数据流语言中数据频率的自动化处理技术①

刘桂林[1], 张 昱[1,2]

[1](中国科学院软件研究所 计算机科学实验室, 北京 100190)
[2](吉林大学珠海学院 符号计算与知识工程公共实验平台, 珠海 519041)

摘 要: 数据流分析和处理是计算机应用最常见的工作之一, 实际系统常常包括不同频率的数据流, 而现有的程序语言要求在程序中对数据频率进行显式的处理. 旨在提出一种新型的变频数据流处理框架, 针对基本的频率运算进行自动化处理. 我们基于函数式程序设计语言和依赖类型系统理论, 定义了数据流语言 FStream, 在程序的类型检查过程中对数据流频率进行检查和处理, 并给出了离散的 Simulink 模型到 FStream 的表示.

关键词: 依赖类型系统; 数据流语言; 类型推断; Haskell; Simulink

Automatic Checking and Calculation of Data Frequencies in Stream Languages

LIU Gui-Lin[1], ZHANG Yu[1,2]

[1](Laboratory for Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](Laboratory of Symbolic Computation and Knowledge Engineering, Zhuhai College, Jilin Univerisity, Zhuhai 519041, China)

Abstract: Data stream analysis and manipulation is very common in computer systems. Practical systems usually consist of data streams of different frequencies, however, dealing with data streams with different frequencies in existing programming languages has to be done explicitly and manually in programs. This paper proposes FStream–a framework for programming data streams with different frequencies, which supports automatic calculation. FStream is designed based on functional languages and dependent type systems. We also show an encoding of a discrete-time Simulink model in FStream.

Key words: dependent type system; stream languages; type inference; Haskell; Simulink

在云计算和大数据产业迅速发展的今天, 人们对计算机的计算速度, 系统的可靠性等方面有了更高的要求. 基于冯诺依曼体系结构的程序语言开始显现出固有的弱点: 对并行计算的支持性不够; 缺少清晰的数学性质, 难以证明程序的正确性等[1]. 以“异步控制, 数据驱动”为基本思想的数据流计算机和由数据的相关性决定程序执行顺序的数据流程序语言的出现极大的提高了计算的并行性.

数据流程序语言是基于数据驱动计算模型, 针对数据流系统结构的高级程序设计语言, 如 Lustre[2]. Lustre 是一种同步数据流语言, 它要求所有参与基本运算的数据流对应同一时间序列. Simulink[3]中离散时间上的“信号”则是每个采样周期就采样一次形成的数据序列, 当采样周期变化时, 只能变为原来的 N 倍或者 N 分之一(N 为正整数).

对数据流的时间信息进行抽象, 使用频率描述数据元素与时间的关系, 考虑到 Lustre 和 Simulink 在运算中对时间的要求, 程序员将不得不在程序中对数据流的频率进行检查: 判断多个数据流的频率是否一致或者是否成倍数关系. 然而, 一般情况下只有在运行时才能得到频率的值, 因此也就只能在动态检查阶段才能检查出不合法的运算. 如果可以在编译时检查出这些非法操作, 就能节约更多的资源, 并保证程序在频率上的正确性. 为实现这一目标, 本文试图提出一种新型的数据流语言—FStream, 来处理不同频率的数据流运算, 期望能够对一些基本的频率运算进行自动化处理, 并且对程序中数据流的频率一致性或倍数关系进行静态检查. 我们将基于依赖类型(dependent

① 收稿时间:2014-06-30;收到修改稿时间:2014-08-15

© 中国科学院软件研究所 http://www.c-s-a.org.cn

type)[4]的理论和技术, 在数据流类型中加入频率的信息, 在类型的层次对频率进行处理. 本文的主要内容包括:

1)设计 FStream 语言的类型系统, 定义相关的操作符和操作语义, 设计 FStream 的类型推断算法;

2)考虑到 Haskell 的诸多有益特征如: 惰性求值、强类型系统、支持 lambda 演算[5]和高阶函数[6]等, 我们使用 Haskell 实现了 FStream 语言的原型.

3)我们认为 FStream 框架具有足够的描述能力来处理常见离散数据流, 并提出了将离散的 Simulink 模型翻译为 FStream 程序的方法.

1 FStream的类型系统

类型系统是一个形式化方法, 它通过将表达式按值分类的方法来避免非法语句[7],减少程序在运行时产生的错误[8]. 定义一种程序语言, 首先要描述其语法: 定义其类型和项;其次要明确操作符和项的语义; 最后定义语言的类型规则. 若需要进一步划分类型的类别, 还需要定义 kind 和 kind 规则[9].

1.1 FStream 的类型、kind 和项

FStream 语言中的基本类型包括: 整数类型(Nat), 布尔类型(Bool). 基本类型上的操作符包括: 算术运算符(op_{math}) +、-、*、/、%; 逻辑运算符(op_{logic}) &&、||、!; 关系运算符(op_{comp}) ==, <,<=,>=;条件运算符 case, case x y z 意为当 x 为真时, 结果为 y, 否则为 z;

FStream 的类型还包括元组类型和函数类型, 适用于元组类型的操作符包括 fst 和 snd, 如 fst <3,false> 的结果为 3.

FStream 中每个数据流是一个定频率的数据序列, 由以下二者构成:

① 一个确定类型的数据序列, 其类型必须为基本类型或基本类型衍生的元组或函数类型;

② 数据流的频率, 即在单位时间内数据流的数据元素个数.

定义 1 是对 FStream 类型系统的形式化定义, 其中 e 是 FStream 的项, 其定义依赖于对 FStream 的操作符和操作语义的进一步讨论.

按照是否是基本类型及其衍生类型, 如定义 2 所示, 可将 FStream 的类型分为两种:

① 基本类型和仅由基本类型衍生的元组类型和函数类型;

② 数据流类型和由数据流类型参与衍生的元组类型和函数类型.

定义 1. FStream 语言的类型

τ ::= Nat	自然数数类型
| Bool	布尔类型
| *Stream* τ e	数据流类型
| $\tau_1 * \tau_2$	元组类型
| $\tau_1 \rightarrow \tau_2$	函数类型

定义 2. FStream 语言的 kind

K :: = NSTREAM | STREAM

FStream 在数据流上的一般性操作符包括:算术运算符(op_{fsmath})add, minus, multiply, divide; 逻辑运算符($op_{fslogic}$)and, or, not; 关系运算符(op_{fscomp})lt(小于), gt(大于), eq(等于), nlt(大于等于), ngt(小于等于)以及条件运算符 fscase. 参与基本运算的数据流的频率大小必须一致.

此外, FStream 还有以下特殊的操作符:

1) repeat, “repeat x e”得到一个频率为 e, 数据序列为“x,x,x…”的数据流.

2)replay, “replay s e”意为将数据流 s 转化为一个频率为 e 的数据流, 要求 e 是 s 的频率的整数倍(设为 n 倍), 转化方式为将 s 中每个数据元素重复 n 次.

3)sample, “sample s e”意为从数据流 s 抽样得到一个频率为 e 的数据流, 要求数据流 s 的频率是 e 的整数倍(设为 n 倍), 抽样的方法为从 s 的每 n 个数据元素中取第 n/2(小于 1 则置为 1)个元素.

4)delay, 其使用形式为“delay s a n”, 意为在数据流 s 的第一个数据元素之前添加 n 个值为 a 的数据元素.

5)replace, 其用法为“replace s a n“, 意为将数据流 s 的前 n 个数据元素替换为 a.

部分操作符的用例可参考表 1, time 为单位时间序列.

表 1　FStream 部分操作符示例

time	1	2	3
x	2 3	4 2	5 1
replay　x　4	2 2 3 3	4 4 2 2	5 5 1 1
sample　x　1	2	4	5

定义 3 形式化的描述了 FStream 的项, 其中, op 为基本类型上的双目操作符; op_{fs} 为数据流类型上的双目操作符; 项 stream $[e_1, e_2, e_3 \ldots]$ e 用于构造一个数据序列为 $e_1, e_2, e_3 \ldots$, 频率为 e 的数据流.

定义 3. FStream 语言的项

© 中国科学院软件研究所 http://www.c-s-a.org.cn

$e ::= x$
| 0,1,2...
| true | false
| $< e_1, e_2 >$ | fst e_1 | snd e_1
| $\lambda x:\tau.e$ | $e_1\ e_2$
| e_1 op e_2 | $!e_1$ | *case* $e_1\ e_2\ e_3$
| *s*tream $[e_1, e_2, e_3...]\ e_0$
| e_1 op_{fs} e_2 | *not* e_1 | *fscase* $e_1\ e_2\ e_3$
| repeat e_1 e_2 | replay e_1 e_2 | sample e_1 e_2
| delay e_1 e_2 e_3 | replace e_1 e_2 e_3

1.2 FStream 的类型规则和 Kind 规则

通过构造依赖类型, 将频率信息蕴含在类型之中, 得以在类型检查时判定频率间的相等关系和倍数关系: $E \mapsto a = b$ 表示 a 和 b 相等, $E \mapsto (a\ \%\ b) = 0$ 表示 a 可以整除 b. 类型规则的目的是对项进行类型上的约束[10], FStream 的类型规则如图 1 所示.

$$\frac{x:\tau \in \Gamma}{\Gamma \mapsto x:\tau}\ (\text{T-VAR}) \quad \frac{n = 0,1,2...}{\Gamma \mapsto x:\text{Nat}}\ (\text{T-NAT}) \quad \frac{\Gamma \mapsto n_1:Nat\ \ \Gamma \mapsto n_2:Nat}{\Gamma \mapsto n_1\ op_{math}\ n_2:Nat}\ (\text{T-ADD})$$

$$\frac{\Gamma \mapsto n_1:\text{Nat}\ \ \Gamma \mapsto n_2:Nat}{\Gamma \mapsto n_1\ op_{comp}\ n_2:\text{Bool}}\ (\text{T-COMP}) \quad \frac{\Gamma \mapsto n_1:\text{Bool}\ \ \Gamma \mapsto n_2:\text{Bool}}{\Gamma \mapsto n_1\ op_{logic}\ n_2:\text{Bool}}\ (\text{T-LOGIC})$$

$$\frac{}{\Gamma \mapsto true:\text{Bool}}\ (\text{T-TRUE}) \quad \frac{}{\Gamma \mapsto false:\text{Bool}}\ (\text{T-FALSE}) \quad \frac{\Gamma \mapsto e_1:\text{Bool}}{\Gamma \mapsto !e_1:\text{Bool}}\ (\text{T-NOT})$$

$$\frac{\Gamma \mapsto e_1:\text{Bool}\ \ \Gamma \mapsto e_2:\tau_1\ \ \Gamma \mapsto e_2:\tau_1}{\Gamma \mapsto case\ e_1\ e_2\ e_3:\tau_1}\ (\text{T-CASE}) \quad \frac{\Gamma \mapsto e_1:\tau_1 * \tau_2}{\Gamma \mapsto \text{snd}\ e_1:\tau_2}\ (\text{T-SND})$$

$$\frac{\Gamma \mapsto e_1:\tau_1\ \ \Gamma \mapsto e_2:\tau_2}{\Gamma \mapsto < e_1, e_2 >: \tau_1 * \tau_2}\ (\text{T-PROD}) \quad \frac{\Gamma \mapsto e_1:\tau_1 * \tau_2}{\Gamma \mapsto \text{fst}\ e_1:\tau_1}\ (\text{T-FST})$$

$$\frac{\Gamma \mapsto e_1:\tau_1 \to \tau_2\ \Gamma \mapsto e_2:\tau_1}{\Gamma \mapsto e_1\ e_2:\tau_2}\ (\text{T-APP})$$

$$\frac{\Gamma \mapsto e_i:\tau_1\ (i = 1,2,3...)\ \ \Gamma \mapsto \tau_1 :: \text{NSTREAM}\ \ \Phi \mapsto e:\text{Nat}\ \ E \mapsto e > 0}{\Gamma \mapsto stream\ [e_1, e_2, e_3...]\ e:\text{Stream}\ \tau_1\ e}\ (\text{T-STREAM})$$

$$\frac{\Gamma \mapsto e_1:\text{Stream Nat}\ e_2\ \ \Gamma \mapsto e_3:\text{Stream Nat}\ e_2}{\Gamma \mapsto e_1\ \text{op}_{\text{fsmath}}\ e_3:\text{Stream Nat}\ e_2}\ (\text{T-FSMATH})$$

$$\frac{\Gamma \mapsto e_1:\text{Stream Nat}\ e_2\ \ \Gamma \mapsto e_3:\text{Stream Nat}\ e_2}{\Gamma \mapsto e_1\ \text{op}_{\text{fscomp}}\ e_3:\text{Stream Bool}\ e_2}\ (\text{T-FSCOMP})$$

$$\frac{\Gamma \mapsto e_1:\text{Stream Bool}\ e_2\ \ \Gamma \mapsto e_3:\text{Stream Bool}\ e_2}{\Gamma \mapsto e_1\ \text{op}_{\text{fslogic}}\ e_3:\text{Stream Bool}\ e_2}\ (\text{T-FSLOGIC})$$

$$\frac{\Gamma \mapsto e_1:\text{Stream Bool}\ e_4\ \ \Gamma \mapsto e_2:\text{Stream}\tau\ e_4\ \Gamma \mapsto e_3:\text{Stream}\tau\ e_4}{\Gamma \mapsto fscase\ e_1\ e_2\ e_3:\text{Stream}\ \ e_2}\ (\text{T-FSCASE})$$

$$\frac{\Gamma \mapsto e_1:\text{Stream}\ \tau_1\ e_3\ \ \Phi \mapsto e_2:\text{Nat}\ \ E \mapsto e_3 <= e_2\ \ E \mapsto (e_2\ \%\ e_3) = 0}{\Gamma \mapsto replay\ e_1\ e_2:\text{Stream}\ \tau_1\ e}\ (\text{T-REPLAY})$$

$$\frac{\Gamma \mapsto e_1:\text{Stream}\ \tau_1\ e\ \Phi \mapsto e_2:\text{Nat}\ \ E \mapsto e_3 <= e_2\ E \mapsto (e_2\ \%\ e_3) = 0}{\Gamma \mapsto \text{sample}\ e_1\ e_2:\text{Stream}\ \tau_1\ e}\ (\text{T-SAMPLE})$$

$$\frac{\Gamma \mapsto e_1:\text{Stream}\ \tau_1\ e\ \Gamma \mapsto e_2:\ \tau_1\ \Gamma \mapsto e_3:Nat}{\Gamma \mapsto \text{delay}\ e_1\ e_2\ e_3:\text{Stream}\ \tau_1\ e}\ (\text{T-DELAY})$$

$$\frac{\Gamma \mapsto e_1:\text{Stream}\ \tau_1\ e\ \Gamma \mapsto e_2:\ \tau_1\ \Gamma \mapsto e_3:Nat}{\Gamma \mapsto \text{replace}\ e_1\ e_2\ e_3:\text{Stream}\ \tau_1\ e}\ (\text{T-REPLACE})$$

$$\frac{\Gamma \mapsto e_1:\ \tau_1\ \ \Gamma \mapsto \tau_1 :: \text{NS}\ \ \Phi \mapsto e_2:Nat\ E \mapsto (e_2 > 0) \Downarrow \text{true}}{\Gamma \mapsto \text{repeat}\ e_1\ e_2:\text{Stream}\ \tau_1\ e_2}\ (\text{T-REPEAT})$$

图 1 FStream 的类型规则

其中 T-Math 是 Nat 类型数据参与算术运算的规则: 两个 Nat 类型算术运算的结果仍然为 Nat 类型; 涉及数据流的基本运算的规则如 T-FSMATH、T-FSLOGIC 等确保了参数中数据流频率的一致性, 而 T-SAMPLE 和 T-REPLAY 中的 $E \mapsto (e_1\ \%\ e_2) = 0$ 则保证了 e_1 和 e_2 之间的“倍数关系”. 规则 T-STREAM 中 $\Gamma \mapsto \tau_1$ :: NSTREAM 则确保了 τ_1 类型的 kind 为 NSTREAM, FStreram 的 kind 规则如图 2 所示.

$$\frac{x :: K \in \Gamma}{\Gamma \mapsto x :: \text{K}}\ (\text{K-VAR}) \quad \frac{}{\Gamma \mapsto Nat :: NSTREAM}\ (\text{K-NAT})$$

$$\frac{}{\Gamma \mapsto Bool :: NSTREAM}\ (\text{K-BOOL})$$

$$\frac{\Gamma \mapsto \tau_1 :: NSTREAM\ \Gamma \mapsto \tau_2 :: NSTREAM}{\Gamma \mapsto \tau_1\ * \tau_2 :: NSTREAM}\ (\text{K-PROD1})$$

$$\frac{\Gamma \mapsto \tau_1 :: STREAM\ \Gamma \mapsto \tau_2 :: STREAM}{\Gamma \mapsto \tau_1\ * \tau_2 :: STREAM}\ (\text{K-PROD2})$$

$$\frac{\Gamma \mapsto \tau_1 :: STREAM\ \Gamma \mapsto \tau_2 :: NSTREAM}{\Gamma \mapsto \tau_1\ * \tau_2 :: \text{STREAM}}\ (\text{K-PROD3})$$

$$\frac{\Gamma \mapsto \tau_1 :: NSTREAM\ \Gamma \mapsto \tau_2 :: STREAM}{\Gamma \mapsto \tau_1\ * \tau_2 :: STREAM}\ (\text{K-PROD4})$$

$$\frac{\Gamma \mapsto \tau_1 :: NSTREAM\ \Gamma \mapsto \tau_2 :: NSTREAM}{\Gamma \mapsto \tau_1\ \to \tau_2 :: NSTREAM}\ (\text{K-FUNC1})$$

$$\frac{\Gamma \mapsto \tau_1 :: NSTREAM\ \Gamma \mapsto \tau_2 :: STREAM}{\Gamma \mapsto \tau_1\ \to \tau_2 :: STREAM}\ (\text{K-FUNC2})$$

$$\frac{\Gamma \mapsto \tau_1 :: STREAM\ \Gamma \mapsto \tau_2 :: STREAM}{\Gamma \mapsto \tau_1\ \to \tau_2 :: STREAM}\ (\text{K-FUNC3})$$

$$\frac{\Gamma \mapsto \tau_1 :: STREAM\ \Gamma \mapsto \tau_2 :: NSTREAM}{\Gamma \mapsto \tau_1\ \to \tau_2 :: STREAM}\ (\text{K-FUNC4})$$

$$\frac{\Gamma \mapsto \tau_1 :: NSTREAM\ \ \Phi \mapsto e:Nat}{\Gamma \mapsto Stream\ \tau_1\ e :: STREAM}\ (\text{K-STREAM})$$

图 2 FStream 的 kind 规则

规则 K_NAT 和 K_BOOL 定义了基本类型 Nat 和 Bool 的 kind 为 NSTREAM; 规则 K_PROD1 和 K_FUNC1 定义了由基本类型衍生的元组类型和函数类型的 kind 是 NSTREAM. 规则 K_STREAM 说明了数据流类型的 kind 为 STREAM. 其他规则说明了由数据流类型参与衍生的类型的 kind 都是 STREAM.

2 FStream的类型推断算法

类型推断在 FStream 中用于求解项的类型, 并对项进行类型检查. 设项 e 的类型为 T(类型变量), 将 $\Phi \mapsto e$: T 作为类型推断算法的输入, 则输出为能将 T 替换为类型的函数, 称为类型替换, 将它应用在 T 上就可以得到项 e 的实际类型, 完成这一替换的算法称为类型替换算法.

为清晰表达算法的层次性, 我们将该算法分成 4 部分: 算法 1 描述了类型替换算法, 将含有类型变量的类型系统看做类型模板, 则类型替换的输入输出都是类型模板. 算法 2 用于求解满足两个类型模板相等的类型替换, 其中操作符“。”[6,11]用于求解最一般化[12]的类型替换. 算法 3 是 kind 推断算法. 算法 4-1、4-2 是类型推断算法: 依据类型规则, 调用算法 2 和算法 3

© 中国科学院软件研究所 http://www.c-s-a.org.cn

求解项的类型.

算法 1. 类型替换算法

$\sigma(X)=T \quad \text{if } X \mapsto T \in \sigma$

$\sigma(Nat) = Nat$

$\sigma(Bool) = Bool$

$\sigma(Stream\ T\ N) = Stream\ \ \sigma(T)\ \ N$

$\sigma(T * S) = \sigma(T) * \sigma(S)$

$\sigma(T \to S) = \sigma(T) \to \sigma(S)$

算法 2. 类型归一算法

$typeUnif(Nat\ ;\ Nat)=[]$

$typeUnif(Bool\ ;\ Bool)=[]$

$typeUnif(T,\vec{T};S,\vec{S}) = \sigma' \circ \sigma \quad where \quad \sigma = typeUnif(T,S)$

$\sigma' = typeUnif(\sigma(\vec{T}),\sigma(\vec{S}))$

$typeUnif(Stream\ T\ \mathbf{M}\ ;\ Stream\ S\ N) = typeUnif(T,M;S,N)$

$typeUnif(e_1\ ;\ e_2) = [] \ \text{if}\ \ (e_1 = e_2)$

$typeUnif(x\ ;\ e) = [x \mapsto e]$ 如果变量 x 不在e中出现

$typeUnif(e\ ;\ y) = [y \mapsto e]$ 如果变量 y 不在e中出现

$typeUnif(T_1 * T_2\ ;\ S_1 * S_2) = typeUnif(T_1,T_2;S_1,S_2)$

$typeUnif(T_1 \to T_2\ ;\ S_1 \to S_2) = typeUnif(T_1,T_2;S_1,S_2)$

$typeUnif(x\ ;\ S) = [x \mapsto S]$ 如果变量 x 不在S中出现

$typeUnif(T;\ y) = [y \mapsto T]$ 如果变量 y 不在T中出现

$$typeUnif(x\ ;\ y) = \begin{cases} [] & \text{若x和y都是值，且相等} \\ [x \mapsto y] & \text{若x为变量} \\ [y \mapsto x] & \text{若y为变量} \\ \text{类型错误} & \text{否则} \end{cases}$$

其他情况下类型错误

算法 3. kind 推断算法

$kindInfere\ Nat = NSTREAM$

$kindInfer\ Bool = NSTREAM$

$$kindInfer(Stream\ tt\ i) = \begin{cases} STREAM & \text{if } kindInfer(tt) = NSTREAM \\ \text{类型错误} & \text{else} \end{cases}$$

$$kindInfer(tt1 * tt2) = \begin{cases} NSTREAM & \text{if（} kindInfer(tt1) = NSTREAM \\ & \&\& kindInfer(tt2) = NSTREAM\text{）} \\ STREAM & \text{else} \end{cases}$$

$$kindInfer(tt1 \to tt2) = \begin{cases} NSTREAM & \text{if（} kindInfer(tt1) = NSTREAM \\ & \&\& kindInfer(tt2) = NSTREAM\text{）} \\ STREAM & \text{else} \end{cases}$$

由于数据流类型的独特性，以下分两部分介绍 FStream 的类型推断算法.

算法 4-1 是基本类型和衍生类型上的类型推断算法. 对 Hindley–Milner 类型系统[13]进行扩展: 增加 Bool 类型、Nat 类型和元组类型可以描述 FStream 的基本类型、函数类型和元组类型. 根据类型规则扩展 Hindley–Milner 类型推断算法[14,15]可以得到算法 4-1.

算法 4-2 是数据流类型上的类型推断算法. 由于数据流类型上的一般性操作符要求参数中数据流的频率相同，因此在类型推断时需要判断两个项是否相等. 如应用算法 4-2 对项 (Stream [1,2] 2) *add* (Stream [3,10] 2) 进行类型推断，其过程如下所示:

$(Stream\ [1,2]\ 2)\ \ add\ \ (Stream\ [3,10]\ 2)$

$typeInfer(\Phi \mapsto (Stream\ [1,2]\ 2)\ \ add\ \ (Stream\ [3,10]\ 2) : T) = \sigma_2 \circ \sigma_1 \circ \sigma$

$\sigma = typeInfer(\Phi \mapsto Stream\ [1,2]\ 2 : Stream\ \ Nat\ M)$

$= [M \mapsto 2]$

$\sigma_1 = typeInfer(\Phi \mapsto Stream\ [3,10]\ 2 : Stream\ \ Nat\ \sigma(M))$

$= []$

$\sigma_2 = typeUnif(T, Stream\ \ Nat\ \ \sigma_1 \circ \sigma(M))$

$= [T \mapsto Stream\ Nat\ 2]$

$typeInfer(\Phi \mapsto (Stream\ [1,2]\ 2)\ \ add\ \ (Stream\ [3,10]\ 2) : T) =$

$[T \mapsto Stream\ Nat\ 2, M \mapsto 2]$

$typeInfer(\Phi \mapsto (Stream\ [1,2]\ 2)\ \ add\ \ (Stream\ [3,10]\ 2) : T)\ (T)$

$= Stream\ Nat\ 2$

类型规则 T-SAMPLE 和 T-REPLAY 中需要判断数据流的频率是否成倍数关系，此时需要先推断出数据流的类型，然后对类型中表示频率的项变量进行类型替换，最后后判断其值之间的关系.

算法 4-1. 类型推断算法(一)

$typeInfer(x_1 : T_1,..,x_n : T_n \mapsto x_i : T_i : T) = typeUnif(T,T_i)$

$typeInfer(\Gamma \mapsto n : T) = typeUnif(T,Nat)$

$typeInfer(\Gamma \mapsto e_1\ op_{math}\ e_2 : T) = \sigma_2 \circ \sigma_1 \circ \sigma$ where

$\sigma = typeUnif(T,Nat)$

$\sigma_1 = typeInfer(\sigma \circ \Gamma \mapsto e_2 : Nat)$

$\sigma_2 = typeInfer(\sigma_1 \circ \sigma \circ \Gamma \mapsto e_1 : Nat)$

$typeInfer(\Gamma \mapsto e_1\ op_{comp}\ e_2 : T) = \sigma_2 \circ \sigma_1 \circ \sigma$ where

$\sigma = typeUnif(T,Bool)$

$\sigma_1 = typeInfer(\sigma \circ \Gamma \mapsto e_2 : Nat)$

$\sigma_2 = typeInfer(\sigma_1 \circ \sigma \circ \Gamma \mapsto e_1 : Nat)$

$typeInfer(\Gamma \mapsto true : T) = typeUnif(T,Bool)$

$typeInfer(\Gamma \mapsto false : T) = typeUnif(T,Bool)$

$typeInfer(\Gamma \mapsto !e_1 : T) = \sigma_1 \circ \sigma$ where

$\sigma = typeUnif(T,Bool)$

$\sigma_1 = typeInfer(\sigma \circ \Gamma \mapsto e_1 : Bool)$

$typeInfer(\Gamma \mapsto e_1\ op_{logic}\ e_2 : T) = \sigma_2 \circ \sigma_1 \circ \sigma$ where

$\sigma = typeUnif(T,Bool)$

$\sigma_1 = typeInfer(\sigma \circ \Gamma \mapsto e_2 : Bool)$

$\sigma_2 = typeInfer(\sigma_1 \circ \sigma \circ \Gamma \mapsto e_1 : Bool)$

$typeInfer(\Gamma \mapsto case\ e_1\ e_2\ e_3 : T) = \sigma_2 \circ \sigma_1 \circ \sigma$ where

$\sigma = typeInfer(\Gamma \mapsto e_1 : Bool)$

$\sigma_1 = typeInfer(\sigma \circ \Gamma, t_1 : \sigma(e_1) \mapsto e_2 : \sigma(T))$

$\sigma_2 = typeInfer(\sigma_1 \circ \sigma \circ \Gamma, e_2 : \sigma_1 \circ \sigma(T) \mapsto e_3 : \sigma_1 \circ \sigma(T))$

$typeInfer(\Gamma \mapsto < e_1, e_2 > : T) = \sigma_2 \circ \sigma_1 \circ \sigma$ where

$\sigma = typeUnif(T, X * Y)$

--X,Y are fresh type variables

$\sigma_1 = typeInfer(\sigma \circ \Gamma \mapsto e_1 : \sigma(X))$

$\sigma_2 = typeInfer(\sigma_1 \circ \sigma \circ \Gamma \mapsto e_2 : \sigma_1 \circ \sigma(Y))$

$typeInfer(\Gamma \mapsto fst\ e_1 : T) = typeInfer(\Gamma \mapsto e_1 : T * Y)$

--*Y is a fresh type* var*iable*

$typeInfer(\Gamma \mapsto snd\ e_1 : T) = typeInfer(\Gamma \mapsto e_1 : X * T)$

--*X is a fresh type* var*iable*

$typeInfer(\Gamma \mapsto \lambda\ x\ e_1 : T) = \sigma_1 \circ \sigma$ where

$\sigma = typeUnif(T, X \to Y)$

--X,Y are fresh type variables

$\sigma_1 = typeInfer(x : \sigma_1(X), \sigma \circ \Gamma \mapsto x\ :\ \sigma(X))$

$typeInfer(\Gamma \mapsto e_1\ \ e_2 : T) = \sigma_1 \circ \sigma$ where

$\sigma = typeInfer(\Gamma \mapsto e_1 : X \to T)$

--X *is* a fresh type variable

$\sigma_1 = typeInfer(\sigma \circ \Gamma \mapsto e_2\ :\ \sigma(X))$

© 中国科学院软件研究所 http://www.c-s-a.org.cn

算法 4-2. 类型推断算法(二)

typeInfer($\Gamma \mapsto$ Stream $[e_1, e_2, ..., e_n]$ N : T)
= $\sigma_n \circ .. \circ \sigma_0 \circ \sigma$ *if* ($kindInfer(\sigma_n \circ ... \circ \sigma_0 \circ \sigma(X)) = NSTREAM$)
where σ = typeUnif(T;Stream X N)
--X *is* a fresh type variable
σ_0 = typeInfer($\Phi \mapsto N : Nat$)
σ_i = typeInfer($\sigma_{i-1} \circ .. \circ \sigma \circ \Gamma \mapsto e_i$: $\sigma_{i-1} ... \circ \sigma(X)$)
--Y *is* a fresh type variable

typeInfer($\Gamma \mapsto$ replay e_1 N : T)
= $\sigma_1 \circ \sigma$ if ($N\% \sigma_1 \circ \sigma(M) = 0 \,\&\&\, N >= \sigma_1 \circ \sigma(M)$)
where σ = typeInfer($\Gamma \mapsto e_1$: Stream X M)
--M *is* a fresh integer variable
σ_1 = typeUnif(T,Stream σ (X) N)

typeInfer($\Gamma \mapsto$ sample e_1 N : T)
= $\sigma_1 \circ \sigma$ if ($N\% \sigma_1 \circ \sigma(M) = 0 \,\&\&\, N >= \sigma_1 \circ \sigma(M)$)
where σ = typeInfer($\Gamma \mapsto e_1$: Stream X M)
--X,M *are* fresh type variables
σ_1 = typeUnif(T,Stream σ(X) N)

typeInfer($\Gamma \mapsto$ delay e_1 e_2 : T) = $\sigma_2 \circ \sigma_1 \circ \sigma$
where σ = typeInfer($\Gamma \mapsto e_1$: Stream X M)
--X,M *are* fresh type variables
σ_1 = typeUnif(T,Stream σ(X) $\sigma(M)$)
σ_2 = typeInfer($\sigma_1 \circ \sigma \circ \Gamma \mapsto e_2 : Nat$)

typeInfer($\Gamma \mapsto$ replace e_1 e_2 : T) = $\sigma_2 \circ \sigma_1 \circ \sigma$
where σ = typeInfer($\Gamma \mapsto e_1$: Stream X M)
--X,M *are* fresh type variables
σ_1 = typeUnif(T,Stream σ(X) $\sigma(M)$)
σ_2 = typeInfer($\sigma_1 \circ \sigma \circ \Gamma \mapsto e_2 : Nat$)

typeInfer($\Gamma \mapsto$ repeat e_1 N : T)
= $\sigma_1 \circ \sigma$ if(kindInfer($\sigma_1 \circ \sigma$(X))=NSTREAM &&N>0)
where σ = typeInfer($\Gamma \mapsto e_1 : X$) --X *is* a fresh type variable
$\sigma_1 = t$ typeUnif(T,Stream σ(X) N)

typeInfer($\Gamma \mapsto$ fsmath e_1 e_2 : T) = $\sigma_2 \circ \sigma_1 \circ \sigma$
where σ = typeInfer($\Gamma \mapsto e_1$: Stream *Nat* M)
--M *is* a fresh integer variable
σ_1 = typeInfer($\sigma \circ \Gamma \mapsto e_2$: Stream *Nat* $\sigma(M)$)
σ_2 = typeUnif(T,Stream Nat $\sigma_1 \circ \sigma(M)$)

typeInfer($\Gamma \mapsto$ fscomp e_1 e_2 : T) = $\sigma_2 \circ \sigma_1 \circ \sigma$
where σ = typeInfer($\Gamma \mapsto e_1$: Stream *Nat* M)
--M *is* a fresh integer variable
σ_1 = typeInfer($\sigma \circ \Gamma \mapsto e_2$: Stream *Nat* M)
σ_2 = typeUnif(T,Stream Bool $\sigma_1 \circ \sigma(M)$))

typeInfer($\Gamma \mapsto$ fslogic e_1 e_2 : T) = $\sigma_2 \circ \sigma_1 \circ \sigma$
where σ = *typeInfer*($\Gamma \mapsto e_1$: Stream Bool M)
--M *is* a fresh integer variable
σ_1 = typeInfer($\sigma \circ \Gamma \mapsto e_2$: Stream Bool $\sigma(M)$)
σ_2 = typeUnif(T,Stream Bool $\sigma_1 \circ \sigma(M)$))

typeInfer($\Gamma \mapsto$ fsnot e_1 : T) = $\sigma_1 \circ \sigma$
where σ = typeInfer($\Gamma \mapsto e_1$: Stream Bool M)
--M *is* a fresh integer variable
σ_1 = typeUnif(T,Stream Bool $\sigma(M)$)

typeInfer($\Gamma \mapsto$ fscase e_1 e_2 e_3 : T) = $\sigma_3 \circ \sigma_2 \circ \sigma_1 \circ \sigma$
where σ = typeInfer($\Gamma \mapsto e_1$: Stream Bool M)
--M *is* a fresh integer variable
σ_1 = typeInfer($\sigma \circ \Gamma \mapsto e_2$: Stream X $\sigma(M)$)
--X *is* a fresh *type* variable
σ_2 = typeInfer($\sigma_1 \circ \sigma \circ \Gamma \mapsto e_3$: Stream $\sigma_1(X)$ $\sigma_1 \circ \sigma(M)$))
σ_3 = typeUnif(T,Stream $\sigma_2 \circ \sigma_1$(X) $\sigma_2 \circ \sigma_1 \circ \sigma(M)$)

3 实现FStream原型

使用 Haskell 实现 FStream 语言的原型, 首先要实现其类型系统, 然后实现 FStream 的各种操作符. Haskell 的类型包括字符类型、布尔类型、整数类型、元组类型、列表类型、函数类型和自定义类型, 并且支持这些类型上的算术、逻辑和关系运算, 可以满足 FStream 对基本类型、元组类型和函数类型的需求, 无需作进一步的扩展.

3.1 Haskell 与依赖类型

FStream 语言的数据流类型是一个依赖类型, 而 Haskell 并不支持由项到类型的函数, 但是可以通过一些机制在 Haskell 中模拟和实现依赖类型的性质. 其中 type class 就是一个将类型和类型联系起来的重要的机制, 通过它可以实现类型层次的编程[16].

除此之外, Haskell 的 type class 机制还有其他的特性, 包括支持多个类型参数, 通过和函数依赖[17,18]的联合使用可以更准确的描述类型和类型之间的关系. McBride 等在文献[19]中利用 class 机制实现了类型层次的自然数, 并在此基础实现了定长的 Vector 类型. Yorgey 等则在文献[20]中利用 data type 实现了定长的 Vector 类型, 并通过 type family 机制实现了长度上的大小比较算符.

然而, 在 Haskell 中实现依赖类型依然存在局限性: 由于 Haskell 强调运行时态的值和编译时态的类型之间的区别, 为了将运行时的值和编译时的类型联系起来, 需要使用 singleton 类型, Richard 等在文献[21]中对这一问题做了详细讨论.

3.2 FStream 中数据流的频率及相关操作符的实现

将 FStream 的数据流看做一个定频率的数据序列, 首先需要实现类型层次的频率、在频率上的操作符; 其次定义数据流类型; 最后实现和数据流类型相关的操作符. FStream 在频率上的操作包括大小比较、相等和倍数关系判断, 以下为频率的定义以及相等判断的实现:

```
data Freq :: * where
  Zero :: Freq
  Succ :: Freq → Freq
  deriving(Eq,Show)

class (m ==? n) ~ True => (m :: Freq) == (n :: Freq)
instance ((m ==? n) ~ True) => m == n
```


© 中国科学院软件研究所 http://www.c-s-a.org.cn

```
type family (a :: Freq) == ? (b :: Freq) :: Bool
type instance Zero == ?  Zero = True
type instance Zero == ?  (Succ n) = False
type instance (Succ n) == ?  Zero = False
type instance (Succ n) == ?  (Succ m) = (m == ? n)
```

类似的可定义频率上大小比较操作符, 但是判断两个频率是否成倍数关系就相对复杂. 考虑到这样的性质: 若设 $m \% n = 0 \ \ (m > 0, n > 0)$ 成立, 则 m 和 n 的最大公约数必为 n, 而实现两个 Freq 类型数据的最大公约数可以用一下算法实现:

```
type family GCD (d::Freq) (m::Freq) (n::Freq)::Freq
type instance GCD d Zero Zero =d
type instance GCD (Succ m) (Succ n) = GCD (Succ d) m n
type instance GCD Zero (Succ m) Zero = Succ m
type instance GCD (Succ d) (Succ m) Zero = GCD (Succ Zero) d m
type instance GCD  Zero Zero (Succ n) =Succ n
type instance GCD (Succ d)  Zero (Succ n)= GCD (Succ Zero) d n
```

在此基础上可以得到约束两个 Freq 类型数据满足 MOD 条件的 class 如下:

```
type instance GCD (Succ d)  Zero (Succ n) = GCD (Succ Zero) d n
class (m >Zero,n>Zero , GCD Zero m  n == n) ⇒
                                MOD (m :: Freq) (n :: Freq)
instance (m >Zero,n>Zero , GCD Zero m  n == n) ⇒
                                MOD (m :: Freq) (n :: Freq)
```

3

.3 数据流及相关操作符的实现

利用 Haskell 中的列表实现数据流的数据序列, 可以如下定义 FStream 的数据流类型:

```
data FStream a (f :: Freq)= FS [a]
```

利用 Haskell 中将基本类型操作符映射到在列表上的操作符 map、zipWith 和 zipWith3, 并且在类型申明时对数据流的频率加以限制, 可以实现数据流类型的基本运算, 如在操作符 add 可以如下定义:

```
add ::(SingI f1,SingI f2,f1 == f2) ⇒
    FStream Integer f1  → FStream Integer f2  → FStream Integer f2
add (FS xs) (FS ys) = FS (zipWith (+) xs ys)
```

其中 add 操作符类型声明中的“f1 == f2 ”确保了两个数据流的频率的一致性.

Haskell 中 repeat 操作符用于将一个数据元素 x 转化为一个长度无限数据序列为[x,x,x,x,x,x…]的列表, 利用这一操作符, 可以实现 FStream 的 repeat 操作符:

```
repeat :: (SingI f) => a  →  Sing f  →  FStream a f
repeat x f = FS (Data.List.repeat x)
```

FStream 的 delay 和 replace 操作符分别用于在数据序列之前添加或替换部分数据:

```
delay :: SingI t => a  →  FStream a t  →  Int  →  FStream a t
delay x (FS xs) n = FS $ ((replicate n x) ++ xs)

replace :: SingI t => a  →  FStream a t  →  Int  →  FStream a t
replace x (FS xs) n  = FS $ ((replicate n x) ++ drop n xs)
```

Haskell 中“++”操作符可用于连接两个列表, 如[1,2,3] ++ [4,5]=[1,2,3,4,5], drop 操作符可用于去掉列表的前 n 个元素,借助于这些操作符可实现 delay 和 replace 操作符的功能.

除此之外, FStream 还有两个特殊的操作符 replay 和 sample, replay fstream f 要求频率 f 为数据流 fstream 频率(记为 f1)的整数倍, 倍数记为(f/f1), 此时我们需要一个从数据流获取频率大小的函数如下:

```
getFreq :: SingI t => FStream a t → Integer
getFreq = sing2Integer . freq
sing2Integer :: Sing (n :: Freq)  →  Integer
sing2Integer = freq2Integer . fromSing
freq2Integer  Zero = 0
freq2Integer  (Succ  m) = 1 + freq2Integer m
freq :: SingI t => FStream a t-> Sing t
freq _ = sing
```

通过使用 replicate 函数将原数据流的每个元素重复(f/f1)次, 结合类型为(a → [b]) → [a] → [b]的函数 concatMap, 可以实现 replay 操作符:

```
replay :: (SingI f1, SingI f2,MOD f2 f1) =>
                FStream a  f1 -> Sing f2 -> FStream a f2
replay ts@(FS xs) f2 = FS (concatMap (replicate n) xs) where
    n =  (fromInteger(div  (sing2Integer f2) (getFreq ts))) ::Int
```

sample 操作符与 replay 相反, 需要对原来的数据序列采样, 设需要在每 n 个元素中采样一个元素, 需要先利用 splitAt 函数将原数据序列划分为每 n 个元素一段, 然后在每段中取第 n/2(小于 1 则置为 1)个, . 可得到 sample 的具体实现代码如下所示:

```
sample :: (SingI f1, SingI f2,MOD f1 f2) =>
                FStream a f1 -> Sing f2 -> FStream a f2
sample  str@(FS xs) f
  = FS (map (take m . last) tschunks)
  where tschunks = chunks n xs
      n =  (fromInteger(div (getFreq str) (sing2Integer f))) ::Int
      m = if n>1 then (n/2) else  1
chunks :: Int -> [a] -> [[a]]
chunks _ [] = []
chunks n xs =
  let (ys, zs) = splitAt n xs
  in  ys : chunks n zs
```


© 中国科学院软件研究所 http://www.c-s-a.org.cn

4 离散的Simulink模型在FStream中的表示

Simulink 是一个用于建模, 信号模拟和分析的视图化数据流编程工具. Simulink 和 FStream 都可以通过“系统”操作数据信号来描述数据流模型.

Tripakis 等曾将离散的 Simulink 模型翻译到 Lustre 程序的表示[22], 本节的主要目的是将离散且只涉及整数和布尔类型数据的 Simulink 模型翻译到 FStream, 形式化的描述 Simulink 模型的语义, 用频率描述 Simulink 数据流元素与时间的关系. 如现有的 Simulink 模型中的有两个数据流 x 和 y, 其采样时间为 2 和 3, 对应的数据元素分别为 $(x_1,x_2,x_3....)$ 和 $(y_1,y_2,y_3....)$, 则对应的 FStream 程序中, 单位时间为 6, x 是频率为 3, y 是频率为 2 的数据流. 然而, Simulink 和 FStream 还有很多差异:

1)FStream 的数据流是离散的, 而 Simulink 的信号则是连续的, 即使其“Discrete library”中的模块生成的也是连续时间上的分段连续信号.

2)FStream 是一个强类型语言, 每一个表达式都有明确的类型, 而 Simulink 却并不强制要求明确系统中信号的类型.

3)数据流和时间关系的描述方式不同. Fstream 中数据流用频率描述了该数据流在单位时间内的元素个数, 而 Simulink 用信号的采样周期描述数据流和时间的对应关系.

4)FStream 语言的基本数据类型较简单, 只有整数类型和布尔类型, 而 Simulink 的数据类型包括实型(double), 布尔类型, 整数类型(int8.int16, int32, unit16, unit32).

由于两个系统(语言)之间的差异性, 在目前的 FStream 系统中实现完整的 Simulink 模型是很困难的, 我们这里仅给出一个 Simulink 的子集模型(离散的 Simulink 模型)在 FStream 中的表示. 转换过程所涉及的 Simulink 模型块如图 5 所示.

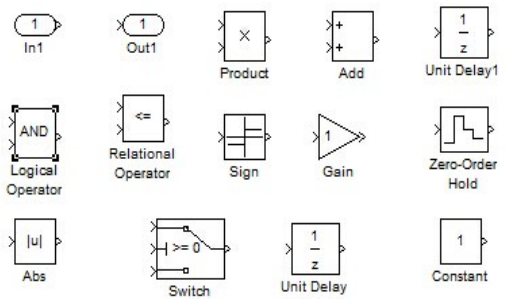

图 5 Simulink 的部分基本模块

为实现这一目标, 首先要形式化的描述 simulink 信号的数据类型; 其次计算每个模块的采样时间, 获得其输出信号的采样时间; 最后按照一定规则将模型翻译为 FStream 程序.

4.1 Simulink 中信号的数据类型

Simulink 中的基本数据类型有 9 种: boolean, double, single, int8, int16, int32, unit8, unit16, unit32. 由于 FStream 的类型系统中没有实型, 因此本文不讨论 double 类型信号的翻译, 设 SimNum = {single，int8，int16，int32，unit8，unit16，unit32}, SimBool={boolean}, 则 Simulink 中基本模块的类型可以用表 2 描述.

表 2 Simulink 中基本模块的类型

模块	类型
Constant	a,a ∈ simuNum
Product	a × … × a → a,a ∈ simuNum
Add	a × … × a → a,a ∈ simuNum
Logical Operator	b × … × b → b,b ∈ simBool
Relational Operator	a × a → b, a ∈ simuNum,b ∈ SimBool
ZOH,Unit Delay	a → a,a ∈ simuNum ⋃ *SimBool*
Inport,Outport	a → a,a ∈ simuNum ⋃ *SimBool*
Sign,Gain,Abs	a → a,a ∈ simuNum
Switch	a × b × a, a ∈ simuNum, b ∈ SimBool

在表 2 的基础上可以进一步归纳 Simulink 类型系统的类型规则, 并设计相应的类型推断算法, 可以推断出模型中信号的类型, 然后按如下规则得到 FStream 程序中对应的数据流的数据类型.

1)若信号的类型为 boolean, 则对应数据流的数据类型为 Bool.

2)若信号的类型为 single, int8, int16, int32, unit8, unit16, unit32, 则对应数据流的数据类型为 Nat.

4.2 Simulink 中信号的的采样时间

Simulink 中信号的采样时间由产生该信号的模块决定, 离散的模块其采样时间可以设定为具体数值, 当该值为-1 时, 表示其采样时间继承自输入, 此时其采样时间为所有输入信号的采样时间的最大公约数.

采样时间在 Simulink 中所起着类似类型系统的作用, 即模型可能由于采样时间错误而被系统所拒绝.

如图 6 中输入信号 In1 和 In2 的采样时间分别为 2 和 3, 模块 Gain 的采样时间也为 3, 则此模型会产生采样时间错误, 原因是 Simulink 模型须遵守以下的规则:

1) 若一个模块的输入信号和输出信号的采样时间不一致, 那么其输入信号必然是模块“Unit Delay”或

© 中国科学院软件研究所 http://www.c-s-a.org.cn

者“Zero-Order Hold”的输出信号.

2) “Unit Delay”模块的采样时间必须为其输出模块采样时间的倍数, 且所有的输出模块的采样时间必须一致.

3) “Zero-Order Hold”模块的输出模块的采样时间必须为其本身采样时间的倍数, 同样的, 所有的输出模块的采样时间必须一致.

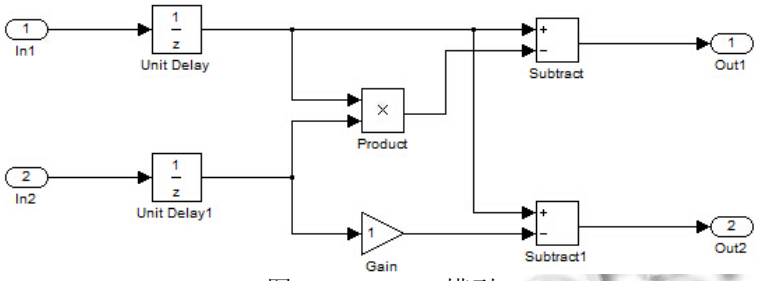

图 6　Simulink 模型

以上规则总结自 Matlab(版本为 7.14.0.739 (R2012a))和 Simulink(版本为 7.9)系统的错误说明和相关文档. 在图 8 的模型上应用规则 2, 可知 Product 模块的采样时间为 1(2 和 3 的公约数), 而 Gain 模块的采样时间为 3, 这和规则 2 中“所有的输出模块的采样时间必须一致”相矛盾.

为计算 Simulink 模型中所有模块的采样时间, 将模块看做点, 则需要翻译的 Simulink 模型符合有向无环图的特点, 在模型中按照节点的拓扑次序求得模型中所有块的采样时间. 由于块的采样时间就是其输出信号的采样时间, 由此可以得到所有信号的采样时间.

设模型信号 s1, s2, s3….的采样时间为 t1,t2,t3…., 则在对应的 FStream 程序中, 单位时间为 t1,t2,t3…的最大公倍数 t, 即数据流 s1,s2,s3 的频率为 t/t1, t/t2,t/t3.

4.3 翻译

Simulink 模型的由模块和连接线组成, 因此翻译工作分为对信号和基本模块的翻译.

在推断出 Simulink 的信号的数据类型, 计算得到每个信号的采样时间之后, 可以按照 4.1 和 4.2 节中的规则得到该信号在 FStream 中的类型, 每个信号对应 FStream 中一个数据流类型的变量, 其命名依据产生该信号的模块, 如模块 Add1 输出信号对应的变量命名为 add1_out.

模型中模块则可以按如下规则翻译为 FStream 中对应的操作符:

1) Product 和 Gain 模块可以翻译为 multiply 和 divide 操作符.

2) Add 模块可以翻译为 add 和 minus 操作符.

3) Logical Operator 模块翻译为 FStream 中对应的逻辑操作符 and, or, not.

4) Relational Operator 模块翻译为 FStream 中对应的关系操作符 lt,gt,nlt,ngt,eq.

5) Unit delay 模块翻译为 Fstream 中的 delay 操作符.

6) Zero-Order Hold 模块翻译为赋值操作符.

7) y=Sign(x)翻译为 y=fscase (x>0) 1 (case (x<0) -1 0), y=Abs(x)翻译为 y = fscase (x>0) x (-1*x), swith 模块也可以翻译为相应的 fscase 操作语句.

8) 当某个模块的输入信号的频率与输出信号的频率不同时, 先通过 sample 或者 replay 操作符将所有输入的频率调整至与输出一致.

4.4 示例

本翻译算法只接受合法的 Simulink 模型, 若图 5 中 Gain 模块的采样时间为 1, 则对应的 FStream 程序中单位时间为 6, 信号 in1 和 in2 的频率分别为 3 和 2, 设数据元素类型为 int32, 翻译得到的程序如下:

```
unit_delay_out=delay  in1  1  0
unit_delay1_out=delay  in2  1  0
product_out=multiply (replay unit_delay_out 6) (eplay unit_delay1_out 6)
gain_out=multiply (replay unit_delay_out 6) 1
subtract_out=minus (replay unit_delay_out 6) product_out
subtract1_out=minus (replay unit_delay_out 6) gain_out
out1=subtract_out
out2=subtract1_out
```

对以上程序中赋值语句中的项进行类型检查和类型推断, 可知 in1 和 unit_delay_out 的类型为 Stream Nat 3, in2 和 unit_delay1_out 的类型为 Stream Nat 2, product_out、gain_out、subtract_out、subtract1_out、out1、out2 的类型都是 Stream Nat 6.

通过将 Simulink 模型翻译为 FStream 程序, 用程序语言形式化的描述了模型的语义, 用类型简洁明了

© 中国科学院软件研究所 http://www.c-s-a.org.cn

的描述了模型中信号的数据类型和采样周期.

5 总结

本文提出了一种新的数据流处理框架语言 FStream, 通过构造依赖类型, 在其数据流类型中引入频率的表示, 并在程序的类型检查过程中实现对数据流频率自动化的检查和处理. 我们给出了 FStream 类型系统的形式化定义, 明确了 FStream 的操作符和操作语义, 实现了其类型推断算法, 并在 Haskell 中实现了系统原型, 最后我们给出了从离散的 Simulink 模型到 FStream 程序的翻译方法.

参考文献

1 郭青,陈国良,陈意云.数据流程序设计语言.计算机研究与发展,1990,27(4):22–30.

2 Halbwachs N, Caspi P, Raymond P, et al. The synchronous data flow programming language LUSTRE. Proc. of the IEEE, 1991, 79(9): 1305–1320 .

3 Simulink user's guide, 2014. http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf.

4 Martin-Lof P. Intuitionistic type theory. Naples: Bibliopolis, 1984.

5 Rojas R. A tutorial introduction to the lambda calculus. http://www. utdallas.edu/~gupta/courses/apl/lambda.pdf, 2004.

6 Lipovača M. Learn You a Haskell for Great Good! A Beginner's Guide. No Starch Press, 2011.

7 Pierce BC. Types and Programming Languages. MIT Press, 2002.

8 Cardelli L. Type systems. ACM Computing Surveys, 1996, 28(1): 263–264.

9 Jones MP. Typing haskell in haskell. Haskell Workshop. 1999, 43: 45–59.

10 Pierce BC, ed. Advanced Topics in Types and Programming Languages. MIT Press, 2005.

11 Gundry A. Type Inference, Haskell and Dependent Types, University of Strathclyde Department of Computer and Information Sciences. 2013.

12 Peyton Jones S, Vytiniotis D, Weirich S, et al. Simple unification-based type inference for GADTs. ACM SIGPLAN Notices. ACM, 2006, 41(9): 50–61.

13 Damas L, Milner R. Principal type-schemes for functional programs. Proc. of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM. 1982. 207–212.

14 Heeren B, Hage J, Swierstra SD. Generalizing hindley-milner type inference algorithms[Technical Report]. Institute of Information and Computing Science. University Utrecht, Netherlands. UU-CS-2002 -031. 2002.

15 Vaughan J. A proof of correctness for the Hindley-Milner type inference algorithm. http://www.jeffvaughan.net/docs/hmproof.pdf.

16 Chakravarty MMT, Keller G, Jones SP, et al. Associated types with class. ACM SIGPLAN Notices, ACM, 2005, 40(1): 1–13.

17 Chakravarty MMT, Keller G, Jones SP. Associated type synonyms. ACM SIGPLAN Notices, ACM, 2005, 40(9): 241–253.

18 Hallgren T. Fun with functional dependencies. Proc. of Joint CS/CE Winter Meeting, Chalmers Univerity. Varberg, Sweden. 2001.

19 McBride C. Faking it Simulating dependent types in Haskell. Journal of Functional Programming, 2002, 12(5): 375–392.

20 Yorgey BA, Weirich S, Cretin J, et al. Giving haskell a promotion. Proc. of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation. ACM, 2012. 53–66.

21 Eisenberg RA, Weirich S. Dependently typed programming with singletons. ACM SIGPLAN Notices, ACM, 2012, 47(12): 117–130.

22 Tripakis S, Sofronis C, Caspi P, et al. Translating discrete-time Simulink to Lustre. ACM Trans. on Embedded Computing Systems (TECS), 2005, 4(4): 779–818.

© 中国科学院软件研究所 http://www.c-s-a.org.cn