IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D32 (WP9a): Application to More Traditional Settings

Due date of deliverable: 31st March 2008
Actual submission date: 31st August 2008

Start date of project: 1st March 2005 Duration: 48 months

Lead contractor: Heriot-Watt University Revision: 1.21

Purpose: The purpose of this deliverable is to asses the applicability of the Hume resource models and analysis for conventional languages.

Results: The main results of this deliverable are the definition of imperative language and translations into Hume for analysis, and the application of the Hume analysis tools on a several small test cases. The results show an interesting correspondence between measuring the imperative program and analysing the Hume code.

Conclusion: The main conclusions are that Hume analysis can be used to compare relative time complexity between imperative programs.

Project co-funded by the European Commission within the 6^{th} Framework Programme (2002-06)			
Dissemination Level			
PU	Public		✳
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Application to More Traditional Settings

Gudmund Grov <G.Grov@hw.ac.uk>
Greg Michaelson <G.Michaelson@hw.ac.uk>
School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland

Hans-Wolfgang Loidl <hwloidl@tcs.ifi.lmu.de>
Institut für Informatik, Theoretische Informatik, Ludwig-Maximilians Universität, München

Christoph Herrmann <ch@cs.st-and.ac.uk>
Steffen Jost <jost@cs.st-andrews.ac.uk>
School of Computing Science, Univ of St Andrews, St Andrews, Scotland

Abstract

In this deliverable we show application of Hume analysis to imperative language through translation. A subset of ANSI-C, called MINI-C, is defined, and syntax, typing rules and operational semantics is given. We then define and implement formal translation rules into both a coordination and an expression layer version of Hume. We then apply both measure and cost the generated Hume, and measure the original MINI-C code for several small case studies, and discuss the results.

The positioning of this deliverable is discussed in Section 2.

Major Revisions		
Revision	Date	Changes
1.20	14 Aug. 2009	table-of-contents, positioning in Sec 2, fixed figures (addressing Review Report Year 4)
1.19	31 Aug. 2008	initial version

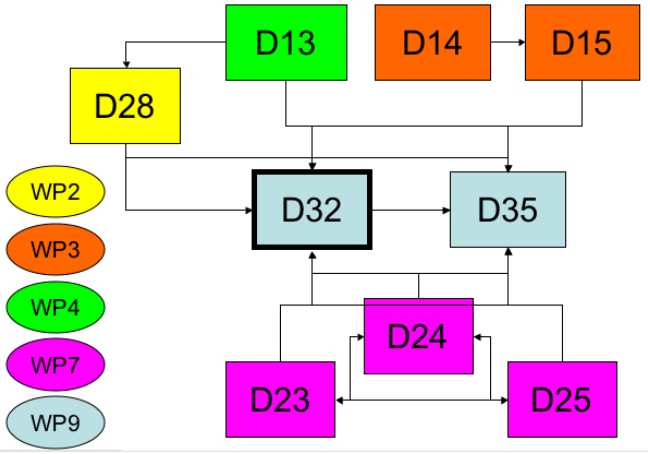
Contents

1 Introduction

This deliverable is part of WP9, where we attempt to determine how our resource models and analysis may be applied to traditional programming languages for embedded systems. Currently, C is still the most commonly used programming language in such systems. To apply our tools and techniques we will define a language MINI-C, which is comparable to a subset of ANSI-C.

We will define the abstract syntax and typing rules — and formalise the operational semantics. We then formally define the translation into two different Hume versions — and describe how these have been implemented in an automatic translator. Finally, we apply both measure and cost the generated Hume, and measure the original MINI-C code for several small case studies, and discuss the results.

2 Relation to other deliverables



D13	WP4	Prototype Implementation of space analyses
D14	WP3a	Report on WCET analysis
D15	WP3b	Prototype implementation of time analysis
D23	WP7c	Hume to HAM Translator
D24	WP7c	Hume Machine Code Compiler
D25	WP7c	Analyses integrated into compiler
D28	WP2b	Validation of Cost Model
D32	WP9a	Application to More Traditional Settings
D35	WP9b	Destructive Update and Dynamic Data Structures

Figure 1: Relationship Diagram of Deliverable

Figure 1 show the dependencies to other deliverables. Since we define a translation into Hume, there is a dependency on work package 7 which describes Hume and its compiler. This is required to measure the Hume code. We apply both the time and space analysis, described in work package 3 and 4, respectively. Moreover, since we rely on the correctness of the analysis, it also depends on deliverable 28 of work package 2, which validates the cost model. Finally, deliverable 35, which is also part of work package 9, extends MINI-C with pointers, thus relying directly on this deliverable.

3 Source Language

The source language MINI-C does not allow any function declaration, except the `main` function, which contains the whole program. Moreover, since MINI-C is a proper subset of ANSI-C, all declarations

precedes all statements. Thus, all variables, with the corresponding type, must be declared first. The types supported are integers and arrays of integers, which is sufficient to directly implement a large set of example. Also, note that C, and also MINI-C, does not have a separate boolean type: True is represented by `0`, and False is any value different from `0`.

Both standard variable and array assignments are provided in MINI-C. Moreover, conditionals (`if`) with and without `else` branching is supported, while repetition is based on a limited version of a `for`-statement. Finally, MINI-C allows reading from standard input via `scanf` and writing to standard output via `printf`. Overall, MINI-C constitutes a Turing-complete imperative language.

3.1 Syntax

program ::=	*prelude* `main()` *body*	
prelude ::=	$include_0$ ... $include_n$	$n \geq 0$
include ::=	`#include <`*id*`.h>`	
body ::=	$\{decl_1 \cdots decl_n\ stmt_1 \cdots stmt_m\}$	$n, m \geq 1$
cbody ::=	$stmt \mid \{stmt_1 \cdots stmt_n\}$	$n \geq 1$
decl ::=	`int` *id;*	
	| `int` *id*`[ int ];`	
stmt ::=	*id* `=` *expr;*	
	| *id*`[` $expr_1$ `]` `=` $expr_2$*;*	
	| `if (` *expr* `)` *cbody*	
	| `if (` *expr* `)` $cbody_1$ `else` $cbody_2$	
	| `for (` $stmt_1$ `;` *expr* `;` $stmt_2$ `)` *cbody*	
	| `printf ( "%d" ,` *id* `);`	
	| `scanf ( "%d" , &`*id* `);`	
expr ::=	*int*	
	| *id*	
	| *id*`[` *expr* `]`	
	| $expr_1$ *binop* $expr_2$	
	| `(` *expr* `)`	
binop ::=	`==` | `!=` | `<` | `<=` | `+` | `-` | `*` | `/` | `%`	

Figure 2: MINI-C Abstract Syntax

Figure 2 shows the abstract syntax of MINI-C. C libraries may be imported. This is required by the C compiler if `printf` or `scanf` is used (`stdio.h`). However, these are not required by the translator.

3.2 Typing Rules

Figure 3 shows the typing rules for MINI-C. M, N and O are variables representing terms and A,B and C are used for types. Two types are supported: arrays (type `array`) and integers (type `int`).

All statements returns either an element of `int` or `array` type, and a block or sequence return a value of the type of the last statement. `scanf` and `printf` are only supported for integers, thus we assume that they return an integer.

Note that the translator is independent of the return value of a statement since, for example, the following statement is not allowed:

```
x = (y = 1);
```

$$\frac{}{\Gamma \vdash N : \texttt{int}} N \in \mathbb{Z}^+ \qquad \frac{}{\Gamma \vdash M : A} (M : A) \in \Gamma, A \neq \texttt{decl} \qquad \frac{\Gamma \vdash M : \texttt{array} \qquad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M\texttt{[}N\texttt{]} : \texttt{int}}$$

$$\frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M\texttt{+}N : \texttt{int}} \qquad \frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M\texttt{-}N : \texttt{int}} \qquad \frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M\texttt{*}N : \texttt{int}}$$

$$\frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M\texttt{/}N : \texttt{int}} \qquad \frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M\texttt{\%}N : \texttt{int}} \qquad \frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M\texttt{==}N : \texttt{int}}$$

$$\frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M\texttt{!=}N : \texttt{int}} \qquad \frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M\texttt{<}N : \texttt{int}} \qquad \frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M\texttt{<=}N : \texttt{int}}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash M\texttt{=}N : A} \qquad \frac{\Gamma \vdash M : \texttt{array} \qquad \Gamma \vdash N : \texttt{int} \qquad \Gamma \vdash O : \texttt{int}}{\Gamma \vdash M\texttt{[}N\texttt{]=}O : O}$$

$$\frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : A}{\Gamma \vdash \texttt{if(}\, M \,\texttt{)}\, N : A} \qquad \frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : A \qquad \Gamma \vdash O : B}{\Gamma \vdash \texttt{if(}\, M \,\texttt{)}\, N \,\texttt{else}\, O : A}$$

$$\frac{\Gamma \vdash M : \texttt{int} \qquad \Gamma \vdash N : A \qquad \Gamma \vdash O : B}{\Gamma \vdash \texttt{if(}\, M \,\texttt{)}\, N \,\texttt{else}\, O : B} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : \texttt{int} \qquad \Gamma \vdash O : B \qquad \Gamma \vdash P : C}{\Gamma \vdash \texttt{for(}\, M \,\texttt{;}\, N \,\texttt{;}\, O \,\texttt{)}\, P : A}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : \texttt{int} \qquad \Gamma \vdash O : B \qquad \Gamma \vdash P : C}{\Gamma \vdash \texttt{for(}\, M \,\texttt{;}\, N \,\texttt{;}\, O \,\texttt{)}\, P : B} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{(}\, M \,\texttt{)} : A} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{\{}\, M \,\texttt{\}} : A}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash M \,\texttt{;}\, N : B} \qquad \frac{}{\Gamma \vdash \texttt{scanf("\%d", \&}M\texttt{)} : \texttt{int}} (M : \texttt{int}) \in \Gamma$$

$$\frac{}{\Gamma \vdash \texttt{printf("\%d", }M\texttt{)} : \texttt{int}} (M : \texttt{int}) \in \Gamma \qquad \frac{}{\Gamma \vdash \texttt{int}\ M : \texttt{decl}} (M : A) \notin \Gamma$$

$$\frac{}{\Gamma \vdash \texttt{int}\, M\texttt{[}N\texttt{]} : \texttt{decl}} (M : A) \notin \Gamma, N \in \mathbb{N}^+ \qquad \frac{\Gamma \vdash M : \texttt{decl} \qquad \Gamma, M : A \vdash N : \texttt{decl}}{\Gamma \vdash M \,\texttt{;}\, N : \texttt{decl}}$$

Figure 3: MINI-C Typing Rules

The value returned from a statement is thus ignored in all MINI-C programs.

Declarations return a dummy type `decl`, which is used to avoid conflicts with statements. For example, the "variable axiom" of Figure 3, require $(M : A) \in \Gamma$, while a declaration require $(M : A) \notin \Gamma$. Thus, annotating declaration with a type variable A, we can show that all programs are type-correct.

Most of the typing rules in Figure 3 are straightforward, but some of them needs some more explanation: unary minus is not supported in Hume, thus we do not support it either. It can simply be encoded as $0 - M$. Thus, all the rules involving `if` and `for` are duplicated to reflect this; the expression in a `for` statement may initially fail, thus it may return the type of the first and last argument.

3.3 Operational Semantics

We will give a stack based operational semantics: σ is used to represent a stack, while ℓ is a location on the stack, which is a natural number. Thus, $\ell + 1$ is the next location and $\ell + 2$ is the one that follows it. A stack is a map from a location to a value.

The environment $\mathcal{V}$ is split from the stack σ. $\mathcal{V}$ is map from a variable to a pair of a location and a tag. Two tags are supported, thus an environment pair is either: (int,n) which represents an integer n , or (array,k) which represents an array of length k.

3.3.1 Expressions

An expression cannot update neither the environment nor the stack, and always returns an integer value. Here, $\mathcal{V}, \sigma \vdash e \rightsquigarrow_e v$ denotes that the expression e under the environment $\mathcal{V}$ and stack σ evaluates (in a finite number of steps) to the integer value v. We let the set BinOp represent the set of binary operators. Note, that *true* is a number $n \neq 0$ and *false* is 0. Thus, for example, the operator == still returns a number.

$$\frac{n \in \mathbb{Z}}{\mathcal{V}, \sigma \vdash n \rightsquigarrow_e n} \quad \text{(Const Int)}$$

$$\frac{\mathcal{V}(x) = (\ell, \mathsf{int}) \qquad \sigma(\ell) = v}{\mathcal{V}, \sigma \vdash x \rightsquigarrow_e v} \quad \text{(Variable)}$$

$$\frac{\mathcal{V}, \sigma \vdash e \rightsquigarrow_e v \qquad v \in \mathbb{Z} \qquad \mathcal{V}(x) = (\ell, \mathsf{array}) \qquad 0 \leq v < k \qquad \sigma(\ell) = k \qquad \sigma(\ell + v + 1) = r}{\mathcal{V}, \sigma \vdash x\texttt{[}e\texttt{]} \rightsquigarrow_e r} \quad \text{(Array Variable)}$$

$$\frac{\mathcal{V}, \sigma \vdash e_1 \rightsquigarrow_e v_1 \qquad \mathcal{V}, \sigma \vdash e_2 \rightsquigarrow_e v_2 \qquad op \in \mathsf{BinOp} \qquad r = apply(op, \sigma, v_1, v_2)}{\mathcal{V}, \sigma \vdash e_1 \ op \ e_2 \rightsquigarrow_e r} \quad \text{(Binary Op)}$$

3.3.2 Declarations

A declaration can modify both the environment and the stack. Furthermore, we need to know the next free location on the stack, which may also be changed. We use *decl* to represent a declaration. Here, $\mathcal{V}, \sigma, \ell \vdash decl \rightsquigarrow_d \mathcal{V}', \sigma', \ell'$ denote that declaration *decl*, under the environment $\mathcal{V}$, stack σ with the next free locatoin ℓ, evaluates to the new environment $\mathcal{V}'$, new stack σ' with the next free location ℓ'. A sequence obviously first executes the first then the next statement; a variable requires one extra element on the stack; and an array uses the 1 plus the length of the array elements on the stack.

$$\frac{\mathcal{V}, \sigma, \ell \vdash decl_1 \rightsquigarrow_d \mathcal{V}', \sigma', \ell' \qquad \mathcal{V}', \sigma', \ell' \vdash decl_2 \rightsquigarrow_d \mathcal{V}'', \sigma'', \ell''}{\mathcal{V}, \sigma, \ell \vdash decl_1 \,; decl_2 \rightsquigarrow_d \mathcal{V}'', \sigma'', \ell''} \quad \text{(Sequence Declare)}$$

$$\frac{x \notin \operatorname{dom} \mathcal{V}}{\mathcal{V}, \sigma, \ell \vdash \texttt{int}\ x \rightsquigarrow_d \mathcal{V}[x \mapsto (\ell, \mathsf{int})], \sigma, \ell + 1} \quad \text{(Var Declare)}$$

$$\frac{x \notin \operatorname{dom} \mathcal{V} \qquad n > 0}{\mathcal{V}, \sigma, \ell \vdash \texttt{int}\ x\texttt{[}\ n\ \texttt{]} \rightsquigarrow_d \mathcal{V}[x \mapsto (\ell, \mathsf{array})], \sigma[\ell \mapsto n], \ell + n + 1} \quad \text{(Array Declare)}$$

3.3.3 Statements

Statements can only update the stack, and is represented by *stmt*. Here, $\mathcal{V}, \sigma \vdash \mathit{stmt} \rightsquigarrow_s \sigma'$ means that statement *stmt* evaluates, under enviroment $\mathcal{V}$ and stack σ, to the new stack σ'. Sequences of statements are executed after each other; a simple integer assignment evaluates the expression and updates the stack; in an array-update the element must be computed as well, and this must be within a valid range; the rules for conditional reflects that the `else` branch is optional; the first statement of a `for` loop is a normal statements, and the inductive *loop* definition is applied to the remaining part.

We do not give full semantics for the streams since this will complicate the rules. We only require that they are of integer type.

$$\frac{\mathcal{V}, \sigma \vdash \mathit{stmt}_1 \rightsquigarrow_s \sigma' \qquad \mathcal{V}, \sigma \vdash \mathit{stmt}_2 \rightsquigarrow_s \sigma''}{\mathcal{V}, \sigma \vdash \mathit{stmt}_1\,\texttt{;}\,\mathit{stmt}_2 \rightsquigarrow_s \sigma''} \quad \text{(Sequence Statement)}$$

$$\frac{\mathcal{V}(x) = (\ell, \mathsf{int}) \qquad \mathcal{V}, \sigma \vdash e \rightsquigarrow_e v}{\mathcal{V}, \sigma \vdash x\ \texttt{=}\ e \rightsquigarrow_s \sigma[\ell \mapsto v]} \quad \text{(Assign)}$$

$$\frac{\begin{array}{c}\mathcal{V}, \sigma \vdash e_1 \rightsquigarrow_e i \qquad \mathcal{V}(x) = (\ell', \mathsf{array}) \qquad 0 \le i < k \\ \sigma(\ell) = k \qquad \mathcal{V}, \sigma \vdash e_2 \rightsquigarrow_e v\end{array}}{\mathcal{V}, \sigma \vdash x\texttt{[}\ e_1\ \texttt{]}\ \texttt{=}\ e_2 \rightsquigarrow_s \sigma[\ell' + i + 1 \mapsto v]} \quad \text{(Array Assign)}$$

$$\frac{\mathcal{V}, \sigma \vdash e \rightsquigarrow_e c \qquad c \neq 0 \qquad \mathcal{V}, \sigma \vdash s \rightsquigarrow_s \sigma'}{\mathcal{V}, \eta \vdash \texttt{if(}\ e\ \texttt{)}\ s \rightsquigarrow_s \sigma'} \quad \text{(Conditional I True)}$$

$$\frac{\mathcal{V}, \sigma \vdash e \rightsquigarrow_e 0}{\mathcal{V}, \sigma \vdash \texttt{if(}\ e\ \texttt{)}\ s \rightsquigarrow_s \sigma} \quad \text{(Conditional I False)}$$

$$\frac{\mathcal{V}, \sigma \vdash e \rightsquigarrow_e c \qquad c \neq 0 \qquad \mathcal{V}, \sigma \vdash s_1 \rightsquigarrow_s \sigma'}{\mathcal{V}, \sigma \vdash \texttt{if(}\ e\ \texttt{)}\ s_1\ \texttt{else}\ s_2 \rightsquigarrow_s \sigma'} \quad \text{(Conditional II True)}$$

$$\frac{\mathcal{V}, \sigma \vdash e \rightsquigarrow_e 0 \qquad \mathcal{V}, \sigma \vdash s_2 \rightsquigarrow_s \sigma'}{\mathcal{V}, \sigma \vdash \texttt{if(}\ e\ \texttt{)}\ s_1\ \texttt{else}\ s_2 \rightsquigarrow_s \sigma'} \quad \text{(Conditional II False)}$$

$$\frac{\mathcal{V}, \sigma \vdash s_1 \rightsquigarrow_s \sigma' \qquad \mathcal{V}', \sigma' \vdash \mathit{loop}(e, s_2, s_3) \rightsquigarrow_s \sigma''}{\mathcal{V}, \sigma \vdash \texttt{for(}\ s_1\ \texttt{;}\ e\ \texttt{;}\ s_2\ \texttt{)}\ s_3 \rightsquigarrow_s \sigma''} \quad \text{(For Init)}$$

$$\frac{\begin{array}{c}\mathcal{V}, \sigma \vdash e \rightsquigarrow_e c \qquad c \neq 0 \\ \mathcal{V}, \sigma \vdash s_3\ \texttt{;}s_2 \rightsquigarrow_s \sigma' \qquad \mathcal{V}', \sigma' \vdash \mathit{loop}(e, s_2, s_3) \rightsquigarrow_s \sigma''\end{array}}{\mathcal{V}, \sigma \vdash \mathit{loop}(e, s_2, s_3) \rightsquigarrow_s \sigma''} \quad \text{(For True)}$$

$$\frac{\mathcal{V}, \sigma \vdash e \rightsquigarrow_e 0}{\mathcal{V}, \sigma \vdash \mathit{loop}(e, s_2, s_3) \rightsquigarrow_s \sigma} \quad \text{(For False)}$$

$$\frac{\mathcal{V}(x) = (\ell', \mathsf{int}) \qquad n \in \mathbb{Z}}{\mathcal{V}, \sigma \vdash \texttt{scanf("\%d",\&}x\texttt{)} \rightsquigarrow_s \sigma[\ell' \mapsto n]} \quad \text{(SCANF)}$$

$$\frac{\mathcal{V}(x) = (\ell, \mathsf{int}) \qquad \eta(\ell) = n \qquad n \in \mathbb{Z}}{\mathcal{V}, \sigma \vdash \texttt{printf("\%d", }x\texttt{)} \rightsquigarrow_s \sigma} \quad \text{(PRINTF)}$$

4 Translating MINI-C into Hume

4.1 Overview

A Hume program consists of boxes and wires. Transitions are achieved by a purely functional expression layer inside of a box, and input and output of boxes are linked by wires to other boxes in the coordination layer.

The translation into the coordination layer is in itself more interesting, while the the resource analysis are not well-developed for the coordination layer. Being able to compare the two representations will be a quite interesting study, although this is taken further here. Following these motivations, two translations from MINI-C into Hume has been defined: an explicit state coordination layer version; and an implicit state expression layer version which enables application of the resource analysis and models.

In the next section we will give the set of formal rules describing the translation from MINI-C into the coordination-layer version of Hume, while the following section provides the formal translation rules from MINI-C into the expression-layer version.

We will follow a functional style were the function $f(x, y)$ is written $f\ x\ y$. Moreover, we use lists as an underlying data structure, where :: is the list constructor, and [] is the empty list. Moreover, we write $[x, y]$ for the list consisting of x and y. @ is used to represent concatenation of two lists. Further, we assume functions that return the first element (*head*), the tail (*tail*) and reverses (*rev*) a list. Our meta-language used to describe the translation has local binding via the *LET-IN* operators. Furthermore, *RETURN* allows a rule to return a value in addition to the code generated – while *FRESH()* generates a fresh name. Conditionals are supported by an *IF-THEN-ELSE* (*IF-THEN*) clause.The *tt* and *ff* constant are the booleans in the language, and the boolean operators *NOT*, *OR* and *AND* are supported. `This font` is used for the generated Hume code, `this font` is used for the MINI-C code, while *this font* is used on the meta-language. Generated Hume code are underlined, to separate MINI-C and Hume. A translation rule is written using the infix $\rightsquigarrow$ operator, and meta-functions are defined by the $=$ operator: $\rightsquigarrow$ will generate code, while $=$ will return meta-values.

Some rules and functions are independent of the two translations. They are described in the remaining parts of this section.

4.2 Types

MINI-C support two types: `int` and arrays of `int`s. A MINI-C `int` is 16 bit, and is thus represented as an `int 16` Hume type. Hume does not support arrays, but the similar vector type. Thus, a MINI-C array is represented as a vector in Hume. The type translation is defined by the $\mathcal{T}_T$ (pronounced 'tt'). It accepts a well-founded MINI-C variable declaration, and produces a Hume type declaration.

$$\begin{array}{lcl} \mathcal{T}_T\ (\texttt{int}\ id) & \rightsquigarrow & \underline{\texttt{int 16}} \\ \mathcal{T}_T\ (\texttt{int}\ id\texttt{[}\ int\texttt{]}) & \rightsquigarrow & \underline{\texttt{vector}\ int\ \texttt{of int 16}} \end{array}$$

Note that the first element of a vector is 1 and 0 for a MINI-C array, which is handled by the translation of get and update operations of arrays(vectors).

4.3 Binary Operators

$\mathcal{T}_o$ defines the translation of the binary operators – it accepts a MINI-C operator and return the Hume representation. We assume the same precedence of the operator, thus making the MINI-C and Hume versions identical:

$$\begin{array}{lcl}
\mathcal{T}_o\ (\texttt{==}) & \leadsto & \underline{\texttt{==}} \\
\mathcal{T}_o\ (\texttt{!=}) & \leadsto & \underline{\texttt{!=}} \\
\mathcal{T}_o\ (\texttt{<}) & \leadsto & \underline{\texttt{<}} \\
\mathcal{T}_o\ (\texttt{<=}) & \leadsto & \underline{\texttt{<=}} \\
\mathcal{T}_o\ (\texttt{+}) & \leadsto & \underline{\texttt{+}} \\
\mathcal{T}_o\ (\texttt{-}) & \leadsto & \underline{\texttt{-}} \\
\mathcal{T}_o\ (\texttt{*}) & \leadsto & \underline{\texttt{*}} \\
\mathcal{T}_o\ (\texttt{/}) & \leadsto & \underline{\texttt{div}} \\
\mathcal{T}_o\ (\texttt{\%}) & \leadsto & \underline{\texttt{mod}}
\end{array}$$

As the typing rules of Figure 3 shows, the `==`, `!=`, `<` and `<=` operators accepts two integers and return an integer. Hence `0` and `0 == 1` are the equivalent. In Hume, however, these operators returns boolean values. These deviates are handled by the expression translation rules.

4.4 The State Space

In the Hume versions, variables are not declared as in MINI-C. Here, the MINI-C declaration are used to create a state space which are input and output to the box(es) of the program. THE *statespace* function uses the declarations, and returns the statespace as a list of pairs. Each pair holds the name and type of each element.

$$\begin{array}{lcl}
\mathit{statespace}\ (\texttt{int}\ \mathit{id}) & = & [(\mathit{id},\ \mathcal{T}_T\ (\texttt{int}\ \mathit{id}))] \\
\mathit{statespace}\ (\texttt{int}\ \mathit{id}\texttt{[}\ \mathit{int}\ \texttt{]}) & = & [(\mathit{id},\ \mathcal{T}_T\ (\texttt{int}\ \mathit{id}\texttt{[}\ \mathit{int}\ \texttt{]}))] \\
\mathit{statespace}\ (\mathit{decl}_1\ \texttt{;}\ \mathit{decl}_2) & = & \mathit{statespace}\ (\mathit{decl}_1)\ @\ \mathit{statespace}\ (\mathit{decl}_2)
\end{array}$$

4.5 Auxiliary Rules and Functions

In addition to the rules and functions above, there are a set of auxiliary rules and functions which are hard to classify. The next section presents these rules, and the following section contains the auxiliary functions.

4.5.1 Auxiliary Rules

Firstly, *head* creates the head (`in`/`out`) list of a box. As input , it accepts the state space list, discussed above, in addition to a label l. This label postfixes the variable name, and is useful since all the input and output definitions for a box can be created by this function. The empty string "" is used for variables that are not post-fixed by a label.

$$\begin{array}{lcl}
\mathit{head}\ ([\mathit{id}, T] :: [])\ l & \leadsto & \underline{\mathit{id}\ l\ ::\ T} \\
\mathit{head}\ ([\mathit{id}, T] :: \mathit{sp})\ l & \leadsto & \underline{\mathit{id}\ l\ ::\ T\ \texttt{,}\ \mathit{head}\ \mathit{sp}\ l}
\end{array}$$

Inside a box, when patterns matches the inputs, the variables are bound to these values. To enhance readability, the bound names are the same as the declarations in MINI-C. The *patt* rule creates such a pattern based on the given state-space:

$$\begin{array}{lcl}
\mathit{patt}\ ([\mathit{id}, T] :: []) & \leadsto & \underline{\mathit{id}} \\
\mathit{patt}\ ([\mathit{id}, T] :: \mathit{sp}) & \leadsto & \underline{\mathit{id}\ \texttt{,}\ \mathit{patt}\ \mathit{sp}}
\end{array}$$

Due to branching, there are cases where we would like to ignore inputs (of the full statespace). This is achieved by a list of `*`s with the length of the state-space. The *ignore* rule creates a list of `*`, based on the given state-space (The length of the state-space would have been sufficient):

$$\begin{array}{lcl} \mathit{ignore}\ (e :: []) & \leadsto & \underline{\texttt{*}} \\ \mathit{ignore}\ (e :: \mathit{sp}) & \leadsto & \underline{\texttt{*}\ ,\ \mathit{ignore}\ \mathit{sp}} \end{array}$$

The full state-space is sent between the boxes. Thus, each wire declaration will be quite similar for all the boxes. The *wires* rule, create a wire declaration with a given box name and the state space. It also uses the labeling described above:

$$\begin{array}{lcl} \mathit{wires}\ \mathit{box}\ ([\mathit{id}, T] :: [])\ l & \leadsto & \underline{\mathit{box}.\mathit{id}\ l} \\ \mathit{wires}\ \mathit{box}\ ([\mathit{id}, T] :: \mathit{sp})\ l & \leadsto & \underline{\mathit{box}.\mathit{id}\ l\ ,\ \mathit{wires}\ \mathit{box}\ \mathit{sp}\ l} \end{array}$$

Since Hume programs, compared to MINI-C, requires that all variables has an initial value, we use the following rules to give them a dummy initial value of `0`:

$$\begin{array}{lcl} \mathit{init\_}\ \mathit{vect}\ 0 & \leadsto & \\ \mathit{init\_}\ \mathit{vect}\ s & \leadsto & \underline{\texttt{,}\ \texttt{0}\ \mathit{init\_}\ \mathit{vect}\ (s-1)} \\ & & \\ \mathit{init\_}\ \mathit{val}\ (\texttt{int 16}) & \leadsto & \underline{\texttt{0}} \\ \mathit{init\_}\ \mathit{val}\ (\texttt{vector 0 ..}\ s\ \texttt{of int 16}) & \leadsto & \underline{\texttt{<<}\ \texttt{0}\ \mathit{init\_}\ \mathit{vect}\ s\ \texttt{>>}} \\ & & \\ \mathit{init\_}\ \mathit{values}\ ([\mathit{id}, T] :: []) & \leadsto & \underline{\mathit{init\_}\ \mathit{val}\ T} \\ \mathit{init\_}\ \mathit{values}\ ([\mathit{id}, T] :: \mathit{sp}) & \leadsto & \underline{\mathit{init\_}\ \mathit{val}\ T\ ,\ \mathit{init\_}\ \mathit{values}\ \mathit{sp}} \end{array}$$

init_ wires behaves like *wires*, but gives each variable the initial dummy value as well.

$$\begin{array}{lcl} \mathit{init\_}\ \mathit{wires}\ \mathit{box}\ ([\mathit{id}, T] :: [])\ l & \leadsto & \underline{\mathit{box}.l\ \texttt{initially}\ \mathit{init\_}\ \mathit{val}\ T} \\ \mathit{init\_}\ \mathit{wires}\ \mathit{box}\ ([\mathit{id}, T] :: \mathit{sp})\ l & \leadsto & \underline{\mathit{box}.\mathit{id}\ l\ \texttt{initially}\ \mathit{init\_}\ \mathit{val}\ T\ ,\ \mathit{init\_}\ \mathit{wires}\ \mathit{box}\ \mathit{sp}\ l} \end{array}$$

Finally, the *replace* rule is for example used to represent a MINI-C assignment. It requires the state-space, the variable to be replace, and an expression as input, and replaces the correct variable with the expression, and leaves the other unchanged:

$$\begin{array}{lcl} \mathit{replace}\ ([\mathit{id}, T] :: [])\ x\ e & \leadsto & \mathit{IF}\ \mathit{id} = x\ \mathit{THEN}\ \underline{e}\ \mathit{ELSE}\ \underline{\mathit{id}} \\ \mathit{replace}\ ([\mathit{id}, T] :: \mathit{sp})\ x\ e & \leadsto & \mathit{IF}\ \mathit{id} = x\ \mathit{THEN}\ \underline{e}\ \mathit{ELSE}\ \underline{\mathit{id}\ ,\ \mathit{replace}\ \mathit{sp}\ x\ e} \end{array}$$

4.5.2 Auxiliary Functions

The first function *istreams* takes a statement as input (including sequences), and creates a list holding only the stream variables. All other statement returns an empty list:

$$\begin{array}{lcl} \mathit{istreams}\ (\mathit{stmt}_1\ \texttt{;}\ \mathit{stmt}_2) & = & \mathit{istreams}\ (\mathit{stmt}_1)\ @\ \mathit{istreams}\ (\mathit{stmt}_2) \\ \mathit{istreams}\ (\texttt{scanf (\%d , \&}\mathit{id}\ \texttt{)}) & = & [(\mathit{FRESH()}, \texttt{int 16})] \\ \mathit{istreams}\ \_ & = & [] \end{array}$$

The second, and final function, is *isNumeric*. It checks if a MINI-C expression returns a number (or a boolean):

$$\begin{array}{lcl} \mathit{isNumerical}(\mathit{int}) & = & \mathit{tt} \\ \mathit{isNumerical}(\mathit{id}) & = & \mathit{tt} \\ \mathit{isNumerical}(\mathit{id}\texttt{[}\mathit{expr}\texttt{]}) & = & \mathit{tt} \\ \mathit{isNumerical}(\mathit{expr}_1\ \mathit{binop}\ \mathit{expr}_2) & = & \mathit{binop} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\} \\ \mathit{isNumerical}(\texttt{ (}\mathit{expr}\texttt{) }) & = & \mathit{isNumerical}(\mathit{expr}) \end{array}$$

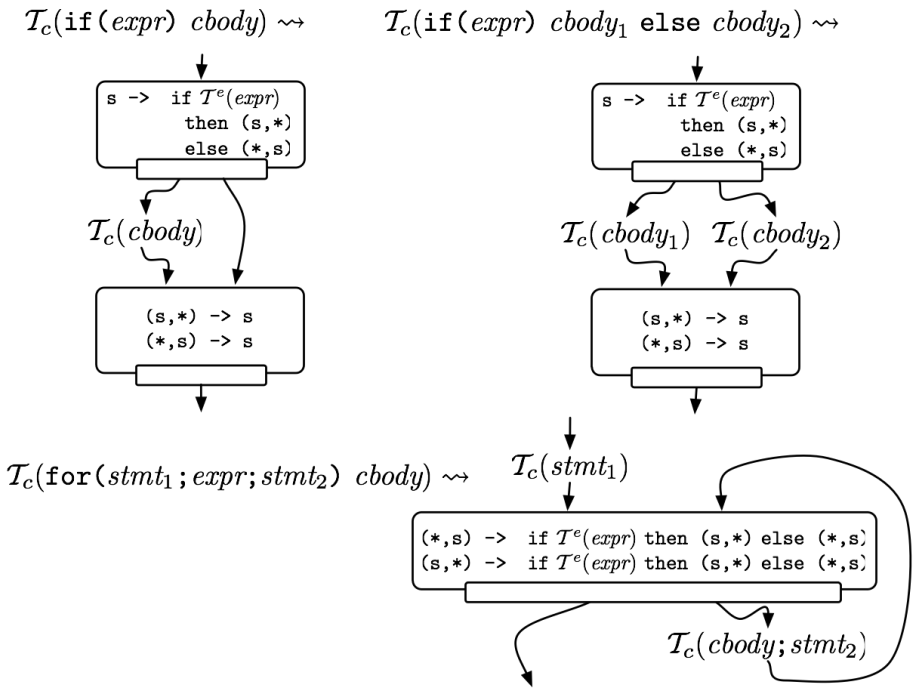


Figure 4: Box representation of control statements

5 Translating MINI-C into Coordination-Layer Hume Version

5.1 Overview

In the coordination representation of MINI-C we exploits the coordination layer as much as possible. Here, sequential execution is achieved directly, since boxes can fail on their input, and the state space is sent through the boxes. Consequently, at a given time all but one box will always fail on their input, and on termination all boxes will (indefinitely) fail. Obviously, with the current scheduling, this will result in a massive scheduling overhead.

Now the following four MINI-C statements are represented as one box:

- in standard assignment, the updated variable is replaced by the translated left hand side of the assignment, while all other variables are left unchanged;
- an array assignment is similar to a normal assignment, but uses the built-in Hume `vector` function to update a vector element;
- `scanf` is similar to an assignment, with the new value accepted as an additional input, wired to standard input;
- `printf` is similar to an identity box, with an additional output wired to standard output.

Figure 4 illustrates graphically the translation rules for the remaining control statement:

- a conditional is represented by a fork and join box, with the left and possible right body of the conditional between them on each branch. If the conditional does not contain an `else` branch, then the right side of the fork is directly wired to the right side of the join;
- after the initialization statement, looping in a `for` statement is achieved by a feedback loop and a test for the termination condition;

In addition to the translated statements, the program will have two additional boxes: one that initialises execution; and one that terminates execution.

The next section defines the translation rules for the expressions, This is followed by the formal translation rules for the statements. Finally, the translation rules for a complete program is given.

5.2 Expressions

The translation of MINI-C expressions is defined by $\mathcal{T}_e^c$. It accepts a MINI-C expression and returns a Hume expression. Since expressions may be nested, $\mathcal{T}_e^c$ is inductive:

$$
\begin{array}{lcl}
\mathcal{T}_e^c\ (int) & \leadsto & \underline{int} \\
\mathcal{T}_e^c\ (id) & \leadsto & \underline{id} \\
\mathcal{T}_e^c\ (id\texttt{[}expr\texttt{]}) & \leadsto & \underline{id\ \texttt{@(}}\ \mathcal{T}_e^c\ (expr)\ \underline{\texttt{+ 1)}} \\
\mathcal{T}_e^c\ (expr_1\ binop\ expr_2) & \leadsto & \mathcal{T}_e^c\ (expr_1)\quad \mathcal{T}_o\ (binop)\quad \mathcal{T}_e^c\ (expr_2) \\
\mathcal{T}_e^c\ (\ \texttt{(}expr\texttt{)}\ ) & \leadsto & \underline{\texttt{(}}\ \mathcal{T}_e^c\ (expr)\ \underline{\texttt{)}}
\end{array}
$$

The first two rules follows directly. Array lookup is replaced by vector lookup (`@`). However, since the first index of an array is 0, and the first of a vector index is 1, `1` is added to the index.

5.3 Statements

$\mathcal{T}_s^c$ defines the translation rules for MINI-C statements. It translates MINI-C statements into boxes and their corresponding wires. The boxes are rather straightforward to translate. The wires, on the other hand, are more problematic. This follows from the fact that source and destinations are required and due to nesting, branching and looping, all required information is not always present. To achieve a correct wiring, the rule is augmented with an environment ρ consisting of the following parts: a boolean context *ctx*, which states whether or not a wire declaration should be created for the *previous* box; a pair *inp* consisting of an identifier and list of wires; and a pair *out* of two list of wires; a list of lists *comp*, required for nesting. An element p of ρ is projected by ρ_p, and p is updated to e by $\rho[p \mapsto e]$. An empty environment ρ_{empty} has the following properties:

$$
\begin{array}{lcl}
\rho_{empty_{ctx}} & = & tt \\
\rho_{empty_{inp}} & = & (\text{""}, []) \\
\rho_{empty_{out}} & = & ([], []) \\
\rho_{empty_{comp}} & = & [[]]
\end{array}
$$

A rule $\mathcal{T}_s^c$ accepts a MINI-C statement and an environment ρ and returns an updated environment ρ', which may or may not be equal to ρ. In addition to ρ, each rule requires the state space s. However, this is static and will not be updated by a rule. Firstly, sequences are trivial:

$$
\begin{array}{lcl}
\mathcal{T}_s^c\ (stmt_1\ \texttt{;}\ stmt_2)\ \rho\ s & \leadsto & LET\ \rho' = \mathcal{T}_s^c\ (stmt_1)\ \rho\ s\ IN \\
 & & RETURN\ \mathcal{T}_s^c\ (stmt_2)\ \rho'\ s
\end{array}
$$

Here, the statement are sequentially translated and the second statement is given the environment, provided by the first statement, and the resulting environment from the second statement is returned.

Assignments in MINI-C have the simplest translations since there are no nesting or extra in-

puts/outputs:

$$
\begin{array}{lll}
\mathcal{T}^c_s\ (id\ \texttt{=}\ expr)\ \rho\ s & \leadsto & LET\ x = FRESH()\ IN \\
& & LET\ (p,i) = \rho_{inp}\ IN \\
& & LET\ (o,e) = \rho_{outp}\ IN \\
& & \underline{\texttt{box}\ x} \\
& & \quad \underline{\texttt{in(}\ head\ s\ \texttt{""}\ \texttt{)}\ \ \texttt{out(}\ head\ s\ \texttt{","}\ \texttt{)}} \\
& & \underline{\texttt{match}} \\
& & \quad \underline{\texttt{(}\ patt\ s\ \texttt{)}\ \texttt{->}\ \texttt{(}\ replace\ s\ id\ (\mathcal{T}^c_e\ expr)\ \texttt{);}} \\
& & IF\ \rho_{ctx}\ THEN\ \underline{\texttt{wire}\ p\ \texttt{(}\ i\ \texttt{)(}\ (wires\ x\ st\ \texttt{""})\ \texttt{,}\ patt\ e\ \texttt{);}} \\
& & LET\ \rho' = \rho[comp \mapsto IF\ \rho_{ctx}\ THEN\ \rho_{comp}\ ELSE\ (wires\ x\ st\ \texttt{""})]\ IN \\
& & LET\ \rho'' = \rho'[inp \mapsto (x,o),\ out \mapsto (wires\ x\ st\ \texttt{","}, [])]\ IN \\
& & RETURN\ \rho''[ctx \mapsto\ tt]
\end{array}
$$

The box is trivial to create. The pre-amble is created using the *head* rule, which creates the variable names with types. The pattern is created by the *patt* rule which writes all the variables, separated by comma. An assignment replaces the value of the given variable (l.h.s) with the translated expression of the assignment (r.h.s). Thus, the expression of the match uses the *replace* rule, which returns a similar to tuple to *patt*, with the exception that the given variable is replaced by the given expression.

The real problem with the translation is achieving a correct wiring. Due to the sequentiality imposed by MINI-C, a box cannot create the `wire` declaration of that box since the destination is unnown. Thus, only the wire declaration of the *previous* box can be defined. However, due to nesting (in the translated control statments) this is not always the case. Here, *ctx* (of ρ) decides if a the `wire` declaration is created or not. If *ctx* then it is declared, if it does not hold then the input of the box is added to *comp*, and the `wire` declaration has to be declared somewhere else (which is the nesting box, as we shall see below). The input and output of ρ is updated, such that the input is the box name and output of the previous box (which is the input of this box), while the output is the output of the box. Note, the extra element of out, which here is empty, is used for cases (e.g. output streams) with extra outputs. Finally, since this is not a box that introduces any nesting, the `wire` declaration for the box can be created by the next box – thus, the *ctx* holds.

The translation of an array assignments is similar. The only difference is that the built in Hume `update` function for vectors is used, and, since the first index of an array is 0, whilst it is 1 for a Hume vector, `1` is added to the index:

$$
\begin{array}{lll}
\mathcal{T}^c_s\ (id\texttt{[}\ expr_1\ \texttt{]}\ \texttt{=}\ expr_2)\ \rho\ s & \leadsto & LET\ x = FRESH()\ IN \\
& & LET\ (p,i) = \rho_{inp}\ IN \\
& & LET\ (o,e) = \rho_{outp}\ IN \\
& & \underline{\texttt{box}\ x} \\
& & \quad \underline{\texttt{in(}\ head\ s\ \texttt{""}\ \texttt{)}\ \ \texttt{out(}\ head\ s\ \texttt{","}\ \texttt{)}} \\
& & \underline{\texttt{match}} \\
& & \quad \underline{\texttt{(}\ patt\ s\ \texttt{)}\ \texttt{->}\ \texttt{(}\ replace\ s\ id\ (\texttt{update}\ id\ (\mathcal{T}^c_e\ expr_1)\ (\texttt{(}\ \mathcal{T}^c_e\ expr_1)\texttt{+1))}\ \texttt{);}} \\
& & IF\ \rho_{ctx}\ THEN\ \underline{\texttt{wire}\ p\ \texttt{(}\ i\ \texttt{)(}\ (wires\ x\ s\ \texttt{""})\ \texttt{,}\ patt\ e\ \texttt{);}} \\
& & LET\ \rho' = \rho[comp \mapsto IF\ \rho_{ctx}\ THEN\ \rho_{comp}\ ELSE\ (wires\ x\ s\ \texttt{""})]\ IN \\
& & LET\ \rho'' = \rho'[inp \mapsto (x,o),\ out \mapsto (wires\ x\ s\ \texttt{","}, [])]\ IN \\
& & RETURN\ \rho''[ctx \mapsto\ tt]
\end{array}
$$

Streams are slightly more complex than assignments. This is due to the fact that an extra wire is required to communicate with the stream. Now, the following rule translates the `printf` statement of MINI-C:

$$
\begin{array}{lll}
\mathcal{T}_s^c\,(\texttt{printf (\%d , } id \texttt{))}\ \rho\ s & \leadsto & \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ \mathit{str} = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ (p, i) = \rho_{\mathit{inp}}\ \mathit{IN} \\
& & \mathit{LET}\ (o, e) = \rho_{\mathit{outp}}\ \mathit{IN} \\
& & \underline{\texttt{stream}\ \mathit{str}\ \texttt{to "std\_out";}} \\
& & \underline{\texttt{box}\ x} \\
& & \quad \underline{\texttt{in(}\ \mathit{head}\ s\ \texttt{"")}} \\
& & \quad \underline{\texttt{out(}\ \mathit{head}\ s\ \texttt{"'" ,}\ \mathit{str} :: \mathcal{T}_T\ (\texttt{int}\ \mathit{str})\ \texttt{)}} \\
& & \underline{\texttt{match}} \\
& & \quad \underline{\texttt{(}\ \mathit{patt}\ s\ \texttt{) -> (}\mathit{patt}\ s\ \texttt{,}\ \mathit{id}\ \texttt{)}\ \texttt{;}} \\
& & \mathit{IF}\ \rho_{\mathit{ctx}}\ \mathit{THEN}\ \underline{\texttt{wire}\ p\ \texttt{(}\ i\ \texttt{)(}\ (\mathit{wires}\ x\ s\ \texttt{""})\ \texttt{,}\ \mathit{patt}\ e\ \texttt{);}} \\
& & \mathit{LET}\ \rho' = \rho[\mathit{comp} \mapsto \mathit{IF}\ \rho_{\mathit{ctx}}\ \mathit{THEN}\ \rho_{\mathit{comp}}\ \mathit{ELSE}\ (\mathit{wires}\ x\ s\ \texttt{""})]\ \mathit{IN} \\
& & \mathit{LET}\ \rho'' = \rho'[\mathit{inp} \mapsto (x, o),\ \mathit{out} \mapsto (\mathit{wires}\ x\ s\ \texttt{"'"}, [(s, \mathcal{T}_T\ (\texttt{int}\ \mathit{str})])]\ \mathit{IN} \\
& & \mathit{RETURN}\ \rho''[\mathit{ctx} \mapsto\ \mathit{tt}]
\end{array}
$$

Here, an extra (integer) output is added to the box, which is wired to the generated output stream (*str*). The expression of the box match is the same as the input, with the exception that an extra element which is the value to printed to the output. Note that the second element of *out* in ρ contains the stream information. This will achieve a correct `wire` declaration of the given box.

The rule to translate a MINI-C `scanf` statement is similar:

$$
\begin{array}{lll}
\mathcal{T}_s^c\,(\texttt{scanf (\%d , } id \texttt{))}\ \rho\ s & \leadsto & \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ \mathit{str} = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ (p, i) = \rho_{\mathit{inp}}\ \mathit{IN} \\
& & \mathit{LET}\ (o, e) = \rho_{\mathit{outp}}\ \mathit{IN} \\
& & \underline{\texttt{stream}\ \mathit{str}\ \texttt{from "std\_in";}} \\
& & \underline{\texttt{box}\ x} \\
& & \quad \underline{\texttt{in(}\ \mathit{head}\ s\ \texttt{"" ,}\ \mathit{str} :: \mathcal{T}_T\ (\texttt{int}\ \mathit{str})\ \texttt{)}} \\
& & \quad \underline{\texttt{out(}\ \mathit{head}\ s\ \texttt{"'"}\ \texttt{)}} \\
& & \underline{\texttt{match}} \\
& & \quad \underline{\texttt{(}\ \mathit{patt}\ s\ \texttt{,}\ \mathit{str}\ \texttt{) -> (}\mathit{replace}\ \ s\ \mathit{id}\ \ \mathit{str}\ \texttt{);}} \\
& & \mathit{IF}\ \rho_{\mathit{ctx}}\ \mathit{THEN}\ \underline{\texttt{wire}\ p\ \texttt{(}\ i\ \texttt{)(}\ (\mathit{wires}\ x\ s\ \texttt{""})\ \texttt{,}\ \mathit{patt}\ e\ \texttt{);}} \\
& & \mathit{LET}\ \rho' = \rho[\mathit{comp} \mapsto \mathit{IF}\ \rho_{\mathit{ctx}}\ \mathit{THEN}\ \rho_{\mathit{comp}}\ \mathit{ELSE}\ (\mathit{wires}\ x\ s\ \texttt{""})\ \texttt{,}\ \mathit{str}]\ \mathit{IN} \\
& & \mathit{LET}\ \rho'' = \rho'[\mathit{inp} \mapsto (x, (o\ \texttt{,}\ \mathit{str})),\ \mathit{out} \mapsto (\mathit{wires}\ x\ s\ \texttt{"'"}, [])]\ \mathit{IN} \\
& & \mathit{RETURN}\ \rho''[\mathit{ctx} \mapsto\ \mathit{tt}]
\end{array}
$$

The second element of *out* in ρ is now empty, since the generated stream is an input and not an output stream. The input is only used for one purpose (generate the `wire` declaration). Thus, it is sufficient to integrate it with *inp* of ρ. *out* of ρ , on the other hand, is used both to generate `wire` declaration and as input to the `wire` declaration of the next box, which should not contain the stream.

The loops and conditionals are more complex since they involve nesting, and translations of the bodies of the statements. Firstly, the following rule translates a MINI-C `for` statement:

$\mathcal{T}_s^c$ (`for ( `$stmt_1$` ; `$expr$` ; `$stmt_2$` ) `$stmt_3$) $\rho\ s \rightsquigarrow$

LET $\rho' = \mathcal{T}_s^c\ (stmt_1)\ \rho\ s$ *IN*
LET $x =$ *FRESH()* *IN*
LET $(p, i) = \rho_{inp}$ *IN*
LET $(o, e) = \rho_{outp}$ *IN*
LET $ex = (\mathcal{T}_e^c\ expr)$ *IF* *isNumeric* $expr$ *THEN* `!= 0` *IN*
`box` x
  `in(` *head* s `"" , head` s `"','" )`
  `out(` *head* s `"'" , head` s `"'''" )`
`match`
  `(` $patt\ s$ `,` $ignore\ s$ `) -> if` ex
                `then (` $patt\ s$ `,` $ignore\ s$ `)`
                `else (` $ignore\ s$ `,` $patt\ s$ `)`
  `|(` $ignore\ s$ `,` $patt\ s$ `) -> if` ex
                `then (` $patt\ s$ `,` $ignore\ s$ `)`
                `else (` $ignore\ s$ `,` $patt\ s$ `);`
LET $(ix_1, ix_2) = (wires\ x\ s\ \text{""}, wires\ x\ s\ \text{"'"})$ *IN*
LET $(ox_1, ox_2) = (wires\ x\ s\ \text{"'"}, wires\ x\ s\ \text{"'''"})$ *IN*
IF ρ_{ctx} *THEN* `wire` p `(` i `)(` ix_1 `,` $patt\ e$ `);`
LET $\rho_1 = \rho'[inp \mapsto (x, o),\ out \mapsto (o_2, []),\ ctx \mapsto ff]$ *IN*
LET $\rho_2 = \mathcal{T}_s^c\ (stmt_3\ ;\ stmt_2)\ \rho_1\ s$ *IN*
LET $(p_1, i_1) = \rho_{2_{inp}}$ *IN*
LET $(o_1, e_1) = \rho_{2_{out}}$ *IN*
LET $cmp = head\ \rho_{2_{comp}}$ *IN*
`wire` p_1 `(` i_1 `)(` i_2 `,` e_1 `);`
LET $\rho_3 = \rho_2[inp \mapsto (x, (o\ ,\ o_1)),\ out \mapsto (ox_1, cmp),\ ctx \mapsto tt]$ *IN*
RETURN $\rho_3[comp\ \mapsto (tail\ \rho_{3_{comp}})]$

The first statement ($stmt_1$) is first translated. The "looping box" is then created. Since, it may initially fail, the input is either from the previous box ($stmt_1$) or the body, thus two matches with different patterns and identical expressions are created. The *ignore* rule prints a sequence of `*`s. In the expression the condition is tested. If it holds the values are sent to the box body, if it fails, the "loop terminates", and the values are sent to the next box. Now, MINI-C does not have any booleans, and false/true is represented as 0 and not 0. In $\mathcal{T}_o$, a MINI-C operator is directly translated into the corresponding Hume operator. In Hume, the condition expressions in Hume `if` statements must be of boolean types, which is returned by the `==`,`!=`, `<` and `<=` operators. However, in MINI-C all operators return a integer type, and this is also the case for the remaining Hume operators. Thus, in the meta language we add a test to see if the operator is of numeric type (by the *isNumeric* function). If it i, `!=0` is added to the conditional, which turns the integer type into a boolean.

As in the above rules, the `wire` declaration for the previous box is created if `ctx` holds, and the input of the box is added to `comp` if not. The body of the loop is then created, and ρ is augmented with the values of the "looping box" x. To translated the body, *ctx* is set to *ff* (false) since the source of the first box in the body is x, and all wire values are not available to create the `wire` declaration for x. After the translation, the first element of *comp* is removed, which will be the first box of the body. This is used to create a correct `wire` declaration for x (which has to be done by the following box in the translation). The `wire` declaration of the last box of the body is created, before the environment is updated and returned.

The `if`-statement, without an `else`-clause is translated as follows:

$\mathcal{T}_s^c$ (`if (` *expr* `)` *stmt*) $\rho\ s \rightsquigarrow$

LET $f =$ *FRESH() IN*
LET $j =$ *FRESH() IN*
LET $(p, i) = \rho_{inp}$ *IN*
LET $(o, e) = \rho_{outp}$ *IN*
LET $ex = (\mathcal{T}_e^c\ expr)$ *IF isNumeric expr THEN* `!= 0` *IN*
`box` f
  `in(` *head s* `"" )`
  `out(` *head s* `","` , *head s* `""," )`
`match`
  `(` *patt s* `) -> if` *ex*
    `then (` *patt s* , *ignore s* `)`
    `else (` *ignore s* , *patt s* `);`
`box` j
  `in(` *head s* `"" ,` *head s* `"," )`
  `out(` *head s* `""," )`
`match`
  `(` *patt s* , *ignore s* `) -> (` *patt s* `)`
 `|(` *patt s* , *ignore s* `) -> (` *patt s* `);`
LET $(fi, fo_1, fo_2) = (wires\ f\ s\ \text{""}, wires\ f\ s\ \text{","}, wires\ f\ s\ \text{""},")$ *IN*
LET $(ji_1, ji_2, jo) = (wires\ j\ s\ \text{""}, wires\ j\ s\ \text{","}, wires\ j\ s\ \text{""},")$ *IN*
IF ρ_{ctx} *THEN* `wire` p `(` i `)(` ji , *patt e* `);`
LET $\rho_1 = \rho[inp \mapsto (f, o),\ out \mapsto (fo_1, []),\ ctx \mapsto ff]$ *IN*
LET $\rho_2 = \mathcal{T}_s^c\ (stmt)\ \rho_1[comp \mapsto (IF\ \rho_{ctx}\ THEN\ \rho_{comp}\ ELSE\ (fi :: \rho_{comp}))]\ s$ *IN*
LET $(p_1, i_1) = \rho_{2_{inp}}$ *IN*
LET $(o_1, e_1) = \rho_{2_{out}}$ *IN*
LET $cmp = head\ \rho_{2_{comp}}$ *IN*
`wire` p_1 `(` i_1 `)(` ji_1 , e_1 `);`
`wire` f `(` o `)(` $comp$, ji_2 `);`
LET $\rho_3 = \rho_2[inp \mapsto (j, (o_1\ ,\ fo_2)),\ out \mapsto (jo, []),\ ctx \mapsto tt]$ *IN*
RETURN $\rho_3[comp\ \mapsto (tail\ \rho_{3_{comp}})]$

The translation is similar to the `for` statement. However, the "looping box" x is now replaced by two boxes: a fork box f; and a join box j. If the condition holds, then the values are sent to the body of the MINI-C `if` statement. If it fails, it is sent directly to the join box. The body of the translation is similar.

The translation rule for a MINI-C `if`-statement with an `else` clause only deviates by also translating the body of the `else` branch, and wiring the values in a similar fashion as with the other branch:

$\mathcal{T}_s^c$ (`if ( expr ) stmt1 else stmt2`) $\rho\ s \rightsquigarrow$

LET $f =$ *FRESH()* *IN*
LET $j =$ *FRESH()* *IN*
LET $(p, i) = \rho_{inp}$ *IN*
LET $(o, e) = \rho_{outp}$ *IN*
LET $ex = (\mathcal{T}_e^c\ expr)$ *IF* *isNumeric expr* *THEN* `!= 0` *IN*
`box` f
  `in(` *head s* `"" )`
  `out(` *head s* `"'" ,` *head s* `"'"" )`
`match`
  `(` *patt s* `) -> if` ex
    `then (` *patt s* `,` *ignore s* `)`
    `else (` *ignore s* `,` *patt s* `);`
`box` j
  `in(` *head s* `"" ,` *head s* `"'" )`
  `out(` *head s* `"'"" )`
`match`
  `(` *patt s* `,` *ignore s* `) -> (` *patt s* `)`
 `|(` *patt s* `,` *ignore s* `) -> (` *patt s* `);`
LET $(fi, fo_1, fo_2) =$ (*wires f s* `""`, *wires f s* `"'"`, *wires f s* `"'""`) *IN*
LET $(ji_1, ji_2, jo) =$ (*wires j s* `""`, *wires j s* `"'"`, *wires j s* `"'""`) *IN*
IF ρ_{ctx} *THEN* `wire` p `(` i `)(` ji `,` *patt e* `);`
LET $\rho_1 = \rho[inp \mapsto (f, o),\ out \mapsto (fo_1, []),\ ctx \mapsto ff]$ *IN*
LET $\rho_2 = \mathcal{T}_s^c\ (stmt_1)\ \rho_1[comp \mapsto (IF\ \rho_{ctx}\ THEN\ \rho_{comp}\ ELSE\ (fi :: \rho_{comp}))]\ s$ *IN*
LET $(p_1, i_1) = \rho_{2\,inp}$ *IN*
LET $(o_1, e_1) = \rho_{2\,out}$ *IN*
LET $cmp_1 = head\ \rho_{2\,comp}$ *IN*
`wire` p_1 `(` i_1 `)(` ji_1 `,` e_1 `);`
LET $\rho_3 = \rho_2[inp \mapsto (f, o),\ out \mapsto (fo_2, []),\ ctx \mapsto ff]$ *IN*
LET $\rho_4 = \mathcal{T}_s^c\ (stmt_2)\ \rho_3[comp \mapsto (tail\ \rho_{3\,comp})]\ s$ *IN*
LET $(p_2, i_2) = \rho_{4\,inp}$ *IN*
LET $(o_2, e_2) = \rho_{4\,out}$ *IN*
LET $cmp_2 = head\ \rho_{4\,comp}$ *IN*
`wire` p_2 `(` i_3 `)(` ji_2 `,` e_2 `);`
`wire` f `(` o `)(` $comp_1$ `,` $comp_2$ `);`
LET $\rho_5 = \rho_4[inp \mapsto (j, (o_1\ ,\ o_2)),\ out \mapsto (jo, []),\ ctx \mapsto tt]$ *IN*
RETURN $\rho_5[comp\ \mapsto (tail\ \rho_{5\,comp})]$

5.4 The Full Program

The full MINI-C *program* is translated into "coordination Hume" by $\mathcal{T}^c$:

$$
\begin{array}{lll}
\mathcal{T}^c\ (\mathit{prelude}\ \texttt{main()\{}\mathit{decls}\ \mathit{stmts}\ \texttt{\}}) & \leadsto & \mathit{LET}\ s = \mathit{statespace}\ \mathit{decls}\ \mathit{IN} \\
& & \mathit{LET}\ f = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ l = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ y = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ z_1 = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ z_2 = \mathit{FRESH()}\ \mathit{IN} \\
& & \underline{\texttt{box}\ f} \\
& & \quad \underline{\texttt{in(}\ x\ \texttt{::bool)}\ \ \texttt{out(}\ \mathit{head}\ s\ \texttt{""}\ \texttt{,}\ y\ \texttt{::bool)}} \\
& & \underline{\texttt{match}} \\
& & \quad \underline{\texttt{(\_) -> (}\mathit{init\_values}\ s\ \texttt{, *) ;}} \\
& & \underline{\texttt{box}\ l} \\
& & \quad \underline{\texttt{in(}\ \mathit{head}\ s\ \texttt{""}\ \texttt{,}\ z_1\ \texttt{::bool)}\ \ \texttt{out(}\ z_2\ \texttt{::bool)}} \\
& & \underline{\texttt{match}} \\
& & \quad \underline{\texttt{(}\ \mathit{patt}\ s\ \texttt{, *) -> (*) ;}} \\
& & \mathit{LET}\ (\mathit{fi}, \mathit{fo}) = (\mathit{wires}\ f\ s\ \texttt{""}, \mathit{wires}\ f\ s\ \texttt{","})\ \mathit{IN} \\
& & \mathit{LET}\ (\mathit{li}, \mathit{lo}) = (\mathit{wires}\ l\ s\ \texttt{""}, \mathit{wires}\ l\ s\ \texttt{","})\ \mathit{IN} \\
& & \mathit{LET}\ \rho = \rho_{\mathit{empty}}[\mathit{ins} \mapsto (f, (\mathit{fi}\ \texttt{,}\ f.x\ \texttt{initially true})]\ \mathit{IN} \\
& & \mathit{LET}\ \rho' = \mathcal{T}^c_s\ (\mathit{stmts})\ \rho[\mathit{out} \mapsto (\mathit{fo}, [])]\ s\ \mathit{IN} \\
& & \mathit{LET}\ (p, i) = \rho_{\mathit{inp}}\ \mathit{IN} \\
& & \mathit{LET}\ (o, e) = \rho_{\mathit{outp}}\ \mathit{IN} \\
& & \underline{\texttt{wire}\ p\ \texttt{(}\ i\ \texttt{)(}\ (\mathit{wires}\ l\ s\ \texttt{""})\ \texttt{,}\ \mathit{patt}\ e\ \texttt{);}} \\
& & \underline{\texttt{wire}\ l\ \texttt{(}\ \rho'_{\mathit{out}}\ \texttt{,}\ l.z_2\ \texttt{)(}\ l.z_1\ \texttt{);}}
\end{array}
$$

It creates a first box f which initialises the execution, and a last box l that terminates execution. This initialisation and termination could have been incorporated into $\mathcal{T}^c_s$, but this would have complicated the rules.

Now, the state space list is first created by the *statespace* function, by using the declarations (*decls*). To achieve a total wiring the f box needs a feedback loop to start the execution. It creates the dummy `0` values for all the state-space variables. The last box l consumes the input and writes nothing to the output. Since Hume does not terminate this will result in an infinite sequence of stuttering steps.

The environment is initialised and given the correct wires for box f. $\mathcal{T}^c_s$ is then applied, which translates the statements of the souce MINI-C program. This will also create the wiring for f since *ctx* holds for an empty environment. Finally, the wires for the last statement-box and the last box l is created. Note, *ctx* will always hold after the last statement, since it cannot be nested inside an `if` or `for` statement.

6 Translating MINI-C into Expression Layer Hume Version

6.1 Overview

While the translated coordination layer version of MINI-C program is represented as "sequence of boxes", the translated expression layer version is represented as a sequence of functional calls. Now, a pure expression version of the MINI-C program would not include any boxes. In Hume this can be executed by the `expression` statement. However, this disable the support for streams. Thus, the expression layer consist of one box, with input streamsas input and output streams as output. The state space is wired

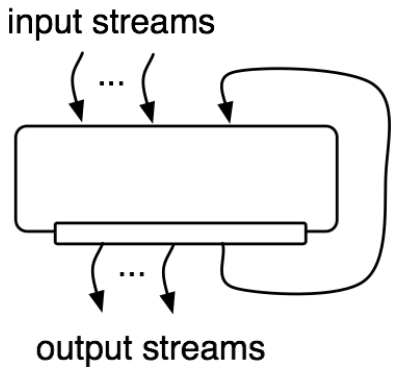


Figure 5: Box representation of expression layer translation

as feedback loops, since wiring must be total – thus, since the box should only execute once, no values are actually sent to these wires. Figure 5 show the box diagram of the MINI-C program translated into the expression-layer version of Hume.

6.1.1 Checking Stream Ordering

Hume handles streams in the coordination layer. Hence, although streams are still supported in the expression layer version shown in Figure 5, two additional requirements are imposed on the MINI-C program:

- a `scanf` statement can only be preceded by a `scanf` statement:

  $$cond_1 \equiv \neg(A \texttt{ ; scanf("\%d",\&}B\texttt{)}) \textit{ where } A \neq \texttt{scanf("\%d",\&}C\texttt{)}$$

  Consequently, all `streamf` statements must be before all non-`streamf` statements.

- a `printf` statement can only be followed by a `printf` statement:

  $$cond_2 \equiv \neg(\texttt{printf("\%d",}A\texttt{)} \texttt{ ; } B) \textit{ where } B \neq \texttt{printf("\%d",}C\texttt{)}$$

  Consequently, all `printf` statements must be after all non-`printf` statements.

The expression layer version is created if, and only if, $cond_1$ and $cond_2$ holds. This is assumed in the following.

6.2 Functions on the state and higher order functions

The expression layer version is the target of the Hume costing. This is achieved by translating Hume code into an intermediate representation using `phamc` and then apply the `art3` tools. The details of these processes are described in the relevant deliverables. Thus, many of the choices in this section are based on requirements from `phamcs` and `art3`. Note that `phamc` requires typing definitions for all functions, which the rules does not reflect.

6.2.1 Boolean type

`art3` doesn't work well with the built in boolean type. Thus, a special boolean type is enumerated:

$$\mathcal{T}^e_{mybool} \rightsquigarrow \underline{\texttt{data mybool = TT | FF;}}$$

6.2.2 `store_` and `load_`

For each variable a `store`, which updates the variable in the state space, and `load` function, which projects the value from the state space, is defined. The naming scheme for these functions is `load`/`store`, followed by the variable name with a capital first letter. For example, a MINI-C program consisting of three variables: `x`, `y` and `z`, generates the following functions:

```
loadX (x,y,z) = x;
loadY (x,y,z) = y;
loadZ (x,y,z) = z;

storeX e (x,y,z) = (e,y,z);
storeY e (x,y,z) = (x,e,z);
storeZ e (x,y,z) = (x,y,e);
```

Let, the function *firstUpper* return the variable name, with a capital first letter. Then, $\mathcal{T}^e_{loadfun}$ generates the `load` functions, while $\mathcal{T}^e_{storefun}$ generates the `store` functions:

$$
\begin{array}{lcl}
\mathcal{T}^e_{load}\ (id) & \leadsto & \underline{\texttt{load}}(\textit{firstUpper id}) \\
\mathcal{T}^e_{store}\ (id) & \leadsto & \underline{\texttt{store}}(\textit{firstUpper id})
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}^e_{loadfun}\ ((x,t):[])\ st & \leadsto & \mathcal{T}^e_{load}\ (x)\ \underline{\texttt{(}\ \textit{patt st}\ \texttt{) = }x\texttt{;}} \\
\mathcal{T}^e_{loadfun}\ ((x,t):xs)\ st & \leadsto & \mathcal{T}^e_{load}\ (x)\ \underline{\texttt{(}\ \textit{patt st}\ \texttt{) = }x\texttt{;}} \\
 & & \mathcal{T}^e_{loadfun}\ xs\ st
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}^e_{storefun}\ ((x,t):[])\ st & \leadsto & \mathcal{T}^e_{store}\ (x)\ \underline{e\texttt{(}\ \textit{patt st}\ \texttt{) = (}\ \textit{replace st x}\ \texttt{e);}} \\
\mathcal{T}^e_{storefun}\ ((x,t):xs)\ st & \leadsto & \mathcal{T}^e_{store}\ (x)\ \underline{e\texttt{(}\ \textit{patt st}\ \texttt{) = (}\ \textit{replace st x}\ \texttt{e);}} \\
 & & \mathcal{T}^e_{storefun}\ xs\ st
\end{array}
$$

Both $\mathcal{T}^e_{load}$ and $\mathcal{T}^e_{store}$ iterates through the state space (list). In addition the full state-space list is required. Thus, both arguments of the rules are the state space. The recursion is on first list, while the second is left unchanged.

6.2.3 Higher order functions

The expression layer version is based around some pre-defined higher-order functions (HOFs). These are identical for each program. The HOFs are used to abstract the three control structures found in MINI-C:

$$
\begin{array}{lcl}
\mathcal{T}^e_{hof} & \leadsto & \underline{\texttt{exIf TT s st = s st;}} \\
 & & \underline{\texttt{exIf FF \_ st = \ \ st;}} \\
 & & \underline{\texttt{exIfElse TT s \_ st = s st;}} \\
 & & \underline{\texttt{exIfElse FF \_ t st = t st;}} \\
 & & \underline{\texttt{loop e s st = if (e st) == TT then (loop e s (s st)) else st;}}
\end{array}
$$

`exIf` accepts an expression (the condition) a statement and the state space. The condition is of `mybool` type, and the function is defined by pattern matching. This is also the case for `exIfElse`, which also contains an `else` clause, represented as `t`. The statement has type `statespace -> statespace`, where `statespace` is a tuple of all the variables.

`loop` is a tail-recursive function, which expresses the repetition part of a `for` loop. Due to the repetition, the condition `e` is a function of type `statespace -> mybool`, since the value of `e` may change between recursive calls.

Note, that the generic `loop` functions is included for completeness. It is too generic for the costing. Only a subset of the `for`-statements can be costed, and they do not apply this HOF.

6.3 Expressions

In the HOF approach expressions are dependent on the state – thus they are functions on the state space. Hume does not support function unnamed functions, like e.g. `fn` in ML, thus all functions must be properly declared. Now, all function uses the name `st` on the state space argument. Thus, it is only looping expressions that requires a function to be defined. $\mathcal{T}^e_{e'}$ defines the translation rules for expressions, while $\mathcal{T}^e_e$ defines the expression as a function:

$$
\begin{array}{lcl}
\mathcal{T}^e_{e'}\ (int) & \leadsto & \underline{int} \\
\mathcal{T}^e_{e'}\ (id) & \leadsto & \underline{\texttt{load}(firstUpper\ id)\ \texttt{st}} \\
\mathcal{T}^e_{e'}\ (id\texttt{[}expr\texttt{]}) & \leadsto & \underline{\texttt{(load}(firstUpper\ id)\ \texttt{st)@(}\ \mathcal{T}^e_{e'}\ (expr)\ \texttt{+1)}} \\
\mathcal{T}^e_{e'}\ (expr_1\ binop\ expr_2) & \leadsto & \mathcal{T}^e_{e'}\ (expr_1)\ \ \mathcal{T}_o\ (binop)\ \ \mathcal{T}^e_{e'}\ (expr_2) \\
\mathcal{T}^e_{e'}\ (\ \texttt{(}expr_1\texttt{)}\ ) & \leadsto & \underline{\texttt{(}}\ \mathcal{T}^e_{e'}\ (expr_2\ )\ \underline{\texttt{)}}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}^e_e\ (expr) & \leadsto & LET\ x = FRESH()\ IN \\
 & & \underline{x\ \texttt{st =}}\ \mathcal{T}^e_{e'}\ (expr)\ \underline{\texttt{;}} \\
 & & RETURN\ x
\end{array}
$$

6.3.1 `mybool` expressions

The following translation rules turns a MINI-C expression into a "`mybool` predicate". Since MINI-C doesn't contain boolean values (`FF` is `0`, and `TT` is not `0`), the cases where a MINI-C expression is a Hume integer must be turned into a Hume booleas, i.e. `!= 0`. $\mathcal{T}^e_{c'}$ creates a Hume expression, while $\mathcal{T}^e_c$ turns such an expression into a function:

$$
\begin{array}{lcl}
\mathcal{T}^e_{c'}\ (expr) & \leadsto & \underline{\texttt{if (}}\ isNumeric\ expr \\
 & & \quad THEN\ \mathcal{T}^e_{e'}\ (expr)\ \underline{\texttt{!= 0) then TT else FF}} \\
 & & \quad ELSE\ \mathcal{T}^e_{e'}\ (expr)\ \underline{\texttt{) then TT else FF}} \\
\mathcal{T}^e_c\ (expr) & \leadsto & LET\ x = FRESH()\ IN \\
 & & \underline{x\ \texttt{st =}}\ \mathcal{T}^e_{c'}\ (expr)\ \underline{\texttt{;}} \\
 & & RETURN\ x
\end{array}
$$

6.4 Statements

To translate a MINI-C stream, an meta-environment is required. The translation rules for a statement $\mathcal{T}^e_s$ returns a pair $\langle x, \rho \rangle$, consisting of the function identifier of the generated function, and the new environment ρ. An environment holds two mappings: the state-space *st*; and the output streams *ostr*. An empty environment ρ_{empty} has the following properties:

$$
\begin{array}{lcl}
\rho_{empty_{st}} & = & [] \\
\rho_{empty_{ostr}} & = & []
\end{array}
$$

The rules for assignments and sequence are straightforward:

$$\begin{array}{lll}
\mathcal{T}^e_s\ (id\ \texttt{=}\ expr)\ \rho & \leadsto & LET\ x = FRESH()\ IN \\
 & & \underline{x\ \texttt{st =}}\ \mathcal{T}^e_{store}(id)\ \ \underline{\texttt{(}}\mathcal{T}^e_{e'}\ (expr)\ \underline{\texttt{) st;}} \\
 & & RETURN\ \langle x, \rho\rangle \\
\mathcal{T}^e_s\ (id\texttt{[}\ expr_1\ \texttt{]}\ \texttt{=}\ expr_2)\ \rho & \leadsto & LET\ x = FRESH()\ IN \\
 & & \underline{x\ \texttt{st =}}\ \mathcal{T}^e_{store}(id)\ \ \underline{\texttt{(update}}\ \mathcal{T}^e_{store}(id)\ \ \mathcal{T}^e_{e'}\ (expr_1)\ \ \mathcal{T}^e_{e'}\ (expr_2)\ \underline{\texttt{) st;}} \\
 & & RETURN\ \langle x, \rho\rangle \\
\mathcal{T}^e_s\ (stmt_1\ \texttt{;}\ stmt_2)\ \rho & \leadsto & LET\ \langle f, \rho'\rangle = \mathcal{T}^e_s\ (stmt_1)\ \rho\ IN \\
 & & LET\ \langle g, \rho''\rangle = \mathcal{T}^e_s\ (stmt_2)\ \rho'\ IN \\
 & & RETURN\ \langle g(f), \rho''\rangle
\end{array}$$

Note, that the environment is only used/changed by streams. Thus, with the exception of sequences, which can contain streams, all remaining statements leave the environment unchanged. Now, the translation rules for the MINI-C `if` statements uses the higher order functions:

$$\begin{array}{lll}
\mathcal{T}^e_s\ (\texttt{if (}\ expr\ \texttt{)}\ stmt)\ \rho & \leadsto & LET\ x = FRESH()\ IN \\
 & & LET\ \langle s, \rho\rangle = \mathcal{T}^e_s\ (stmt)\ \rho\ IN \\
 & & \underline{x\ \texttt{st = exIf}}\ \mathcal{T}^e_{c'}\ (expr)\ \underline{s\ \texttt{st;}} \\
 & & RETURN\ \langle x, \rho\rangle \\
\mathcal{T}^e_s\ (\texttt{if (}\ expr\ \texttt{)}\ stmt_1\ \texttt{else}\ stmt_2)\ \rho & \leadsto & LET\ x = FRESH()\ IN \\
 & & LET\ \langle s, \rho\rangle = \mathcal{T}^e_s\ (stmt_1)\ \rho\ IN \\
 & & LET\ \langle t, \rho\rangle = \mathcal{T}^e_s\ (stmt_2)\ \rho\ IN \\
 & & \underline{x\ \texttt{st = exIfElse}}\ \mathcal{T}^e_{c'}\ (expr)\ \underline{s\ t\ \texttt{st;}} \\
 & & RETURN\ \langle x, \rho\rangle
\end{array}$$

Here, $\mathcal{T}^e_{c'}$ ensures that a correct `mybool` type is returned, regardless of whether or not a numerical MINI-C type is used. There are two versions of the translation rule for MINI-C `for`-statements: one for the costable subset and one for the generic case. We will first give the rule for the costable case:

$$\begin{array}{l}
\mathcal{T}^e_s\ (\texttt{for (}\ id\ \texttt{=}\ expr_1\texttt{;}\ expr_2\ binop\ id\ \texttt{;}\ id\ \texttt{=}\ id\ \texttt{-}\ expr_3\texttt{)}\ stmt_3)\ \rho\ \leadsto \\
\quad LET\ x = FRESH()\ IN \\
\quad LET\ y = FRESH()\ IN \\
\quad LET\ \langle f, \rho\rangle = \mathcal{T}^e_s\ (id\ \texttt{=}\ expr_1)\ \rho\ IN \\
\quad LET\ \langle s, \rho\rangle = \mathcal{T}^e_s\ (stmt_3\ \texttt{;}\ id\ \texttt{=}\ id\ \texttt{-}\ expr_3)\ \rho\ IN \\
\quad \underline{y\ \texttt{st}\ id\ \texttt{=}\ \texttt{if}}\ \mathcal{T}^e_{e'}\ (expr_2)\ \mathcal{T}_o\ (binop)\ \underline{id\ \texttt{then}\ y\ \texttt{(}\ s\ \texttt{st)}\ \ \texttt{(}id\ \texttt{-}}\ \mathcal{T}^e_{e'}\ (expr_3)\ \underline{\texttt{) else st;}} \\
\quad \underline{x\ \texttt{st =}\ y\ \texttt{st}}\ \mathcal{T}^e_{e'}\ (expr_1)\ \underline{\texttt{;}} \\
\quad RETURN\ \langle x(f), \rho\rangle
\end{array}$$

In addition, the rule requires that *binop* is either `<` or `<=`, and that $stmt_3$ does not contain a statement of the form id `=` $\cdots$. In the rule, the y function introduces an accumulator variable which is always reduced towards a lower bound (assumed to be greater or equal to `0`). This reduction is used by the costing.

The generic case uses the higher-order `loop` function:

$$\begin{array}{lll}
\mathcal{T}^e_s\ (\texttt{for (}\ stmt_1\ \texttt{;}\ expr\ \texttt{;}\ stmt_2\ \texttt{)}\ stmt_3)\ \rho & \leadsto & LET\ x = FRESH()\ IN \\
 & & LET\ \langle f, \rho\rangle = \mathcal{T}^e_s\ (stmt_1)\ \rho\ IN \\
 & & LET\ e = \mathcal{T}^e_c\ (expr)\ IN \\
 & & LET\ \langle s, \rho\rangle = \mathcal{T}^e_s\ (stmt_3\ \texttt{;}\ stmt_2)\ \rho\ IN \\
 & & \underline{x\ \texttt{st = loop}\ e\ s\ \texttt{st;}} \\
 & & RETURN\ \langle x(f), \rho\rangle
\end{array}$$

An output stream (`printf` statement in MINI-C) adds another element (wire) to the state space. Further, the stream that this output is wired to is created, and the environment (*ostr*) is updated with this extra stream. This is required to achieve the correct wiring and box declaration when the main program box is declared. Moreover, since there can be several output streams, the extra element must be added to the state space (*st*):

$$\begin{array}{rcl}
\mathcal{T}^e_s\ (\texttt{printf (\%d , } id \texttt{))}\ \rho & \leadsto & \mathit{LET}\ s = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
& & \rho' = \rho[\mathit{ostr} \mapsto \rho_{\mathit{ostr}}\ @\ [(s, \texttt{int 16})],\ \mathit{st} \mapsto \rho_{\mathit{st}}\ @\ [(s, \texttt{int 16})]] \\
& & \underline{\texttt{stream } s \texttt{ to "std_out";}} \\
& & \underline{x\ \texttt{(}\ \mathit{patt}\ \rho_{\mathit{st}}\ \texttt{) = (}\ \mathit{patt}\ \rho_{\mathit{st}}\ \texttt{, } \mathit{id}\ \texttt{);}} \\
& & \mathit{RETURN}\ \langle x, \rho' \rangle
\end{array}$$

An input stream (`scanf` statement in MINI-C) removes one (the last) element from the state space, and updates the correct variable with this value. However, this requires a pre-processing of the streams before the translation. Hence, we assume that the state space (*st*) is already augmented with all the input streams. The last element is thus removed from the state space (`st`), and a input stream is created with this name. The *replace* function is used to replace the correct element with the value of the last element. Note that the new state space is used in the expression of the declared function x below, which removes one element from the state space:

$$\begin{array}{rcl}
\mathcal{T}^e_s\ (\texttt{scanf (\%d , \&}\mathit{id}\ \texttt{))}\ \rho & \leadsto & \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}(s,t) = \mathit{head}(\mathit{rev}\ \rho_{\mathit{st}})\ \mathit{IN} \\
& & \rho' = \rho[\mathit{st} \mapsto \mathit{rev}(\mathit{tail}(\mathit{rev}\ \rho_{\mathit{st}}))] \\
& & \underline{\texttt{stream } s \texttt{ from "std_in";}} \\
& & \underline{x\ \texttt{(}\ \mathit{patt}\ \rho_{\mathit{st}}\ \texttt{) = (}\ \mathit{replace}\ \mathit{id}\ s\ \rho'_{\mathit{st}}\ \texttt{)}} \\
& & \mathit{RETURN}\ \langle x, \rho' \rangle
\end{array}$$

6.5 The Full Program

Finally, $\mathcal{T}^e$ translated a full MINI-C program into Hume:

$$\begin{array}{rcl}
\mathcal{T}^e\ (\mathit{prelude}\ \texttt{main()\{}\mathit{decls}\ \mathit{stmts}\ \texttt{\}}) & \leadsto & \mathit{LET}\ s = \mathit{statespace}\ \mathit{decls}\ \mathit{IN} \\
& & \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
& & \mathit{LET}\ \mathit{istrs} = \mathit{istreams}\ \mathit{stmts}\ \mathit{IN} \\
& & \mathit{LET}\ \rho = \rho_{\mathit{empty}}[\mathit{st} \mapsto s\ @\ \mathit{istrs}]\ \mathit{IN} \\
& & \mathcal{T}^e_{\mathit{mybool}} \\
& & \mathcal{T}^e_{\mathit{loadfun}}\ s\ s \\
& & \mathcal{T}^e_{\mathit{storefun}}\ s\ s \\
& & \mathcal{T}^e_{\mathit{hof}} \\
& & \mathit{LET}\ \langle e, \rho' \rangle = \mathcal{T}^e_s\ (\mathit{stmts})\ \rho\ \mathit{IN} \\
& & \underline{x\ \texttt{(}\ \mathit{patt}\ \rho'_{\mathit{st}}\ \texttt{) = (}\ \mathit{ignore}\ s\ \texttt{,}\ \mathit{patt}\ \rho'_{\texttt{ostr}}\ \texttt{);}} \\
& & \underline{\texttt{box}\ \mathit{program}} \\
& & \quad \underline{\texttt{in(}\ \mathit{head}\ s\ \texttt{""}\ \texttt{,}\ \mathit{head}\ \mathit{istrs}\ \texttt{""}\ \texttt{)}} \\
& & \quad \underline{\texttt{out(}\ \mathit{head}\ s\ \texttt{","}\ \texttt{,}\ \mathit{head}\ \rho'_{\mathit{ostr}}\ \texttt{""}\ \texttt{)} \\
& & \underline{\texttt{match}} \\
& & \quad \underline{\texttt{(}\ \mathit{patt}\ s\ \texttt{,}\ \mathit{patt}\ \mathit{istrs}\ \texttt{)}\ \texttt{->}\ x\ \texttt{(}\ e\ (\mathit{patt}\ s\ \texttt{,}\ \mathit{patt}\ \mathit{istrs}\ \texttt{));}} \\
& & \\
& & \underline{\texttt{wire}\ \mathit{program}} \\
& & \quad \underline{\texttt{(}\ \mathit{init_wires}\ \mathit{program}\ s\ \texttt{","}\ \texttt{,}\ \mathit{patt}\ \mathit{istrs}\ \texttt{)}} \\
& & \quad \underline{\texttt{(}\ \mathit{wires}\ \mathit{program}\ s\ \texttt{""}\ \texttt{,}\ \mathit{patt}\ \rho'_{\mathit{ostr}}\ \texttt{);}}
\end{array}$$

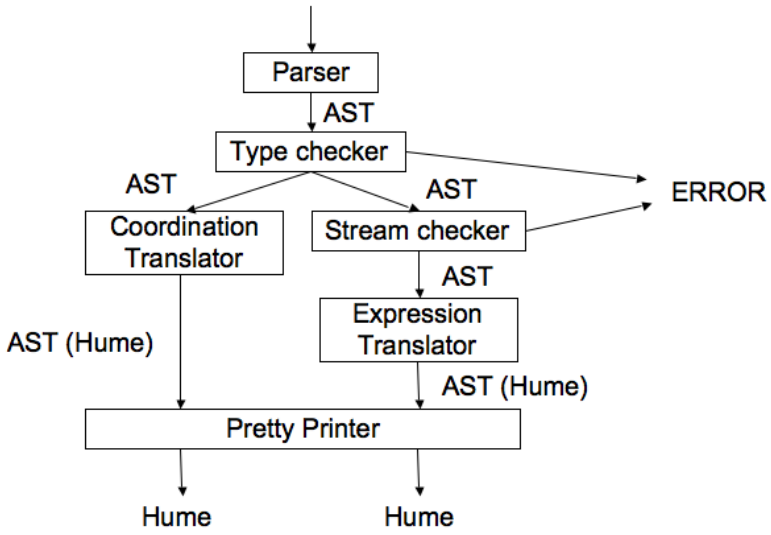


Figure 6: System Diagram of Translator Program.

Firstly, the state space is derived from the declarations in the MINI-C program. The initial environment ρ is created, where the input streams, obtained by *istream*, is added to the state space – which is derived from the declarations. Then, the `store`/`load` and HOF functions are generated. This is followed by a call to $\mathcal{T}_s^e$, which creates all the functions and returns the expression holding the correct order of function application of the generated functions.

The x function above ensures that the program is only executed once, by only sending output to the output streams. The box `input` is the state space and the input streams while the `output` is the state space and the output streams. This is also reflected in the `wire` declaration. Note that *init_wires* give all the state-space variables dummy `0` values.

7 The Implementation

Both the coordination- and expression-layer rules are implemented in a Haskell program. The system diagram for this implemented is shown in Figure 6. The MINI-C program is first parsed, and this front-end was created using Alex and Happy. The generated abstract syntax three (AST) is then type checked. After that the program branches out into two streams: one translating the coordination-level version and one translating the expression layer version. Both operate on the Hume abstract syntax tree, and a generic pretty printer is used to create the source code. Before translating, the expression layer version first checks that the stream ordering is correct. In both versions, a state monad is used to implement the meta-environment. The expression layer translation has an additional test, to verify that streams are in the required order. The program is executed as follows:

```
c2hume <program>.c
```

This will generate two programs if the type checker succeeds and the streams are in a correct order:

- `<program>-coord.hume` contains the coordination layer version.
- `<program>-expr.hume` contains the expression layer version. This will only be generated if the stream ordering is correct. If it is not, then only the coordination layer version is created.

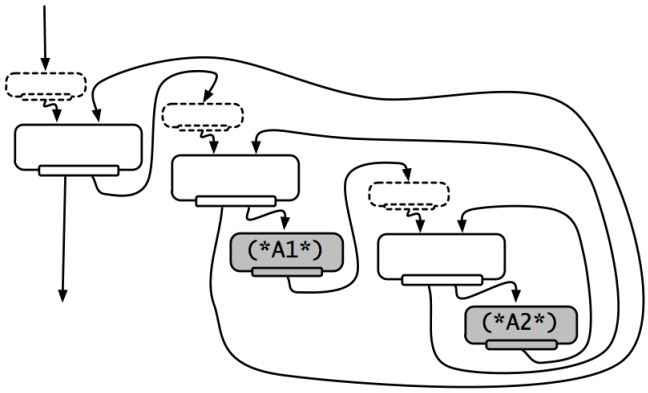


Figure 7: Box diagram of nested loops

8 Example

To illustrate MINI-C, and the translation into Hume in the next section, we will implement an algorithm which multiplies two 3×3 matrices (`a` and `b`) and stores the result in a 3×3 matrix (`c`):

```
#include<stdio.h>
main (){
int a[9]; int b[9]; int c[9];
int i; int j; int k;

for(i=3;0 < i;i=i-1){
  for(j=3;0 < j;j=j-1){
    c[((i-1)*3)+j-1] = 0; (*A1*)
    for(k=3;0 < k;k=k-1)
      c[((i-1)*3)+j-1] = c[((i-1)*3)+j-1]+a[((i-1)*3)+k-1]*b[((k-1)*3)+j-1]; (*A2*)
  }
 }
}
```

Since only one-dimensional arrays are supported, a 3×3 matrix is flattened into an array of length 9. Thus accessing element `a[i][j]` is instead written `a[(i*3)+j]` where 3 relates to the number of columns. In the algorithm `i` enumerates over the columns, while `j` enumerates of the rows. When multiplying the `a` matrix with the `b` matrix, each element of the resulting `c` matrix is the sum of each each corresponding column of `a` multiplied with the corresponding row of `b`. In the inner-most loop, `k` enumerates over these elements.

The two versions of the translated Hume code is discussed in the next sections, while Appendix B lists the full code.

8.1 Coordination layer version

In the translated *coordination layer version*, *assignment* is represented using one box containing one match, which updates the correct variable by the expression, and leaves the remaining variables unchanged. For example, the match of the statement of the matrix multiplication with the `(*A1*)` comment is

```
(a,b,c,i,j,k) -> (a,b,update c (((i*3)+j)+1) 0,i,j,k).
```

Program	Hume Time	Hume Time	HTA/HT	C-code Time	HTA/CT	Hume Heap Analysis	Hume Heap	HHA/HH
`fact`	121527	83659	1.45	1751	629	259	259	1.00
`fibsum`	211062	133193	1.59	1413	149	508	484	1.05
`matmult2`	425850	229469	1.86	713	597	962	834	1.15
`matmult3`	1271170	665363	1.91	2126	598	2981	2646	1.12
`arrayrotate`	506548	186851	2.71	746	679	1181	690	1.71
`mediumarrayrotate`	1092800	378035	2.89	1737	629	3011	1780	1.69
`smallarraysearch`	1072090	477408	2.25	2705	396	2485	1950	1.27

Table 1: Analysis and Measurement Results

Figure 7 shows a diagrammatical representation of the statements resulting from the nested `for`-loops of the matrix multiplication example. Here, the shaded boxes refers to the `(*A1*)` and `(*A2*)` statements, and the dotted boxes the initialisation of the loop.

8.2 Expression layer version

The tuple `(a,b,e,i,j,k)` is the state space of the *expression layer* version. Here,

```
storeC e (a,b,c,i,j,k) = (a,b,e,i,j,k);
loadC (a,b,c,i,j,k) = c;
```

are generated for the `c` variable in the above example. To illustrate the translation of a statement, the line with the `(*A1*)` comment results in a function `f st`, with the body:

```
storeC (update (loadC st)((((loadI st)*3)+(loadJ st))+1)0) st;
```

and the inner-most `for` statement, is translated into

```
g st k = (if (0 <= k) then (g (fbody st) (k - 1)) else st);
h st = g (i st) 2;
```

where `fbody st` is the result of translating the loop body (including `k=k-1`), and `i` translates the initialisation statement `k = 2`, since `k` may be used in the body. `h` initialises the accumulator variable.

9 Results

Table 1 summarises the results of analysing and measuring a range of example programs written in miniC and compiled to Hume. The Hume is then compiled to native code using the `humec` compiler chain which successively generates Hume Abstract Machine code and C for final compilation with `gcc`. Analysis and instrumentation are for a Renesas board incorporating an M32C processor with 24KBytes of memory for data, which restricts the heap usage to about 4000 cells. All the programs are listed in Appendix A.

HTA/HT in Table 1 compares analysis results with measured runtime for the Hume generated from the miniC. This is really a "sanity check" on the analysis and shows fair consistency of actual with predicted times for Hume. The heap part of the table also works as a "sanity check" since the C uses mostly the stack in all of these examples. The over-estimation is explained by the special structure of the automatically generated Hume code: it consists of many small functions, with heavy use of higher-order functions. The former leads to a high impact of inaccuracies in costing function calls. The latter

necessarily leads to defensive approximations, since the analysis must account for all possible instances of the supplied function argument. However, considering that these results are guaranteed upper bounds on execution time, we find them acceptable.

Of more interest is HTA/CT which shows the ratio of Hume analysis to miniC runtime. We would not expect the Hume analysis itself to correspond closely to the miniC time: after all, we have compiled miniC to Hume to C and so the resulting program is inevitably far less efficient than the original. Nonetheless, the ratios show considerable consistency for five of the test programs: `fact`, the `matmult`s and the `arrayrotate`s.

10 Relevant work

The translation of MINI-C to expression layer Hume, is really a translation from imperative to functional code. Such translations are very well known since the first use of functional notations in denotational semantics [Sch86]. In particular, functional meta-languages are commonly deployed in both semantics-directed compiler-compilers[Mic86] and theorem provers. For example, [NOP00] formalises a Java-like language in Isabelle/HOL, while in [Sch05] a more C-like language in mechanised in the same theorem prover.

Translating miniC into the Hume coordination bear resemblance to embedding the semantics of an imperative language into a model checker, since the modeling language of most model checker are finite state machine. For example, [HP00] translates Java code into the Promela modeling language for model checking with Spin.

11 Discussion

We have elaborated a translation from a canonical core imperative language miniC to Hume, to explore direct use of the Hume WCET analysis to characterise miniC programs. Our results suggests that:

- the Hume WCET analysis is in itself fairly robust;
- naive translation from miniC to Hume captures salient components of the complexity of the source programs;
- hence, the WCET analyses of translated miniC programs may be used to at least compare their relative time complexities.

Of course, our translation is naive and we have only conducted experiments on a small cohort of test programs. Nonetheless, our results suggest that this approach would repay further study and that analytic techniques for one language may fruitfully be used directly with another.

References

[HP00] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000. 10

[Mic86] G. Michaelson. Interpreters from functions and grammars. *Computer Languages*, 11(2):85–104, 1986. 10

[NOP00] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a Programming Language in a Theorem Prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000. 10

[Sch86] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn and Bacon, 1986. 10

[Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2005. 10

A Listing of all MINI-C programs

A.1 arrayrotate.c

```
#include<stdio.h>
main (){
  int arr[10];
  int rand;
  int cnt;
  arr[9] = 13;
  for(cnt=9; 0 < cnt; cnt = cnt - 1){
    arr[cnt-1] = (17 * arr[cnt]) % 25;
   }
  for(cnt=9; 4 < cnt;cnt = cnt - 1){
      rand = arr[cnt];
      arr[cnt] = arr[9-cnt];
      arr[9-cnt] = rand;
  }
}
```

A.2 fact.c

```
#include<stdio.h>
main (){
 int x;
 int res;
 res = 1;
 for(x=15;0 < x;x=x-1)
   res = res*x;
 printf("%d",res);
}
```

A.3 fibsum.c

```
#include<stdio.h>
main (){
 int x;
 int y;
 int res;
 res = 0;
 y = 20;
 for(x=20;1 < x;x=x-1)
   res = res + (x-1)+(x-2);
 printf("%d",res);
}
```

A.4 matmult2.c

```
#include<stdio.h>
main (){
int a[4]; int b[4]; int c[4];
```

```
int i; int j; int k;

for(i=2;0 < i;i=i-1){
  for(j=2;0 < j;j=j-1){
    c[((i-1)*2)+(j-1)] = 0;
    for(k=2;0 < k;k=k-1)
      c[((i-1)*2)+(j-1)] = c[((i-1)*2)+(j-1)]+a[((i-1)*2)+(k-1)]*b[((k-1)*2)+(j-1)];
  }
 }
}
```

A.5 matmult3.c

```
#include<stdio.h>
main (){
int a[9]; int b[9]; int c[9];
int i; int j; int k;

for(i=3;0 < i;i=i-1){
  for(j=3;0 < j;j=j-1){
    c[((i-1)*3)+j-1] = 0;
    for(k=3;0 < k;k=k-1)
      c[((i-1)*3)+j-1] = c[((i-1)*3)+j-1]+a[((i-1)*3)+k-1]*b[((k-1)*3)+j-1];
  }
 }
}
```

A.6 matmult4.c

```
#include<stdio.h>
main (){
int a[16]; int b[16]; int c[16];
int i; int j; int k;

for(i=4;0 < i;i=i-1){
  for(j=4;0 < j;j=j-1){
    c[((i-1)*4)+j-1] = 0;
    for(k=4;0 < k;k=k-1)
      c[((i-1)*4)+j-1] = c[((i-1)*4)+j-1]+a[((i-1)*4)+k-1]*b[((k-1)*4)+j-1];
  }
 }
}
```

A.7 meduimarrayrotate.c

```
#include<stdio.h>
main (){
  int arr[20];
  int rand;
  int cnt;
```

```
  arr[19] = 13;
  for(cnt=19; 0 < cnt; cnt = cnt - 1){
    arr[cnt-1] = (17 * arr[cnt]) % 25;
   }
  for(cnt=19; 9 < cnt;cnt = cnt - 1){
      rand = arr[cnt];
      arr[cnt] = arr[19-cnt];
      arr[19-cnt] = rand;
  }
}
```

A.8 smallarraysearch.c

```
#include<stdio.h>
main (){
  int arr[25];
  int rand;
  int cnt;
  int res;
  rand = 13;
  res = 1;
  arr[24] = (rand+5) % 13;
  for(cnt=24; 0 < cnt; cnt = cnt - 1){
    arr[cnt-1] = (17 * arr[cnt]) % 13;
   }
  for(cnt=25;0 < cnt;cnt = cnt - 1){
    if(arr[cnt-1] == rand)
       res = arr[cnt-1];
  }
  printf("%d",res);
}
```

B Generated code from `fact.c`

B.1 Expression layer version generated code

```
data mybool = TT  |  FF;

storeA e (a,b,c,i,j,k) = (e,b,c,i,j,k);
storeB e (a,b,c,i,j,k) = (a,e,c,i,j,k);
storeC e (a,b,c,i,j,k) = (a,b,e,i,j,k);
storeI e (a,b,c,i,j,k) = (a,b,c,e,j,k);
storeJ e (a,b,c,i,j,k) = (a,b,c,i,e,k);
storeK e (a,b,c,i,j,k) = (a,b,c,i,j,e);

loadA (a,_,_,_,_,_) = a;
loadB (_,b,_,_,_,_) = b;
loadC (_,_,c,_,_,_) = c;
loadI (_,_,_,i,_,_) = i;
loadJ (_,_,_,_,j,_) = j;
loadK (_,_,_,_,_,k) = k;

exIf TT s st = (s st);
exIf FF _ st = st;
exIfElse TT s _ st = (s st);
exIfElse FF _ t st = (t st);
loop e s st = (if ((e st) == TT) then (loop e s (s st)) else st);

f1 st = (storeI 3 st);
f2 st = (storeJ 3 st);
f3 st = (storeC (update (loadC st) (((((loadI st) - 1) * 3) + ((loadJ st) - 1)) + 1) 0) st);
f4 st = (storeK 3 st);
f5 st = (storeC (update (loadC st) (((((loadI st) - 1) * 3) + ((loadJ st) - 1)) + 1)
                   (((loadC st) @ (((((loadI st) - 1) * 3) + ((loadJ st) - 1)) + 1)) +
                   (((loadA st) @ (((((loadI st) - 1) * 3) + ((loadK st) - 1)) + 1)) *
                   ((loadB st) @ (((((loadK st) - 1) * 3) + ((loadJ st) - 1)) + 1))))) st);
f6 st = (storeK ((loadK st) - 1) st);
f8 st = (f6 (f5 st));
f7 st k = (if (0 < k) then (f7 (f8 st) (k - 1)) else st);
f9 st = (f7 st 3);
f10 st = (storeJ ((loadJ st) - 1) st);
f12 st = (f10 (f9 (f4 (f3 st))));
f11 st j = (if (0 < j) then (f11 (f12 st) (j - 1)) else st);
f13 st = (f11 st 3);
f14 st = (storeI ((loadI st) - 1) st);
f16 st = (f14 (f13 (f2 st)));
f15 st i = (if (0 < i) then (f15 (f16 st) (i - 1)) else st);
f17 st = (f15 st 3);
f18 (a,b,c,i,j,k) = (*,*,*,*,*,*);

box mainprogram
 in (a::vector 9 of int 16,b: vector 9 of int 16,
      c::vector 9 of int 16,i::int 16,j::int 16,k::int 16)
 out (a'::vector 9 of int 16,b'::vector 9 of int 16,
         c': vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
match
(a,b,c,i,j,k) -> (f18 (f17 (f1 (a,b,c,i,j,k)))) ;

wire mainprogram
 ( mainprogram.a' initially << 0,0,0,0,0,0,0,0,0 >>,
   mainprogram.b' initially << 0,0,0,0,0,0,0,0,0 >>,
   mainprogram.c' initially << 0,0,0,0,0,0,0,0,0 >>,
```

```
   mainprogram.i' initially 0,
   mainprogram.j' initially 0,
   mainprogram.k' initially 0)
 ( mainprogram.a,mainprogram.b,mainprogram.c,
    mainprogram.i,mainprogram.j,mainprogram.k);
```

B.2 Coordination layer version generated code

```
box first
 in (_fv0::bool)
 out (a::vector 9 of int 16,b::vector 9 of int 16,
    c::vector 9 of int 16,i::int 16,j::int 16,k::int 16,_fv1::bool)
match
_ -> (<< 0,0,0,0,0,0,0,0,0 >>,<< 0,0,0,0,0,0,0,0,0 >>,<< 0,0,0,0,0,0,0,0,0 >>,0,0,0,*) ;

box box1
 in (a::vector 9 of int 16,b::vector 9 of int 16,
       c::vector 9 of int 16,i::int 16,j::int 16,k::int 16)
 out (a'::vector 9 of int 16,b'::vector 9 of int 16,
         c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
match
(a,b,c,i,j,k) -> (a,b,c,3,j,k) ;

box box2
 in (a::vector 9 of int 16,b::vector 9 of int 16,
      c::vector 9 of int 16,i::int 16,j::int 16,k::int 16,
      a'::vector 9 of int 16,b'::vector 9 of int 16,
       c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
 out (a''::vector 9 of int 16,b''::vector 9 of int 16,
         c''::vector 9 of int 16,i''::int 16,j''::int 16,k''::int 16,
         a'''::vector 9 of int 16,b'''::vector 9 of int 16,
         c'''::vector 9 of int 16,i'''::int 16,j'''::int 16,k'''::int 16)
match
(*,*,*,*,*,*,a,b,c,i,j,k) -> (if (0 < i) then (*,*,*,*,*,*,a,b,c,i,j,k) else (a,b,c,i,j,k,*,*,*,*,*,*)) |
(a,b,c,i,j,k,*,*,*,*,*,*) -> (if (0 < i) then (*,*,*,*,*,*,a,b,c,i,j,k) else (a,b,c,i,j,k,*,*,*,*,*,*)) ;

wire box1 (first.a,first.b,first.c,first.i,first.j,first.k) (box2.a,box2.b,box2.c,box2.i,box2.j,box2.k);

box box3
 in (a::vector 9 of int 16,b::vector 9 of int 16,
         c::vector 9 of int 16,i::int 16,j::int 16,k::int 16)
 out (a'::vector 9 of int 16,b'::vector 9 of int 16,
         c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
match
(a,b,c,i,j,k) -> (a,b,c,i,3,k) ;

box box4
 in (a::vector 9 of int 16,b::vector 9 of int 16,
       c::vector 9 of int 16,i::int 16,j::int 16,k::int 16,
       a'::vector 9 of int 16,b'::vector 9 of int 16,
       c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
 out (a''::vector 9 of int 16,b''::vector 9 of int 16,
         c''::vector 9 of int 16,i''::int 16,j''::int 16,k''::int 16,
         a'''::vector 9 of int 16,b'''::vector 9 of int 16,
         c'''::vector 9 of int 16,i'''::int 16,j'''::int 16,k'''::int 16)
match
(*,*,*,*,*,*,a,b,c,i,j,k) -> (if (0 < j) then (*,*,*,*,*,*,a,b,c,i,j,k) else (a,b,c,i,j,k,*,*,*,*,*,*)) |
(a,b,c,i,j,k,*,*,*,*,*,*) -> (if (0 < j) then (*,*,*,*,*,*,a,b,c,i,j,k) else (a,b,c,i,j,k,*,*,*,*,*,*)) ;

wire box3 (box2.a''',box2.b''',box2.c''',box2.i''',box2.j''',box2.k''')
```

```
        (box4.a,box4.b,box4.c,box4.i,box4.j,box4.k);

box box5
 in (a::vector 9 of int 16,b::vector 9 of int 16,
       c::vector 9 of int 16,i::int 16,j::int 16,k::int 16)
 out (a'::vector 9 of int 16,b'::vector 9 of int 16,
        c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
match
(a,b,c,i,j,k) -> (a,b,(update c ((((i - 1) * 3) + (j - 1)) + 1) 0),i,j,k) ;

box box6
 in (a::vector 9 of int 16,b::vector 9 of int 16,
       c::vector 9 of int 16,i::int 16,j::int 16,k::int 16)
 out (a'::vector 9 of int 16,b'::vector 9 of int 16,
        c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
match
(a,b,c,i,j,k) -> (a,b,c,i,j,3) ;

wire box5 (box4.a''',box4.b''',box4.c''',box4.i''',box4.j''',box4.k''')
        (box6.a,box6.b,box6.c,box6.i,box6.j,box6.k);

box box7
 in (a::vector 9 of int 16,b::vector 9 of int 16,
       c::vector 9 of int 16,i::int 16,j::int 16,k::int 16,
       a'::vector 9 of int 16,b'::vector 9 of int 16,
       c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
 out (a''::vector 9 of int 16,b''::vector 9 of int 16,
        c''::vector 9 of int 16,i''::int 16,j''::int 16,k''::int 16,
        a'''::vector 9 of int 16,b'''::vector 9 of int 16,
        c'''::vector 9 of int 16,i'''::int 16,j'''::int 16,k'''::int 16)
match
(*,*,*,*,*,*,a,b,c,i,j,k) -> (if (0 < k) then (*,*,*,*,*,*,a,b,c,i,j,k) else (a,b,c,i,j,k,*,*,*,*,*,*)) |
(a,b,c,i,j,k,*,*,*,*,*,*) -> (if (0 < k) then (*,*,*,*,*,*,a,b,c,i,j,k) else (a,b,c,i,j,k,*,*,*,*,*,*)) ;

wire box6 (box5.a',box5.b',box5.c',box5.i',box5.j',box5.k') (box7.a,box7.b,box7.c,box7.i,box7.j,box7.k);

box box8
 in (a::vector 9 of int 16,b::vector 9 of int 16,
       c::vector 9 of int 16,i::int 16,j::int 16,k::int 16)
 out (a'::vector 9 of int 16,b'::vector 9 of int 16,
        c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
match
(a,b,c,i,j,k) -> (a,b,(update c ((((i - 1) * 3) + (j - 1)) + 1)
             ((c @ ((((i - 1) * 3) + (j - 1)) + 1)) + ((a @ ((((i - 1) * 3) + (k - 1)) + 1)) *
             (b @ ((((k - 1) * 3) + (j - 1)) + 1))))),i,j,k) ;

box box9
 in (a::vector 9 of int 16,b::vector 9 of int 16,c::vector 9 of int 16,i::int 16,j::int 16,k::int 16)
 out (a'::vector 9 of int 16,b'::vector 9 of int 16,c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
match
(a,b,c,i,j,k) -> (a,b,c,i,j,(k - 1)) ;

wire box8 (box7.a''',box7.b''',box7.c''',box7.i''',box7.j''',box7.k''')
             (box9.a,box9.b,box9.c,box9.i,box9.j,box9.k);

wire box9 (box8.a',box8.b',box8.c',box8.i',box8.j',box8.k')
(box7.a',box7.b',box7.c',box7.i',box7.j',box7.k');

box box10
 in (a::vector 9 of int 16,b::vector 9 of int 16,
```

```
     c::vector 9 of int 16,i::int 16,j::int 16,k::int 16)
 out (a'::vector 9 of int 16,b'::vector 9 of int 16,
     c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
match
(a,b,c,i,j,k) -> (a,b,c,i,(j - 1),k) ;

wire box7 (box6.a',box6.b',box6.c',box6.i',box6.j',box6.k',box9.a',box9.b',box9.c',box9.i',box9.j',box9.k')
     (box10.a,box10.b,box10.c,box10.i,box10.j,box10.k,box8.a,box8.b,box8.c,box8.i,box8.j,box8.k);

wire box10 (box7.a'',box7.b'',box7.c'',box7.i'',box7.j'',box7.k'')
  (box4.a',box4.b',box4.c',box4.i',box4.j',box4.k');

box box11
 in (a::vector 9 of int 16,b::vector 9 of int 16,
      c::vector 9 of int 16,i::int 16,j::int 16,k::int 16)
 out (a'::vector 9 of int 16,b'::vector 9 of int 16,
      c'::vector 9 of int 16,i'::int 16,j'::int 16,k'::int 16)
match
(a,b,c,i,j,k) -> (a,b,c,(i - 1),j,k) ;

wire box4 (box3.a',box3.b',box3.c',box3.i',box3.j',box3.k',
        box10.a',box10.b',box10.c',box10.i',box10.j',box10.k')
  (box11.a,box11.b,box11.c,box11.i,box11.j,box11.k,
  box5.a,box5.b,box5.c,box5.i,box5.j,box5.k);

wire box11 (box4.a'',box4.b'',box4.c'',box4.i'',box4.j'',box4.k'')
(box2.a',box2.b',box2.c',box2.i',box2.j',box2.k');

wire first (first._fv1 initially true) (box1.a,box1.b,box1.c,box1.i,box1.j,box1.k,first._fv0);

box last
 in (a::vector 9 of int 16,b::vector 9 of int 16,
       c::vector 9 of int 16,i::int 16,j::int 16,k::int 16,_fv0::bool)
 out (_fv1::bool)
match
(a,b,c,i,j,k,_) -> * ;

wire box2 (box1.a',box1.b',box1.c',box1.i',box1.j',box1.k',
        box11.a',box11.b',box11.c',box11.i',box11.j',box11.k'
(last.a,last.b,last.c,last.i,last.j,last.k,
  box3.a,box3.b,box3.c,box3.i,box3.j,box3.k);

wire last (box2.a'',box2.b'',box2.c'',box2.i'',box2.j'',box2.k'',last._fv1) (last._fv0);
```