层的 *cella_pb* 表最终生成为如图 12 所示格式的二进制硬件配置表.

```
//12
//cella_pb(407bit*32)
//hdr_id(7)+mask(24*8b)+value(24*8b)+sub_id(7)+nxt_id(7)+bypass(2:mainbypass(1)+subbypass(1))
//    1 2 3 4 5 6 7  8 9 10 11 12 13 14 15 16 17 18 19 20 21 22          1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
01_00 00 ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00_00 00 81 00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00_01_03_1  //eth+vlan+ipv4
01_00 00 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00_00 00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00_03_03_1  //eth+ipv4
```

Fig.12 Information for the binary hardware configuration

图 12 二进制硬件配置信息

从这张配置表的标注信息可知,当前 L2 层 *cella_pb* 表的每一项包含 407bit 数据,假设共 32 项.由于该硬件配置表的语义涉及到硬件的实现细节,读者不必深入理解,若感兴趣,可参考 P3C 编译器的设计文档[17].

5.2 核心编译过程的可信设计

可编程的数据包解析器是可编程数据平面的第 1 环,为之设计并实现一个高可信的编译器,将对 SDN 的安全性有着十分重要的意义.同时,本文研究的背景是面向高安全等级的网络方面的课题,可信编译的需求更为突出.因此,P3C 编译器在设计之初就明确了定位:其主体将在交互式证明辅助工具 Coq 中实现,核心过程均实现基于定理证明的形式化验证.限于篇幅,下面仅结合图 10 简要介绍.

5.2.1 语法分析器的正确性

如图 10 所示,P3C 的设计考虑了语法分析过程的可信验证.语法分析的理论和技术已经相当成熟,并且现有各种语法分析器自动构造工具已被认为足够可信.然而,作为编译器的重要组成部分,语法分析器的形式化验证工作也是获得编译器正确性证据链的重要一环.已有成功的方法可以借鉴,是本文对 P3C 语法分析过程进行形式化验证的一个重要原因.

本文采用 Jourdan 等人[18]提出的方法来构建经过形式化验证的语法分析器,如图 13(源于文献[18])所示.这种方法即使用一个经过形式化验证的确认程序(validator)来对未经验证的语法分析器生成器(parser generator)所产生的 LR(1)自动机进行确认.在“编译-编译期”时,确认程序接受语法分析器生成器生成的 LR(1)自动机、原文法(grammer)以及用作证书的辅助信息(certificate)作为输入,然后验证 LR(1)自动机和原文法的等价性.若通过验证,就证明该语法分析器是正确的,从而得到了一个经过形式化验证的语法分析器.该方法实现较为简单,且适用于各类LR(1)自动机.Jourdan 等人运用该方法成功构造出了 CompCert 编译器[19]的语法分析器.他们对语法分析器进行的形式化验证,主要目的是证明语法分析器拥有某些特定的性质.这些性质分别是可靠性(soundness)、安全性(safety)、完备性(completeness)和无二义性(unambiguity).

限于篇幅,详情请参阅文献[18].

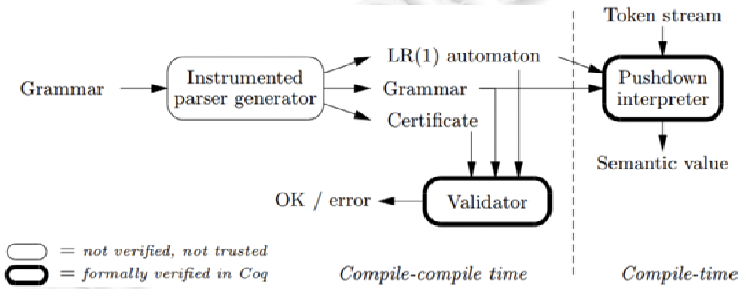

Fig.13 Process of the formal verification of the syntactic parser

图 13 语法分析器的形式化验证整体流程

语法分析过程是目前 P3C 中唯一完整实现并且经过形式化验证的部分,实现情况及其各模块的功能如图 14 所示.

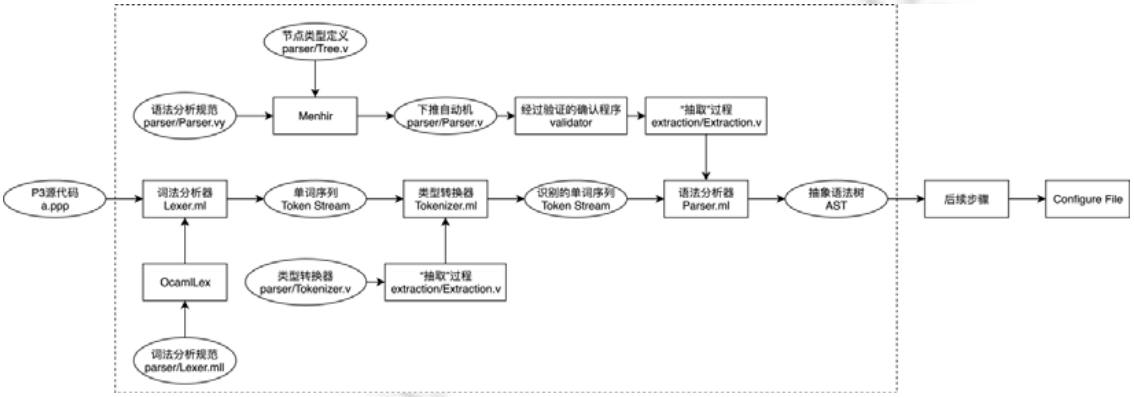

Fig.14 Parser component of P3C

图 14 P3C 语法分析部分

P3C 编译器的实现语言是 Ocaml.如同 Jourdan 等人在 CompCert 编译器中的实现,P3C 的语法分析程序是

由 Menhir[20](Ocaml 的一种常用的语法分析器生成器)自动生成的,词法分析程序是由面向 Ocaml 的词法分析器构造工具 Ocamllex 生成的.词法和语法分析相关代码中,除去来自 CompCert 的 validator(Coq 代码 3 138 行)以及 Coq 的库代码,包含 Coq 代码(Tree.v,Tokenizer.v,Extraction.v)522 行、Ocaml 代码(.ml)676 行、lexer 描述文件(.mll)183 行以及 parser 描述文件(.vy)617 行.

5.2.2 静态类型检查的正确性

静态类型检查过程对 P3 抽象语法树实施类型检查,其正确性可从两个方面描述:(1) 通过类型检查程序的 AST 一定是类型良好的;(2) 类型良好的 AST 一定能通过类型检查(即类型检查过程能正常结束).假设类型检查过程实现为函数 *typecheck_prog*;基于类型规则(见第 3 节)可定义 AST 形式的 P3 规范,*p* 是类型良好的,用 *wt_spec*(*p*)表示.这两个性质可以表示如下.

(1) $\forall p.typecheck\_prog(p)=OK(p)\rightarrow wt\_spec(p)$;

(2) $\forall p.wt\_spec(p)\rightarrow typecheck\_prog(p)=OK(p)$.

其中,$typecheck_prog(p)=OK(p)$表示函数 *typecheck_prog* 作用于 P3 规范 *p* 可正常终止,且不对 *p* 进行任何修改(返回 *p* 本身).

5.2.3 产生汇编的正确性

生成 P3 汇编语言是 P3C 编译器的核心工作,它将容易理解和使用的 P3 语言翻译成更加接近目标文件格式的一种中间语言.因此,保证翻译前后两种语言语义的一致性,将是保证编译过程正确性的关键.形式化验证是确保编译过程正确性的必要途径,主要有两种方法:(1) 构造并证明编译过程本身的正确性,类似于 CompCert 编译器[19]的构造方法;(2) 采用翻译确认[12]的思路,构造一个 validator 检查翻译前后的程序/规范在语义上的一致性,同时确保该确认程序的正确性(最好能够给出证明).P3C 编译器选择的是后一种方法(参考图 10),即构造从 P3 AST 到 P3 ASM 的一个确认程序,并证明其正确性.这一选择的出发点主要有:

1) 因需要软硬件协同设计,翻译程序需要尽早实现;而同时,其正确性证明的难度和工作量很大,时间上不能承受.即使能够按期实现,但协同设计的许多不确定性,意味着翻译过程的修改比较频繁.然而证明过程无法重用,导致整个工作成本大增.
2) 从我们的问题出发,设计从 P3 AST 到 P3 ASM 的确认程序的方案是相对容易的,而且其正确性证明比起直接验证翻译过程更加容易.
3) P3 AST 到 P3 ASM 的定义(语法、静态和动态语义)已经明确,比如本文前面几节的 P3 类型系统和操作语义等内容.P3 ASM 的语法和语义定义相对简单一些(限于篇幅本文不予介绍),而这些内容不再变化,因此构造一个确认程序并证明其正确性具有更好的可重用性.

确认程序也可以借助于被认为可信的工具(如模型检验器和自动求解器)实现.比如,Lopes 等人基于 Z3 求解器的原型工具[9]完全可以作为确认程序,用于确认翻译前后的某种语义一致性;Kheradmand 等人基于 KEQ[11]提出了一种 P4 编译器的翻译确认实现方案[10].然而,一般不会尝试对此类工具进行正确性证明,一方面难度很大,另一方面,人们对此类工具已足够信任.与此不同,P3C 中确认程序将进行正确性证明.

确认程序的正确性可描述为[21]:$\forall S,C.Validate(S,C)=\text{true}\Rightarrow S\approx C$,其中,*S* 和 *C* 分别为任一具体的 P3 AST 和 P3 ASM,*Validate*(*S*,*C*)成真表示确认成功,$S\approx C$ 表示 *S* 和 *C* 语义上是一致的.

所有自定义寄存器的访问区间不重叠(参见第 2.4.2 节及第 3.4.1 节)的特性可简化确认程序正确性的证明.

确认程序是编译器的一部分,将会附加于翻译过程之后.包含确认程序的翻译过程可描述如下[21].

Comp′(*S*)=match *Comp*(*S*) with

|*Error*→*Error*

|*OK*(*C*)→if *Validate*(*S*,*C*) then *OK*(*C*) else *Error*

6 总结及未来工作展望

本文的主要贡献是设计了一种面向安全网络的可重构数据包解析器的专用硬件配置描述语言 P3,并提出

其可信实现方案.基于对可重构硬件基本需求的充分理解,经过软硬件设计人员的反复沟通与协同设计,最终明确了如文中展示的 P3 语言核心特性(第 2 节)及其编译结构(第 5 节).P3 语言及其编译器 P3C 的设计侧重于可信需求,因此本文重点描述了该语言的静态语义(第 3 节的类型系统)以及动态语义(第 4 节的操作语义)定义.

P3 语言语法的完整定义参见设计文档[17],该文档中也包含了 P3 编译器 P3C 设计中的中间语言(AST 和汇编语言)和目标语言(硬件配置文件描述语言)的完整定义,其他内容也会随着项目进展补充完善.限于篇幅,关于中间语言和目标语言,本文仅通过样例予以说明,参见第 5.1.3 节和第 5.1.4 节.

本文的核心内容是 P3 语言的设计,对于编译器 P3C 仅介绍其设计框架(参见第 5 节),不包括其实现.基于图 10 所示的编译器结构,P3C 各个部分的实现工作正在进展中.当前已完整实现部分及其各模块功能如图 14 所示,代码可下载[22],其中包括词法和语法分析(含语法分析的确认程序及其正确性验证)的完整实现(参见第 5.1.1 节和第 5.2.1 节),下载后可安装运行.

现阶段 P3C 项目正在开展的工作包括静态类型检查和产生汇编的模块,同时,硬件方面也在设计更多的使用样例,用于测试和协同设计.未来剩余的工作中,最具挑战性的部分是产生汇编过程的正确性验证(参见第 5.2.3 节),将采用翻译确认的方法实现一个 validator,并证明其正确性.确认程序本身的实现并不复杂,仅与 AST 和汇编代码的结构相关,与产生汇编代码的过程无关.确认程序的正确性证明涉及到 AST 和汇编语言的语义定义,前者参见第 4 节,后者已完成.剩余工作均在 Coq 中完成,有一定的工作量.一些看似简单的工作,在实现时也可能不简单(至少从代码量角度来看).比如,根据本文第 3 节和第 4 节的定义,针对 P3 操作语义,P3 类型系统的可靠性是显然的,在 Coq 实现中,虽然证明思路也是显然的,但给出相应的证明会有不小的工作量.

期待 P3C 项目的开展能促进相关工作的进一步研究,比如实现 P4 语言编译器本身或者翻译确认程序的机器证明(对于 P4 语言编译器的正确性验证,目前已有的工作主要集中在基于模型检验器或自动求解器进行翻译确认).

References:

[1] Broadcom Corporation. Broadcom BCM56850 StrataXGS®Trident II Switching Technology. Broadcom, 2013.

[2] Davie BS, Rekhter Y. MPLS: Technology and Applications. Morgan Kaufmann Publishers Inc., 2000.

[3] Hanks S, Meyer D, Farinacci D, *et al*. Generic Routing Encapsulation (GRE). RFC 2784, 2000.

[4] Mahalingam M, Dutt DG, Duda K, *et al*. Virtual eXtensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. RFC 7348, 2014.

[5] McKeown N. Software-defined networking. INFOCOM Keynote Talk, 2009,17(2):30−32.

[6] Bosshart P, Gibb G, Kim HS, *et al*. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. ACM SIGCOMM Computer Communication Review, 2013,43(4):99−110.

[7] Bosshart P, Daly D, Gibb G, *et al*. P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 2014,44(3):87−95.

[8] Liu J, Hallahan W, Schlesinger C, *et al*. P4v: Practical verification for programmable data planes. In: Proc. of the 2018 Conf. of the ACM Special Interest Group on Data Communication. ACM, 2018. 490−503.

[9] Lopes N, Bjørner N, McKeown N, *et al*. Automatically verifying reachability and well-formedness in P4 networks. Technical Report, 2016.

[10] Kheradmand A, Rosu G. P4K: A formal semantics of P4 and applications. arXiv Preprint arXiv:1804.01468, 2018.

[11] The K Framework Development Team. KEQ. 2017. https://github.com/kframework/k/tree/keq

[12] Pnueli A, Siegel M, Singerman E. Translation validation. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer-Verlag, 1998. 151−166.

[13] Benácek P, Pu V, Kubátová H. P4-to-Vhdl: Automatic generation of 100 Gbps packet parsers. In: Proc. of the 2016 IEEE 24th Annual Int'l Symp. on Field-programmable Custom Computing Machines (FCCM). IEEE, 2016. 148−155.

[14] Zhao Y, Yin SJ, Li XY. Reconfigurable unit design in a reconfigurable Ethernet packet parser. Computer Engineering & Science, 2020,42(2):220−228 (in Chinese with English abstract).

[15] P4 Language Consortium. P4 language specification, Version 1.0.4. 2017. https://p4.org/specs/

[16] Peterson LL, Davie BS. Computer Networks: A Systems Approach. Elsevier, 2007.

[17] P3 language specification, draft. 2019. https://github.com/leeehh/P3_language_compiler/raw/master/doc/P3_Compiler.pdf

[18] Jourdan JH, Pottier F, Leroy X. Validating LR (1) parsers. In: Proc. of the European Symp. on Programming. Berlin, Heidelberg: Springer-Verlag, 2012. 397–416.

[19] CompCert Home. 2008. http://compcert.inria.fr/

[20] Pottier F, Régis-Gianas Y. Menhir reference manual. 2018. http://gallium.inria.fr/~fpottier/menhir/manual.pdf

[21] Leroy X. Formal verification of a realistic compiler. Communications of the ACM, 2009,52(7):107–115.

[22] P3 language compiler. 2019. https://github.com/leeehh/P3_language_compiler

附中文参考文献:

[14] 赵宇,殷树娟,李翔宇.一种可重构以太网数据包解析器中可重构单元的设计.计算机工程与科学,2020,42(2):220–228.

李璜华(1996－),男,硕士生,主要研究领域为编译器,形式化验证.

李凌(1995－),男,学士,主要研究领域为编译器,形式化验证.

赵宇(1995－),男,硕士生,主要研究领域为数字集成电路系统与设计.

王生原(1964－),男,博士,副教授,CCF 高级会员,主要研究领域为程序设计语言与系统,编译器设计,形式化方法.

李翔宇(1977－),男,博士,副研究员,博士生导师,CCF 专业会员,主要研究领域为信息安全芯片,硬件安全.