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

Specific Targeted Research Project (STReP)
FET Open

D35 (WP9b): Destructive Update and Dynamic Data Structures

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: 1.25

Purpose: The purpose of work package 9 is to show application of Hume analysis to more traditional settings, i.e. imperative languages. The purpose of this deliverable is to show application of Hume analysis to a language with pointers and memory managment.

Results: The main results of this deliverable are a translator program from a small subset of C with pointers, into Hume, and the application of the Hume analysis tools to the resulting Hume code.

Conclusion: The main conclusion is that a applying the Hume analysis to a language with pointers and memory management is inclusive. In particularly, compared to the work in deliverable 32 (of work package 9), where the Hume analysis were applied to imperative programs without pointers and memory management.

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)	

Destructive Update and Dynamic Data Structures

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

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

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

Abstract

We define miniC*, an extension of miniC from Deliverable 32 with pointers and dynamic memory management. We then define the translation rules for miniC* into Hume, and implement them in a Haskell program. The Hume analysis are applied to a set of generated miniC* programs, via the translator. The original miniC*, and generated Hume programs are also measured, and the measurements are compared to the analysis results.

The positioning of this deliverable is discussed in Section 2.

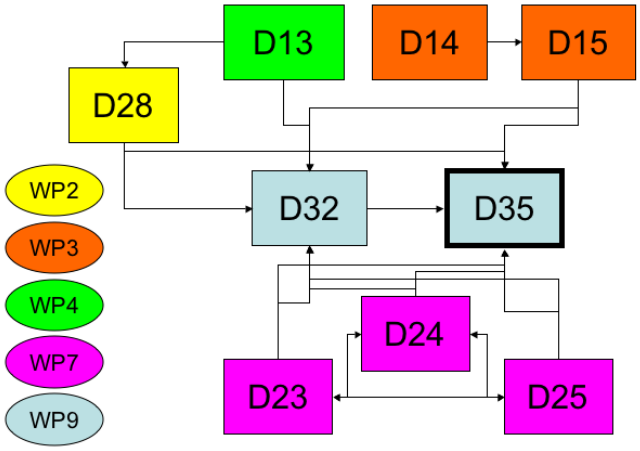
Major Revisions		
Revision	Date	Changes
1.25	14 Aug. 2009	table-of-contents, positioning in Sec 2, fixed figures (addressing Review Report Year 4)
1.22	31 Aug. 2008	initial version

Contents

1 Introduction

In D32 [GML+08] we defined MINIC, a small imperative subset of ANSI-C, with integers and arrays. This was then translated into Hume, to apply the Hume resource analysis. Here, we extend MINIC with pointers and dynamic memory managment. This new language MINIC*, removes the array type, since this can be implemented using pointers and the memory management. Since this is merely a proof-of-concept experiment, we have implemented the simplest management model in [KR88], i.e. with `alloc` and `afree` functions. However, since this is not supported by newer `gcc` versions, we use `malloc` and `free`, although we ensure that `free` is always applied to the last allocated variable, so that the `alloc`/`afree` model is still valid.

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	WP3b	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 shows the dependencies with the other deliverables. This is a direct extension of deliverable 32, thus it relies directly on it. Moreover, 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 Hume analysis, described in work package 3 and 4. 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.

3 Source language: MINIC*

Figure 2 shows the abstract syntax of the MINIC* language. It contains standard arithmetic operators together with tests for equality, inequality, smaller than, and smaller or equal to. However, since this is

program ::=	*prelude* `main()` *body*	
prelude ::=	$include_0$ ... $include_n$	$n \geq 0$
include ::=	`# include <`*id*`.h>` | `# include "`*id*`.h"`	
body ::=	$\{decl_1 \cdots decl_n\ [scan]\ stmt_1 \cdots stmt_m\ print_0 \cdots print_l\}$	$n, m \geq 1, l \geq 0$
cbody ::=	$stmt \mid \{stmt_1 \cdots stmt_n\}$	$n \geq 1$
star ::=	*id* | `*`*star*	
decl ::=	`int` *star;*	
scan ::=	`scanf ( "%d" ,` *expr* `);`	
print ::=	`printf ( "%d" ,` *expr* `);`	
caststar ::=	`int` | *caststar*`*`	
cast ::=	`(` *caststar* `)`	
stmt ::=	*id* `=` *expr;*	
	| `if (` *expr* `)` *cbody*	
	| `if (` *expr* `)` $cbody_1$ `else` $cbody_2$	
	| `for (` $stmt_1$ `;`*expr* `;` $stmt_2$`)` *cbody*	
	| `*`$expr_1$ `=` $expr_2$`;`	
	| $expr_1$ `= [` *cast* `]` `malloc(` $expr_2$ `[ * sizeof(int) ] )`	
	| `free(` *expr* `);`	
expr ::=	*int*	
	| *id*	
	| $expr_1$ *binop* $expr_2$	
	| `(` *expr* `)`	
	| *cast expr*	
	| `*`*expr*	
	| `&`*id*	
binop ::=	`==` | `!=` | `<` | `<=` | `+` | `-` | `*` | `/` | `%`	

Figure 2: MINIC* abstract syntax

a subset of ANSI-C, there is not a separate boolean type. Moreover, conditional with an optional `else` branch, and repetition (`for`) are supported.

Now, our target is an expression version of Hume, represented by one box. Thus, as shown in the syntax, an input stream* precedes all other statements, while output streams are preceded by all the other statements.

The key of this deliverable is to test how Hume's costing models relates to destructive update and dynamic data structures. Thus, the key of the syntax is the pointer types and the memory management via `malloc` and `free`. We support multiple level of indention for a pointer type, thus arrays (pointers), arrays of arrays (pointers to pointers) and so on is supported. Note, that the array type is omitted from deliverable 32 since this an array can be implemented using a pointer and `malloc`.

Most C-compilers require casting between pointer and integer type. This is not required in the translated Hume version, and is thrown away by the translator. However, casting is supported in the syntax, to enable compilation of the C programs, which is required for the analysis. Finally, to support portability, the `sizeof(int)` function is supported with `malloc` to ensure portability. Note, that the translator assumed the use of this, and may be wrong if it omitted in an example.

*Since all input streams are from standard input (`scanf`), i.e. the same source, it does not make sense with more than 1 stream. If more than one were supported, they would just stream the same value.

4 Translation rules

4.1 Notation

We will use the same meta-language as in deliverable 32 to describe the translation: a functional style is used, where we 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. We write $[x, y]$ for the list consisting of x and y. @ represents 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* return a meta-value in addition to the generated code – while *FRESH()* generates a fresh name. Conditionals are supported by *IF-THEN-ELSE* and *IF-THEN*.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 and miniC* code, while *this font* is used on the meta-language. Generated Hume code are separated from miniC* by underlining. 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.

4.2 Memory management

Since miniC* is state-based, and the Hume expression layer is purely functional, the state-space must be explicitly passed between function calls. Now, since only integers (and pointers) are allowed, we have simplified the typing by not separating the type of an address location and an actual value. Moreover, we have chosen to represent the state as a vector with the size provided by the user. Thus, the memory usage will be the same, or at least not more than the size of the vector. Moreover, since manipulating the vector size would be a very time-consuming operation, we have decided to keep the state-space vector size constant. Thus, when more space than allocated by the vector is attempted, the program will fail. Consequently, since the vector size is fixed, we will *only target time resources* in the analysis.

The space vector consists of three elements: the stack, which hold the variables; the allocated heap, containing the dynamically allocated space: and the unallocated heap, containing the free memory:

$$\textbf{the state vector: } \texttt{<<}\underbrace{v_1, \cdots, v_{\texttt{stack}}}_{\textbf{1.stack}}, \underbrace{\overbrace{v_{\texttt{stack+1}}, \cdots, v_{\texttt{l-1}}}^{\textbf{2.allocated}}, \overbrace{v_{\texttt{l}}, \cdots v_{\texttt{maxsize}}}^{\textbf{3.unallocated}}}_{\textbf{heap}}\texttt{>>}$$

where `stack` and `maxsize` are constants identifying the size of the stack and total available memory respectively. `l` is a variable holding the current allocation of the heap. Now, allocating and de-allocating the memory is achieved by moving `l` up and down the vector. Although the more complex `malloc` and `free` are used, we have implemented the simplest forms of memory management in C: `alloc` and `afree`, which enable this representation by requiring that `afree` can only be applied to the last variable allocated to `alloc`. This is an assumption in the code, and we will not implement checks for it. It is assumed that `malloc` and `free` behaves in the same way, when applied with the same assumptions. The reason `malloc` and `free` are used, is that most newer C compilers does not support `alloc` and `afree`. The following two functions are functional representations the imperative C definitions of `afree` and `alloc` in Hume:

$\mathcal{T}_{memory} \rightsquigarrow$

```
afree n l =  if n > stack && n < l then n else 0;
alloc n l =  if n <= maxsize - l then l+n else 0;
```

`afree` is given the location of the last allocated variable, and the current last location l, and returns the new location; `afree` is given the requested size and current last location, and attempt to allocate the requested size, and returns the new last location.

4.3 Declarations

MINIC* is a proper ANSI-C subset. Moreover, local declarations are not allowed, thus, the environment is static. To represent the variables an enumeration type is created. We will first discuss the creation of the state enumeration type, before the rules which creates the environment is defined.

4.3.1 The state space enumeration type

We assume here that there is a function *toUpper* which converts a char to its upper case, and that strings are represented as list of chars. Since a variable cannot be an empty, and all MINIC* variables are unique, the following rule converts an identifier into a constructor:

$$\mathcal{T}_{var}\ \ (x :: xs)\ \leadsto\ \underline{\mathit{toUpper}\ x} :: \underline{xs}$$

$\mathcal{T}_{data}$ creates the `Vars` datatype, which enumerates the state space. It uses the rule $\mathcal{T}_{cons}$ to translate the variables, which uses $\mathcal{T}_{var}$:

$$\begin{array}{lcl}
\mathcal{T}_{cons}\ (\texttt{int}\{\texttt{*}\}^{+}x) & \leadsto & \mathcal{T}_{var}\ (x) \\
\mathcal{T}_{cons}\ (d_1\,;d_2) & \leadsto & \mathcal{T}_{cons}\ (d_1)\ \ \underline{|}\ \ \mathcal{T}_{cons}\ (d_2) \\
 & & \\
\mathcal{T}_{data}\ (\mathit{decls}) & \leadsto & \underline{\texttt{data Var =}}\ \mathcal{T}_{cons}\ (\mathit{decls})\ \ \underline{;}
\end{array}$$

Note that $\mathcal{T}_{var}$ is defined separately since it used in the translation rules for expressions and statements

4.3.2 The environment: `env`

The environment maps each variable to an unique element in the state space vector. Uniqueness is achieved by providing the translation function $\mathcal{T}_{env}$ the first available location (which should initially be `1`):

$$\begin{array}{lcl}
\mathcal{T}_{env}\ \ (\texttt{int}\{\texttt{*}\}^{+}x)\ n & \leadsto & \underline{\texttt{env}}\ \mathcal{T}_{var}\ (x)\ \underline{\texttt{= } n\ \texttt{;}} \\
 & & \mathit{RETURN}\ (n+1) \\
\mathcal{T}_{env}\ \ (d_1\,;d_2)\ n & \leadsto & \mathit{LET}\ n' = \mathcal{T}_{env}\ (d_1)\ n\ \mathit{IN} \\
 & & \mathit{RETURN}\ (\mathcal{T}_{env}\ (d_2)\ n')
\end{array}$$

4.3.3 Separating the stack from the heap: the `stack` constant

The *stacksize* meta-function is used to create the `stack` constants which separates the stack from the heap in the state space vector (see above). It returns the size of the stack, i.e. the number of program variables:

$$\begin{array}{lcl}
\mathit{stacksize}\ (\texttt{int}\ \{\texttt{*}\}^{*}x\ ) & = & 1 \\
\mathit{stacksize}\ (d_1\ ;d_2) & = & (\mathit{stacksize}\ d_1) + (\mathit{stacksize}\ d_2)
\end{array}$$

4.4 Operators

Figure 3 shows the translation rules for MINIC* operators. These are trivial, however, note that `%` is represented by `mod` in Hume. We assume the same precedence of MINIC* and Hume operators.

4.5 Expressions

Figure 4 shows the translation rules for MINIC* expressions. $\mathcal{T}'_e$ is assumed to be applied in a setting where the state vector `state` is available, while $\mathcal{T}_e$ explicitly declares a function which takes this variables as input, and uses $\mathcal{T}'_e$ to translate the function expression. $\mathcal{T}_e$ returns the identifier of the generated function. Finally, $\mathcal{T}''_e$ is required to translate x in x `= alloc(`e`)`. Here, the lookup in `state`

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

Figure 3: $\mathcal{T}_o$: translation rules for MINIC* operators.

$$
\begin{array}{lcl}
\mathcal{T}'_e\ (\ int\ ) & \leadsto & \underline{int} \\
\mathcal{T}'_e\ (\ id\ ) & \leadsto & \underline{\texttt{state@(env}}\ \mathcal{T}_{var}\ (id)\ \underline{)} \\
\mathcal{T}'_e\ (\ expr_1\ binop\ expr_2\ ) & \leadsto & \mathcal{T}'_e\ (expr_1)\quad \mathcal{T}_o\ (\ binop\ )\quad \mathcal{T}'_e\ (expr_2) \\
\mathcal{T}'_e\ (\ (expr)\ ) & \leadsto & \underline{(}\ \mathcal{T}'_e\ (expr)\ \underline{)} \\
\mathcal{T}'_e\ (\ cast\ expr\ ) & \leadsto & \mathcal{T}'_e\ (expr) \\
\mathcal{T}'_e\ (\ \&id\ ) & \leadsto & \underline{\texttt{env\_}}\ \mathcal{T}_{var}\ (id) \\
\mathcal{T}'_e\ (\ *expr\ ) & \leadsto & \underline{\texttt{state@(}}\ \mathcal{T}'_e\ (\ expr\ )\underline{)} \\
 & & \\
\mathcal{T}_e\ (\ expr\ ) & \leadsto & LET\ x = FRESH()\ IN \\
 & & \underline{x\ \texttt{state =}}\ \mathcal{T}'_e\ (\ expr\ )\ \underline{;} \\
 & & RETURN\ x \\
 & & \\
\mathcal{T}''_e\ (\ int\ ) & \leadsto & \underline{int} \\
\mathcal{T}''_e\ (\ id\ ) & \leadsto & \underline{\texttt{env\_}}\ \mathcal{T}_{var}\ (id) \\
\mathcal{T}''_e\ (\ expr_1\ binop\ expr_2\ ) & \leadsto & \mathcal{T}''_e\ (expr_1)\quad \mathcal{T}_o\ (\ binop\ )\quad \mathcal{T}''_e\ (expr_2) \\
\mathcal{T}''_e\ (\ (expr)\ ) & \leadsto & \underline{(}\ \mathcal{T}''_e\ (expr)\ \underline{)} \\
\mathcal{T}''_e\ (\ \&id\ ) & \leadsto & \underline{\texttt{env\_}}\ \mathcal{T}_{var}\ (id) \\
\mathcal{T}''_e\ (\ *expr\ ) & \leadsto & \mathcal{T}'_e\ (\ expr\ )
\end{array}
$$

Figure 4: $\mathcal{T}_e$,$\mathcal{T}'_e$ and $\mathcal{T}''_e$: translation rules for MINIC* expression

is removed for variables and pointers, compared to $\mathcal{T}'_e$. This is a result of the use of the Hume built-in `update` function for vectors.

Now, since a memory location and a value is of the same type, the translation rules are rather uncomplicated: An identifier is looked up in the environment and the vector lookup function `@` returns the value for the given location/index; The de-reference operator `&` returns the location of a variable; while its complement `*` returns the value of the location, which the expression holds.

4.5.1 Conditional expressions

In MINIC*, boolean values are represented as 0 and not 0. In Hume, this is represented with a boolean type, with the constants `true` and `false`. Thus, in MINIC*, $0 + 1$ and `0!=1` both succeeds while, in this have different types in Hume. Thus, in a MINIC* condition, found in `if` and `for` statements, both numeric and boolean values can be used. Thus, numeric values must in these case be converted into boolean values. Now, the Hume analyser toolset does not work well with the built in boolean type. Thus, a special boolean type `mybool` is first defined:

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

$\mathcal{T}'_c$ and $\mathcal{T}_c$ converts MINIC* expression into Hume expression returning a `mybool` type. *isNumeric*, defined as follows:

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

is a predicate, checking whether the MINIC* expression is of numeric, and not "boolean" type. $\mathcal{T}'_c$ returns a Hume `mybool` expression, whilst $\mathcal{T}_c$ turns this into a function over the state. The latter is required when translating `for` loops.

$$\begin{array}{lcl}
\mathcal{T}'_c\ (\mathit{expr}) & \rightsquigarrow & \underline{\texttt{if (}}\ \mathit{isNumeric}\ \mathit{expr} \\
 & & \quad \mathit{THEN}\ \mathcal{T}'_e\ (\mathit{expr})\ \underline{\texttt{!= 0) then TT else FF}} \\
 & & \quad \mathit{ELSE}\ \mathcal{T}'_e\ (\mathit{expr})\ \underline{\texttt{) then TT else FF}} \\
\mathcal{T}_c\ (\mathit{expr}) & \rightsquigarrow & \mathit{LET}\ x = \mathit{FRESH()}\ \mathit{IN} \\
 & & \underline{x\ \texttt{state =}}\ \mathcal{T}'_c\ (\mathit{expr})\ \underline{\texttt{;}} \\
 & & \mathit{RETURN}\ x
\end{array}$$

4.6 Statements

4.6.1 Higher order functions (HOFs)

As in the expression version of deliverable 32, the Hume program resulting from the MINIC* translation, uses higher order functions to express several statements. Moreover, the type is simpler here since the state space is vector. Thus, assignment can be represented as a HOF as well. The following HOFs are generated for all programs:

$\mathcal{T}_{HOF} \rightsquigarrow$

```
type int_t = int 16;
type state_t = (vector maxsize of int_t,int_t)
type stmt_t = state_t -> state_t;
```

```
assign ::  int_t -> int_t -> stmt_t
assign loc v (state,l) = ((update state loc v),l);

exIf :: mybool -> stmt_t -> stmt_t
exIf TT s (state,l) = (s (state,l));
exIf FF _ (state,l) = (state,l);

exIfElse :: mybool -> stmt_t -> stmt_t -> stmt_t
exIfElse TT s _ (state,l) = (s (state,l));
exIfElse FF _ t (state,l) = (t (state,l));

loop :: (state_t -> mybool) -> stmt_t -> stmt_t
loop e s (state,l) = (if ((e state) == TT)
                         then (loop e s (s (state,l)))
                         else (state,l));
```

Firstly, `maxsize` is the size of the state space vector and is discussed below. The state type is thus a vector of this size (of the 16 bit integer type), paired with an integer keeping the first position of the unallocated heap; `assign` uses the Hume built-in vector `update` function to change the given location `loc` to the value `v`; `exIf` is the result of an `if`-statement without and `else` clause; while `exIfElse` contain an else branch. Both of these assumes an expression of the `mybool` type as condition type; finally, `loop` is a tail recursive function, where the condition is of type `state_t -> mybool`. The type cannot be `mybool` due to recursion and the value of the expression may change between calls.

4.6.2 Translating the statements

Figure 5 shows the translation rules for MINIC* statements. Firstly, `malloc` is not represented as an expression, since it changes the `state` and last location `l`. Representing it as an expression would either require expressions to return the state and last location, or complicate the translation rules drastically.

Now, $expr_1$ `=` [*cast*] `malloc(` $expr_2$ [`* sizeof(int)`] `)` updates the location $expr_1$ to the first free location `l` on the heap. Both the potential casting and `* sizeof(int)` is ignored in translated Hume code. For correctness, the latter is in fact assumed, unless a heap cell is 16 Bytes in the architecture (which is the case for the Hume vector in the resulting program). `l` is changed to the new next free location (if existing), by applying the previously defined `alloc` function, which increments `l` with $expr_2$ locations; `afree(` *expr* `)` de-allocates the location *expr* and all subsequent locations. *expr* is assumed to be the last allocated location. If not, the program may behave differently compared to compiling and running the MINIC* program using `gcc`. It sets the value in the given *expr* location to `0`, as the case is in ANSI-C; assignment, i.e. *id* `=` *expr*, is handled by the `assign` HOF. Here, $\mathcal{T}''_e$ is used to translate the identifier, and look up the location in the environment; statement sequences is handled as functional composition; while `if` statements are handled by the HOFs. Note that $\mathcal{T}'_c$ is applied to the condition, to return a `mybool` type.

The translation rule for a `for`-statement is not shown in Figure 5. The Hume analysis tools will fail on a generic use of `loop` HOF. Thus, two rules are provided for a `for`-statement: one for a costable subset which does not use the `loop` HOF; and one for the generic case. The latter is added for completeness, to avoid over-restricting the MINIC* language.

The costable subset is first discussed: here, in addition to the syntactic constraints shown by the rule, there is an additional requirements that *binop* is either `<` or `<=`. Moreover, the location of *id* should not be updated by $stmt_3$, although this has to be assumed, since it cannot be syntactically checked due to the support of pointers in MINIC*. The translation rule for this costable subset is defined as follows:

$\mathcal{T}_s$ (*expr*$_1$ = [*cast*] malloc(*expr*$_2$ [* sizeof(int)]))
$\rightsquigarrow$ *LET* x = *FRESH() IN*
x (state,l) = (update state ($\mathcal{T}''_e$ (*expr*$_1$)) l, alloc $\mathcal{T}'_e$ (*expr*$_2$) l);

$\mathcal{T}_s$ (free(*expr*)) $\rightsquigarrow$ *LET* x = *FRESH() IN*
x (state,l) = let v = $\mathcal{T}'_e$ (*expr*) in (update state v 0, afree v l);

$\mathcal{T}_s$ (*$expr_1$ = $expr_2$) $\rightsquigarrow$ *LET* x = *FRESH() IN*
x (state,l) = assign ($\mathcal{T}'_e$ (*expr*$_1$)) ($\mathcal{T}'_e$ (*expr*$_2$)) (state,l);
RETURN x

$\mathcal{T}_s$ (*id* = *expr*) $\rightsquigarrow$ *LET* x = *FRESH() IN*
x (state,l) = assign $\mathcal{T}''_e$ (*id*) ($\mathcal{T}'_e$ (*expr*)) (state,l);
RETURN x

$\mathcal{T}_s$ (*stmt*$_1$; *stmt*$_2$) $\rightsquigarrow$ *LET* $f = \mathcal{T}_s$ (*stmt*$_1$) *IN*
LET $g = \mathcal{T}_s$ (*stmt*$_2$) *IN*
RETURN $g(f)$

$\mathcal{T}_s$ (if (*expr*) *stmt*) $\rightsquigarrow$ *LET* x = *FRESH() IN*
LET $s = \mathcal{T}_s$ (*stmt*) *IN*
x (state,l) = exIf ($\mathcal{T}'_c$ (*expr*)) s (state,l);
RETURN x

$\mathcal{T}_s$ (if (*expr*) *stmt*$_1$ else *stmt*$_2$) $\rightsquigarrow$ *LET* x = *FRESH() IN*
LET $s = \mathcal{T}_s$ (*stmt*$_1$) *IN*
LET $t = \mathcal{T}_s$ (*stmt*$_2$) *IN*
x (state,l) = exIfElse ($\mathcal{T}'_c$ (*expr*)) s t (state,l);
RETURN x

Figure 5: $\mathcal{T}_s$: translation rules for MINIC* statements

$\mathcal{T}_s$ (for (*id* = *expr*$_1$; *expr*$_2$ *binop id* ; *id* = *id* - *expr*$_3$) *stmt*$_3$) $\rightsquigarrow$
LET x = *FRESH() IN*
LET y = *FRESH() IN*
LET $f = \mathcal{T}_s$ (*id* = *expr*$_1$) *IN*
LET $s = \mathcal{T}_s$ (*stmt*$_3$; *id* = *id* - *expr*$_3$) *IN*
y (state,l) *id* = if $\mathcal{T}'_e$ (*expr*$_2$) $\mathcal{T}_o$ (*binop*) *id* then y (s (state,l)) (*id* - $\mathcal{T}'_e$ (*expr*$_3$)) else st;
x (state,l) = y (state,l) $\mathcal{T}'_e$ (*expr*$_1$) ;
RETURN $x(f)$

The y function introduces an accumulator variable which is reduced towards a lower bound (which we assume is greater or equal to 0). This reduction is exploited by the Hume analyser tool.

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

$$\begin{array}{lll} \mathcal{T}_s\,(\texttt{for (} stmt_1 \texttt{ ; } expr \texttt{ ; } stmt_2 \texttt{) } stmt_3) & \leadsto & LET\ x = FRESH()\ IN \\ & & LET\ f = \mathcal{T}_s\ (stmt_1)\ IN \\ & & LET\ e = \mathcal{T}_c\ (expr)\ IN \\ & & LET\ s = \mathcal{T}_s\ (stmt_3 \texttt{ ; } stmt_2)\ IN \\ & & \underline{x \texttt{ (state,l) = loop } e\ s \texttt{ (state,l);}} \\ & & RETURN\ x(f) \end{array}$$

where $\mathcal{T}_c$ is a `mybool`-predicate on the state.

4.7 Streams

$$\begin{array}{lll} \mathcal{T}_{printf}\,(\texttt{printf (\%d , } expr \texttt{)})\ \rho & \leadsto & LET\ s = FRESH()\ IN \\ & & LET\ x = FRESH()\ IN \\ & & \rho' = \rho[st \mapsto \rho_{st}\ @\ [(s, \texttt{int 16})]] \\ & & \underline{\texttt{stream } s \texttt{ to "std\_out";}} \\ & & \underline{x \texttt{ (state,l) (} patt\ \rho_{st} \texttt{) = (state,l) (} patt\ \rho_{st} \texttt{ ,}\ \mathcal{T}'_e\ (expr) \texttt{));}} \\ & & RETURN\ \langle x, \rho' \rangle \\ \mathcal{T}_{printf}\,(stmt_1 \texttt{ ; } stmt_2)\ \rho & \leadsto & LET\ \langle f, \rho' \rangle = \mathcal{T}_{printf}\,(stmt_1)\ \rho\ IN \\ & & LET\ \langle g, \rho'' \rangle = \mathcal{T}_{printf}\,(stmt_2)\ \rho'\ IN \\ & & RETURN\ \langle g(f), \rho'' \rangle \\ \\ \mathcal{T}_{scanf}\,(\texttt{scanf (\%d , } expr \texttt{))} & \leadsto & LET\ x = FRESH()\ IN \\ & & LET\ s = FRESH()\ IN \\ & & \underline{\texttt{stream } s \texttt{ from "std\_in";}} \\ & & \underline{x \texttt{ (state,l) } s \texttt{ = assign (}\ \mathcal{T}'_e\ (expr) \texttt{)}\ s \texttt{ (state,l);}} \end{array}$$

Figure 6: $\mathcal{T}_{printf}$ and $\mathcal{T}_{scanf}$: translation rules for MINIC* streams

Figure 6 shows the translation rules for output streams $\mathcal{T}_{printf}$ and an input stream $\mathcal{T}_{scanf}$. Streams are handled in the coordination and the heading and wire declaration of the box containing the function calls require this information. Since a program may have more than one output stream, the $\mathcal{T}_{printf}$ required an environment in the meta-logic, which captures all the required information.

Each of the two output stream rules $\mathcal{T}_{printf}$ returns a pair $\langle x, \rho \rangle$, consisting of the identifier of the generated function, and the new environment ρ. An environment holds one mappings, *st*, which is a list of all the output streams. An empty environment ρ_{empty} has the following property:

$$\rho_{empty_{st}} \quad = \quad []$$

An output stream (`printf` statement in MINIC*) adds another output wire to the box, meaning the environment (*st*) is updated with this extra stream. Moreover, $\mathcal{T}'_e$ is used to translated the value which should be printed. It uses the *patt* function from deliverable 32, which is defined as follows:

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

An input streams works as an assignment, and assigns the read input value s to the location given by *expr* in the `scanf` statement.

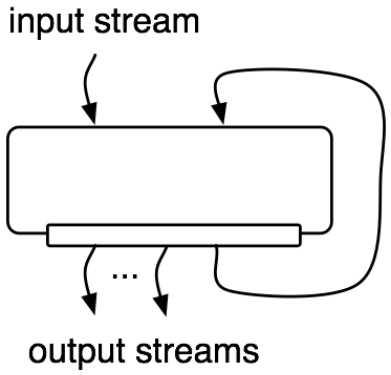


Figure 7: Box diagram of MINIC* program in Hume

4.8 The Full Program

The translation of a MINIC* program into Hume, requires the *head* function from deliverable 35:

$$\begin{array}{lcl} head\ ([id] :: []) & \leadsto & \underline{id\ \texttt{::int 16}} \\ head\ ([id] :: sp) & \leadsto & \underline{id\ \texttt{::int 16 ,}}\ head\ sp \end{array}$$

Figure 8 shows the translation of a full program without an input stream, while Figure 9 shows the translation of a full program with an input stream. These versions differs only slightly. Both rules have an additional *size* variable, which defines the size of the state space vector, and must be provided by the user. Most of the rules are self-contained, while some features requires more explanation: `vecdef` is a built-in Hume function that creates a vector; the *last* function chops the state and `l`, and only the output stream values are returned; finally, the box has a boolean input value ensuring that a box is executed only once, as is the case for a MINIC* program.

5 The translator program

As in deliverable 32, the rules has been implemented in Haskell. The program `c2hume` is called with 2 arguments: the name of the MINIC* file; and a natural number specifying the size of the state space vector:

```
c2hume <program_name>  <vector_size>
```

The resulting Hume program can then be passed through the Hume analyser (`phamc` followed by `art3`) for costing. The program can also be compiled by the Hume compiler (`humec`) or executed directly by the Hume interpreter (`hume`).

6 Example

The following example multiplies two 3×3 matrices, representing using 2 dimensional arrays:

```
#include <stdio.h>
#include <stdlib.h>

main (){
  int **a; int **b; int **c;
```

$\mathcal{T}^e$ (*prelude* `main(){`*decls stmts prints* `}`) *size* $\leadsto$
LET i = *FRESH() IN*
LET o = *FRESH() IN*
LET $last$ = *FRESH() IN*
$\mathcal{T}_{mybool}$
$\mathcal{T}_{data}(decls)$
$\mathcal{T}_{env}(decls)$ `1`
$\mathcal{T}_{HOF}$
`constant maxsize =` *size* `;`
`constant stacksize =` *stacksize decls* `;`
`i_vecfun x = 0;`
$\mathcal{T}_{memory}$
LET $sts = \mathcal{T}_s\ (stmts)$ *IN*
LET $\langle os, \rho \rangle = \mathcal{T}_{printf}\ (prints)\ \rho_{empty}$ *IN*
LET $c_p = \rho_{ostr} \neq []$ *IN*
last `_` (*IF* c_p *THEN* `(` *patt* ρ_{st} `)`) `= (*` (*IF* c_p *THEN* `,` *patt* ρ_{st}) `);`
`box mainprogram`
`in(` i `::` `bool)`
`out(` o `::` `bool` *IF* c_p *THEN* `(` `,` *head* ρ_{st} `"")`
`match`
`(_)` `->` *last* (*IF* c_p *THEN* os) (*sts* `(vecdef maxsize i_vecfun,stacksize+1)));`

`wire mainprogram`
`(mainprogram.`o `initially true)`
`(mainprogram.`i *IF* c_p *THEN* `(` `,` *patt* ρ_{st} `);`

Figure 8: $\mathcal{T}^e$: translation rules for a MINIC* program without input stream

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

a = (int**) malloc(3 * sizeof(int));
b = (int**) malloc(3 * sizeof(int));
c = (int**) malloc(3 * sizeof(int));

for(i = 3; 0 < i; i = i - 1){
  *(a+i-1) = (int*) malloc(3 * sizeof(int));
  *(b+i-1) = (int*) malloc(3 * sizeof(int));
  *(c+i-1) = (int*) malloc(3 * sizeof(int));
}

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

$\mathcal{T}^e$ (*prelude* `main(){`*decls scan stmts prints* `}`) *size* $\leadsto$
LET i = *FRESH() IN*
LET o = *FRESH() IN*
LET s = *FRESH() IN*
LET last = *FRESH() IN*
$\mathcal{T}_{mybool}$
$\mathcal{T}_{data}(decls)$
$\mathcal{T}_{env}(decls)$ `1`
$\mathcal{T}_{HOF}$
`contant maxsize =` *size* `;`
`constant stacksize =` *stacksize decls* `;`
`i_vecfun x = 0;`
$\mathcal{T}_{memory}$
LET $is = \mathcal{T}_{scanf}\,(scan)$ *IN*
LET $sts = \mathcal{T}_s\,(stmts)$ *IN*
LET $\langle os, \rho \rangle = \mathcal{T}_{printf}\,(prints)\ \rho_{empty}$ *IN*
LET $c_p = \rho_{ostr} \neq []$ *IN*
last `_` (*IF* c_p *THEN* `(` *patt* ρ_{st} `)`) `= (*` (*IF* c_p *THEN* `,` *patt* ρ_{st} `)` `);`
`box mainprogram`
`in(` i `::` `bool,` s `::` `int 16)`
`out(` o `::` `bool` *IF* c_p *THEN* `(` `,` *head* ρ_{st} `""` `)` `)`
`match`
`(_,` s `)` `->` *last* ((*IF* c_p *THEN* os) `(`*sts* `(` *is* `(vecdef maxsize i_vecfun,stacksize+1)` s `));`

`wire mainprogram`
`(mainprogram.`o `initially true , program.`s `)`
`(mainprogram.`i *IF* c_p *THEN* `(` `,` *patt* ρ_{st} `)` `);`

Figure 9: $\mathcal{T}^e$: translation rules for a MINIC* program with input stream

```
}
```

Here `a` is multiplied with `b`, and the result is stored in `c`. A matrix has two dimensions, thus it is represented as an array of pointers, i.e. a pointer to a pointer, in MINIC*. Before the multiplication, all elements are allocated space in the heap. Matrix multiplication requires three nested loops and the `i`,`j` and `k` variables are used for this iteration. Here, `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. Note, that two dimensional array projection `a[i][j]` is written `*(*(a+j)+i)` in MINIC*. Further, note that the "strange iteration like `for(i = 2; 0 < i; i = i - 1)`, with `-1` in the array projection, instead of `for(i = 1; 0 <= i; i = i - 1)` and direct projection, is to obtain more accurate costing.

The generated Hume code with a state space vector of 42 locations is shown in Appendix B.

Program	Hume Time Analysis	Hume Time	HTA/HT	C-code Time	HTA/CT
`arrayrotate`	506548	186851	2.71	746	679
`fact`	121527	83659	1.45	1751	629
`fibsum`	211062	133193	1.59	1413	149
`matmult2`	425850	229469	1.86	713	597
`matmult3`	1271170	665363	1.91	2126	598
`mediumarrayrotate`	1092800	378035	2.89	1737	629
`smallarraysearch`	1072090	477408	2.25	2705	396

Table 1: miniC analysis and measurement results

Program	Hume Vector Size	Hume Time Analysis	Hume Time	HTA/HT	C-code Time	HTA/CT
`arraycopy`	23	775498	732158	1.06	1094	708.86
`arrayrotate`	13	936805	303949	3.08	1798	521.03
`fact`	13	690179	301328	2.29	3513	96.46
`matmult2`	27	1130590	500271	2.25	1861	607.52
`mediumarrayrotate`	16	1259360	376552	3.34	2347	536.58
`tinyarraysearch`	15	376552	302224	1.25	2596	145.05

Table 2: miniC* analysis and measurement results

7 Results

To estimate the quality of our analysis, we compare the values it delivers with measurements on a real embedded systems processor, the Renesas M32C. The actual costs worst-case execution costs for the M32C/85U processor have been determined by the aiT tool. While the analysis results are necessarily overestimations, the measurements are significant underestimations of the worst case, because they deal with particular input data.

Table 1 shows the results for miniC from deliverable 32. The final column shows the ratios of Hume analysis predicted worst-case execution times to measured miniC times. The ratios are very similar for the programs `fact`, `matmult2`, `matmult3`, `arrayrotate` and `mediumarrayrotate`.

Table 2 shows comparable results for miniC*. First of all, note that the absolute times for miniC* programs are considerably larger than those of the equivalent miniC programs. This is to be expected as there is an inherent higher overhead of using a language with dynamic memory management. Furthermore, the explicit use of addresses and indirection has added additional artefactual layers of Hume in translation.

More important, there is considerably less consistency in the ratios of miniC* times to Hume predictions suggesting that the translation to Hume is not adequately capturing the implementations of miniC*.

The miniC* source codes for all the programs of Table 2 are listed in Appendix A.

8 Relevant work

As noted in deliverable 32, the translation of imperative to functional languages is 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. The most relevant mechanisation of an imperative language with pointers and dynamic memory management we are familiar with is Tuch's PhD thesis [Tuc08]. Here a C language was formalised in the Isabelle/HOL theorem prover. However, his motivation was correctness verification and not resource analysis.

9 Summary

We have explored the use of the Hume WCET analyses directly with MINIC*, an imperative language with explicit memory allocation and pointers, through translation. Initial results are inconclusive, both absolutely and by comparison with earlier work on translation from MINIC, which lacks pointers and memory allocation.

Because of the introduction of pointers and dynamic memory models in MINIC*, the translation of MINIC* statements and the underlying representation of the state space deviates from that for MINIC. A MINIC program allows integers and arrays of integers, and a MINIC variable is translated into a Hume variable, where an (static) array is represented as a Hume vector. The state-space is then a tuple of all the variables. However, due to the dynamic memory model and the use of pointers, such a "shallow" state representation cannot be used in MINIC*.

Essentially, there is a mismatch between the memory model implicit in the C implementation which underlies MINIC* and our attempt to realise it in translation to Hume. This might be substantially reduced given a formal semantics for C which accurately reflected the implementation and hence could inform the translation.

References

[GML+08] G. Grov, G. Michaelson, H-W. Loidl, C. Herrmann, S. Jost, and K. Hammond. Application to More Traditional Settings. EmBounded Project Deliverable, August 2008. Deliverable D32. 1

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 2nd edition.* Prentice-Hall, 1988. 1

[Mic86] G. Michaelson. Interpreters from functions and grammars. *Computer Languages*, 11(2):85–104, 1986. 8

[Sch86] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn and Bacon, 1986. 8

[Tuc08] Harvey Tuch. *Formal Memory Models for Verifying C Systems Code.* PhD thesis, The University of New South Wales, 2008. 8

A MINIC* source code

A.1 arraycopy.c

```
#include <stdio.h>
#include <stdlib.h>

main (){
  int *src;
  int *res;
```

```
  int cnt;
  res = (int*) malloc(10 * sizeof(int));
  src = (int*) malloc(10 * sizeof(int));
  for(cnt=10;0 < cnt;cnt=cnt-1){
    *(src+cnt-1) = cnt;
  }
  for(cnt=10; 0 < cnt;cnt=cnt-1){
    *(res+cnt-1) =  *(src+cnt-1);
  }
  free(src);
  free(res);
}
```

A.2 arrayrotate.c

```
#include<stdio.h>
#include<stdlib.h>
main (){
  int *arr;
  int rand;
  int cnt;
  arr = (int*) malloc(10 * sizeof(int));
  *(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.3 fact.c

```
#include <stdio.h>
#include <stdlib.h>
main (){
  int res;
  int *aux;
  int cnt;
  aux = (int*) malloc(10 * sizeof(int));
  for(cnt=10;0 < cnt;cnt=cnt-1){
    *(aux+cnt-1) = cnt;
  }
  res = 1;
  for(cnt=10; 0 < cnt;cnt=cnt-1){
    res = *(aux+cnt-1) * res;
  }
  free(aux);
```

```
  printf("%d",res);
}
```

A.4 matmul2.c

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

  a = (int**) malloc(2 * sizeof(int));
  b = (int**) malloc(2 * sizeof(int));
  c = (int**) malloc(2 * sizeof(int));

  for(i = 2; 0 < i; i = i - 1){
    *(a+i-1) = (int*) malloc(2 * sizeof(int));
    *(b+i-1) = (int*) malloc(2 * sizeof(int));
    *(c+i-1) = (int*) malloc(2 * sizeof(int));
  }

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

A.5 mediumarrayrotate.c

```
#include<stdlib.h>
main (){
  int *arr;
  int rand;
  int cnt;
  arr = (int*) malloc(13 * sizeof(int));
  (*(arr+12)) = 13;
  for(cnt=12; 0 < cnt; cnt = cnt - 1){
    (*(arr+(cnt-1))) = (17 * (*(arr+cnt))) % 25;
   }
  for(cnt=12; 6 < cnt;cnt = cnt - 1){
    rand = (*(arr+cnt));
    (*(arr+cnt)) = (*(arr+(12-cnt)));
```

```
    (*(arr+(12-cnt))) = rand;
  }
}
```

A.6 tinyarraysearch.c

```
#include<stdio.h>
#include<stdlib.h>
main (){
  int *arr;
  int rand;
  int cnt;
  int res;
  arr = (int*) malloc(10 * sizeof(int));
  rand = 13;
  res = 1;
  (*(arr+9)) = (rand+5) % 13;
  for(cnt=9; 0 < cnt; cnt = cnt - 1){
    (*(arr+(cnt-1))) = (17 * (*(arr+cnt))) % 13;
   }
  for(cnt=9;0 < cnt;cnt = cnt - 1){
    if((*(arr+(cnt-1))) == rand)
      res = (*(arr+(cnt-1)));
  }
  printf("%d",res);
}
```

B Example Hume code: matmult3.hume

```
data mybool = TT  |  FF;
data var = A  |  B  |  C  |  I  |  J  |  K;

env A = 1;
env B = 2;
env C = 3;
env I = 4;
env J = 5;
env K = 6;

assign loc v (state,l) = ((update state loc v),l);
exIf TT s (state,l) = (s (state,l));
exIf FF _ (state,l) = (state,l);
exIfElse TT s _ (state,l) = (s (state,l));
exIfElse FF _ t (state,l) = (t (state,l));
loop e s (state,l) = (if ((e state) == TT) then (loop e s (s (state,l))) else (state,l));

constant maxsize = 42;
constant stacksize = 6;
i_vecfun x = 0;
afree n l = (if ((n > stacksize) && (n < l)) then n else 0);
alloc n l = (if (n <= (maxsize - l)) then (l + n) else 0);

f1 (state,l) = ((update state (env A) l),(alloc 3 l));
f2 (state,l) = ((update state (env B) l),(alloc 3 l));
```

```
f3 (state,l) = ((update state (env C) l),(alloc 3 l));
f4 (state,l) = (assign (env I) 3 (state,l));
f5 (state,l) = ((update state ((state @ (env A)) + ((state @ (env I)) - 1)) l),(alloc 3 l));
f6 (state,l) = ((update state ((state @ (env B)) + ((state @ (env I)) - 1)) l),(alloc 3 l));
f7 (state,l) = ((update state ((state @ (env C)) + ((state @ (env I)) - 1)) l),(alloc 3 l));
f8 (state,l) = (assign (env I) ((state @ (env I)) - 1) (state,l));
f10 (state,l) = (f8 (f7 (f6 (f5 ((state,l))))));
f9 (state,l) i = (if (0 < i) then (f9 (f10 (state,l)) (i - 1)) else (state,l));
f11 (state,l) = (f9 (state,l) 3);
f12 (state,l) = (assign (env I) 3 (state,l));
f13 (state,l) = (assign (env J) 3 (state,l));
f14 (state,l) = (assign ((state @ ((state @ (env C)) + ((state @ (env J)) - 1)))
     + ((state @ (env I)) - 1)) 0 (state,l));
f15 (state,l) = (assign (env K) 3 (state,l));
f16 (state,l) = (assign ((state @ ((state @ (env C)) + ((state @ (env J)) - 1)))
      + ((state @ (env I)) - 1)) ((state @ ((state @ ((state @ (env C)) + ((state @ (env J)) - 1)))
      + ((state @ (env I)) - 1))) + ((state @ ((state @ ((state @ (env A)) + ((state @ (env K)) - 1)))
      + ((state @ (env I)) - 1))) * (state @ ((state @ ((state @ (env B))
      + ((state @ (env J)) - 1))) + ((state @ (env K)) - 1))))) (state,l));
f17 (state,l) = (assign (env K) ((state @ (env K)) - 1) (state,l));
f19 (state,l) = (f17 (f16 ((state,l))));
f18 (state,l) k = (if (0 < k) then (f18 (f19 (state,l)) (k - 1)) else (state,l));
f20 (state,l) = (f18 (state,l) 3);
f21 (state,l) = (assign (env J) ((state @ (env J)) - 1) (state,l));
f23 (state,l) = (f21 (f20 (f15 (f14 ((state,l))))));
f22 (state,l) j = (if (0 < j) then (f22 (f23 (state,l)) (j - 1)) else (state,l));
f24 (state,l) = (f22 (state,l) 3);
f25 (state,l) = (assign (env I) ((state @ (env I)) - 1) (state,l));
f27 (state,l) = (f25 (f24 (f13 ((state,l)))));
f26 (state,l) i = (if (0 < i) then (f26 (f27 (state,l)) (i - 1)) else (state,l));
f28 (state,l) = (f26 (state,l) 3);
f29 _ = *;


box mainprogram
 in (i :: bool)
 out (o :: bool)
match
_ -> (f29 (f28 (f12 (f11 (f4 (f3 (f2 (f1 ((vecdef maxsize i_vecfun),(stacksize + 1)))))))))));
wire mainprogram (mainprogram.o initially true) (mainprogram.i);
```