Bounded-Construction-Types for Functional Reactive Programming

Akihiko Yokoyama
akihiko@psg.c.titech.ac.jp
Department of Computer Science
Tokyo Institute of Technology
Tokyo, Japan

Sosuke Moriguchi
chiguri@acm.org
Department of Computer Science
Tokyo Institute of Technology
Tokyo, Japan

Takuo Watanabe
takuo@acm.org
Department of Computer Science
Tokyo Institute of Technology
Tokyo, Japan

ABSTRACT

We introduce a new type system to Emfrp, a functional reactive programming language designed for resource-constrained embedded systems. To ensure such property that the language can statically determine the amount of runtime memory and guarantee the termination of reactive actions, it disallows the use of recursive data types and functions. However, such restrictions often impose unnatural representations of data structures and algorithms used in various applications. Our new type system, named Bounded-Construction-Types (BCTs), enhances Emfrp by introducing recursive data types with size annotations yet the improved language keeps the static property mentioned above. In this paper, we formalize Emfrp extended with BCTs, present the algorithm for statically computing the runtime memory bounds, and prove its soundness.

1 INTRODUCTION

Functional Reactive Programming (FRP) is a programming paradigm that supports the effective description of reactive systems such as embedded systems and GUIs. It employs the notion of *time-varying values*, which abstract values that change continuously or discretely over time. The idea of FRP was first introduced in Fran [5], an interactive animation library for Haskell. After that, various FRP libraries, languages, and DSLs have been proposed, for example, Yampa [13], Elm [4] (version 0.16 or earlier [3]), and so on.

Sawada *et al.* designed and implemented Emfrp [18], a statically typed pure FRP language for small-scale embedded systems. Polling and callbacks (and their mixtures) are common patterns used in describing reactive behaviors in small-scale embedded systems. However, they often complicate the code by, for example, splitting the code's control flow into multiple small pieces. Some example applications demonstrates that the simple FRP mechanism in Emfrp is useful in programming embedded systems. The Emfrp compiler emits platform-independent C code runnable in several resource-constrained environments including microcontrollers. For example, their demonstrations include the controller of a reverse pendulum robot equipped with an 8-bit microcontroller with 2.5KB RAM and 32KB flash[1].

In a reactive system running in a resource-constrained environment, it is important to keep the program running with a bounded memory space and to continue reactions (*i.e.*, to guarantee the termination of each reaction to keep the system responsive). Programs written in Emfrp are guaranteed to satisfy these two properties. For this reason, Emfrp imposes several language design constraints (Section 2.3). One of them is the prohibition of recursive definitions in functions and data types. Even under these constraints, we can write many application programs. However, the problem is that the natural representation of common data structures such as strings, lists, and trees becomes difficult.

In this paper, we propose a method to relax the restriction on recursive definitions in Emfrp while satisfying the two properties. We introduce recursive data types with parameters about the size of the structure, named Bounded-Construction-Types, and primitive recursive functions whose arguments are structurally reduced on recursive calls. In addition to the usual type checking, the object size constraints are checked to detect programs whose objects are increasing continuously or whose node update process is not guaranteed to stop. We also propose an algorithm to determine the amount of memory used at runtime.

The main contributions of this paper are the following:

- We introduced Emfrp$^{\mathrm{BCT}}$, which is an extension of Emfrp with Bounded-Construction-Types, and presented a useful program example.
- We formalized Emfrp$^{\mathrm{BCT}}$, and developed determination algorithm for upper bounds of memory used.
- Then, we proved their soundness.

The rest of this paper is organized as follows. Section 2 introduces Emfrp and the problems arose from the language restrictions that motivated this work. Section 3 describes the extension of Emfrp with Bounded-Construction-Types and presents two examples. Section 4 presents the syntax, operational semantics, type system, and used memory determination algorithm and show their soundness. Section 5 shows overview of related work and Section 6 concludes the paper.

2 BACKGROUND

Emfrp[18] is a strongly-typed functional reactive programming language designed for resource-constrained embedded systems. This section shows overview of the language and discusses issues that motivate this study.

2.1 Overview of Emfrp

Listing 1 is a simple Emfrp program that calculates the position of a two-wheel robot. This example is an adapted version of one written in Yampa (an FRP library for Haskell)[13]. The program takes four inputs (v_l, v_r, θ, and t) and gives a single output (x). They are the speeds of left and right wheels (v_l and v_r), the angle between the robot's orientation and the x-axis (θ), the elapsed time (t), and the x-coordinate of the robot (x). Note also that they are time-varying

[1] https://github.com/psg-titech/emfrp_samples/

```
1  module RobotPos    # module name
2  in  vl : Float,    # left wheel speed (m/sec)
3      vr : Float,    # right wheel speed (m/sec)
4      theta : Float, # angle from x-axis (rad)
5      t(0) : Int     # elapsed time (msec)
6  out x : Float      # x-position of the robot (m)
7  use Std            # library name
8
9  # a small amount of elapsed time (sec)
10 node dt = (t - t@last) / 1000.0
11
12 # x-position of the robot (m)
13 node init[0.0] x = x@last + (vr+vl) * cos(theta) * dt/2
```

Listing 1: A Two-Wheel Robot in Emfrp

(*i.e.*, functions of t) and $x(t) = \frac{1}{2}\int_0^t (v_l(u) + v_r(u))\cos(\theta(u))du$ holds[13].

An Emfrp program is composed of units called *modules*. A module declares its name, input/output nodes, and library names required, and defines types, constants, functions, and nodes. *Nodes* are continuous or discrete time-varying values in Emfrp. They are classified into three types: *input* and *output* nodes supposed to have connections with external devices and *internal* nodes having no such connections. The program in Listing 1 consists of a single module named `RobotPos`. It declares `vl`, `vr`, `theta`, `t` as the input nodes and `x` as the output node (lines 2–6). They correspond to v_l, v_r, θ, t, and x, respectively.

The values of input nodes are determined by the external devices to which they are connected. In Listing 1, the values of `vl` and `vr` are obtained from the rotary encoders connected to the wheels, and the value of `theta` can be calculated from the measurements of the direction (azimuth) sensor. Also, the reading of the system's timer gives the value of `t`. Note that, in the example, we use adjusted values instead of raw sensor readings for simplicity.

The values of output and internal nodes are determined with *node definitions*. A node definition has the syntax `node` n `=` e or `node init[`c`]` n `=` e, where n is the node name, and e is an expression that gives the value of the node. The optional `init[`c`]` specifies the initial value (described later) of the node as c. Listing 1 has definitions of nodes `dt` (line 10) and `x` (line 13).

A node name followed by the operator `@last` represents the *previous value* of the node, that is, the value at the previous moment. In Listing 1, the definitions of `dt` and `x` use the operator. In the example, we express the amount of change and cumulative values by using `@last`. In this way, we can concisely express the value represented by the time integration without using special functions such as `integral` in Yampa[13]. The *initial value* of a node must be specified if the node's previous value is referenced in the program. The initial value is used as the previous value at the start of the program execution. For example, in Listing 1, the definition of `x` specifies its initial value using `init[0.0]` (line 13). For input nodes such as `t`, the initial value is specified as `t(0)` (line 5).

2.2 Execution Model

We briefly describe the execution model of Emfrp in this subsection. Let n and n' be nodes defined in the body of a module. We say that n depends on n' if n' appears in the definition of n without `@last`. We can construct a directed graph of nodes where the dependency is represented as a directed edge from n' to n. Emfrp requires that the graph is a directed acyclic graph (DAG), and every input node does not have incoming edges. As will be discussed later in Section 2.3.1, the DAG is statically determined and does not change at runtime.

The Emfrp compiler topologically sorts the DAG and generates a sequence of nodes. In the case of Listing 1, the sequence can be (`t`, `dt`, `vl`, `vr`, `theta`, `x`). The values of the nodes are updated along the sequence. A single update cycle (in the example, from `t` to `x`) is called an *iteration*. The runtime system achieves reactive behavior by repeating the iteration. Note that n`@last` is implemented to have the value of n in the last (previous) iteration.

2.3 Language Restrictions and Issues

As mentioned in Section 1, Emfrp is designed to write programs that run on resource-constrained systems such as microcontrollers. For this purpose, the language (1) does not treat nodes (time-varying values) as first-class values, and (2) does not allow recursive definitions of functions and data structures. The restrictions realize that the amount of memory to be used at runtime can be statically determined, and the termination of the iteration is guaranteed.

2.3.1 Nodes Are Not First-Class. Emfrp does not treat nodes (time-varying values) as first-class data. That is, a node is not assigned to a variable, nor is it an argument or return value of a function. When a node name occurs as a function argument, what is passed is the node's value, not the node itself. Also, the language does not provide mechanisms that dynamically create or delete nodes. Nodes are defined only using keyword `node` or declared in the module header. Furthermore, there is no means to define higher-order nodes, which take other nodes as their values. Thus the number of nodes and the dependency relation between nodes do not change at runtime. The constraints make it possible to place the data structures representing nodes in a static area rather than a heap.

2.3.2 Recursive Definitions Are Not Allowed. Emfrp allows us to define functions and algebraic data types in the module body but prohibits recursions in them. The reason is to guarantee the termination of every iteration (updating cycle of nodes) and to determine the amount of memory to be used at runtime.

Emfrp is a purely functional language so that it has no indeterminate loop constructs like while statements. Thus, prohibiting recursive functions implies the termination of the evaluation of expressions. So every iteration should terminate. Moreover, by prohibiting recursive definitions in algebraic data types, the Emfrp compiler can determine the size of data. Since there are no recursive functions, the compiler can also determine the size of the call stack. Thus, the total amount of the memory used at runtime can be determined.

However, the restriction may impose unnatural implementations of common data structures such as lists and trees that are useful even in resource-constrained embedded systems. In this study, we propose Bounded-Construction-Types (BCT) to introduce recursive data types and functions to Emfrp. BCT allows us to define recursive data types with explicit constraints on their sizes that determine the amount of memory usage at runtime.

```
module DupCheck
in  reset(false) : Bool # reset signal
    v(0)         : Int  # input value
out detect : Bool
use Std

type OptI = N | S(Int) # None, Some

func chk(x : Int, a : OptI): Bool =
  a of: N -> false | S(y) -> x = y

node init [(N, N, N, N)] history:
  (OptI,OptI,OptI,OptI) =
    if reset then (N, N, N, S(v))
    else
      history@last of
        (a1, a2, a3, a4) -> (a2, a3, a4, S(v))

node init [false] detect: Bool =
  history@last of: (a1, a2, a3, a4) ->
    chk(v,a1)||chk(v,a2)||chk(v,a3)||chk(v,a4)
```

Listing 2: Duplication Checking Module in Emfrp

3 BOUNDED-CONSTRUCTION-TYPES

Bounded-Construction-Types (BCTs) is a type system that specifies the data types of objects along with their sizes. Through two examples, this section shows overview of Emfrp$^{\mathrm{BCT}}$, an extension of Emfrp with BCTs. The type system guarantees that a well-typed program is consistent wrt types and object sizes. We formally define the notion of sizes in Section 4.

3.1 Overview of Emfrp$^{\mathrm{BCT}}$

Emfrp$^{\mathrm{BCT}}$ is an extension of Emfrp with BCTs. Its execution model is the same as that of Emfrp. The main difference at runtime is that each data object of BCT has actual size information. BCTs determine the maximum object size of nodes and of `@last` nodes, and the memory size estimation algorithm (described in Section 4.4) gives the amount of temporary memory used. Thus it is possible to update nodes in the constant memory space. Moreover, the program can continue to run in constant space by doing garbage collection for temporary objects between each iterations.

In Emfrp$^{\mathrm{BCT}}$, we can define recursive data structures, such as tree of integer by writing `type Tree = Leaf | Node(Tree, Int, Tree)`. `Leaf` and `Node` are constructors. At the definition, we do not write size parameter, but we annotate types with size parameter to use it. We write `Tree[`ψ`]` for type of a `Tree` of size ψ. ψ is the size parameter. In this situation, value `Leaf` has `Tree[1]` type, and `Node(Node(Leaf,2,Leaf),1,Leaf)` has `Tree[5]` type. Generally, if variable `left` has `Tree[`α`]` type and variable `right` has `Tree[`β`]` type, term `Node(left,1,right)` has type `Tree[`$1+\alpha+\beta$`]`.

`adj` and **`fit`** expression is introduced by the extension. These allow us to coerce size parameter of an expression. e **`adj`**`[`ψ`]` is a conversion expression that enlarges the size parameter of e. It is checked statically that ψ is greater then the size of e. So this conversion is always successful at runtime. For example, the expression `Leaf` **`adj`**`[5]` converts `Leaf` of size 1 to `Leaf` of size 5. This means that only one element is stored in a memory area of five elements at most. **`fit`** e **`to`** `x:P[`k`] ->` e_1 `|` **`fail`** `->` e_2 is an expression that shrinks the size of e to a constant size. This conversion is performed by runtime size information. Thus, the conversion process may fail. If the conversion succeeds, the value of e is bound to new variable x whose size is k and first clause e_1 is executed; otherwise **`fail`** clause e_2 is executed.

```
module DupCheck
in  reset  : Bool init false # reset signal
    v      : Int  init 0     # input value
out detect : Bool

type L = N | C(Int, L) # Nil, Cons

func insert(x: Int, l: L[m]): L[m+1] where {m > 0} [m] =
  case l return L[m+1] of
  | N -> C(x, N) adj[m+1]
  | C(h: Int, t: L[n]) -> C(h,insert(x,t))

func search(x: Int, l: L[m]): Bool where {m > 0} [m] =
  case l return Bool of
  | N -> false
  | C(h: Int, t: L[n]) ->
      if x = h then true else search(x,t)

func tail(l: L[m]): L[m-1] where {m-1 > 0} =
  case l return L[m-1] of
  | N -> N adj[m-1]
  | C(h: Int, t: L[n]) -> t

node history: L[5] init (N adj[5]) =
  if reset then C(v, N) adj[5]
  else
    fit history@last to
    | hl: L[4] -> insert(v,hl)
    | fail -> insert(v,tail(history@last))

node detect: Bool init false = search(x,history@last)
```

Listing 3: Duplication Checking Module using List in Emfrp$^{\mathrm{BCT}}$

`case` expression allow us to destruct data structures. Each branch of **`case`** expression introduce new size variables (*e.g.*, Listing 3 line 11). The result types of each branch must be the same τ by annotating **`return`** τ.

We can now define first-order primitive recursive functions (*e.g.*, Listing 3 line 8-11). If the argument type is BCT, a new size variable that can be used in the function is introduced. **`where`** clause describes the arithmetic constraints that should be satisfied when the function is called (*e.g.*, **`where`** `{m > 0}`). This constraint is an equality or an inequality between size parameters. When the function is called, the size parameter of the actual argument is assigned to the size variable of the formal argument, and this constraint is checked statically. In the case of a recursive function definition, a sequence of size variables (*e.g.*, `[m]`, `[n, m]`) is required after **`where`** clause. The sum of these size variables is statically checked to be decreasing on a recursive call. Passing this check guarantees the termination of the recursive call.

3.2 Example 1: Duplication Checking Module

The first example is `DupCheck` module. This module checks duplication of input values within past 4 iteration loops. Listing 2 shows the Emfrp source code for `DupCheck`. The history node of input value is represented by tuple of optional integer and has sequence of `N` as initial values. Updating nodes, the tuple is used as a queue and the input value is appended to the last value of the history node by shifting it. We decompose the tuples and compare each element when searching for input values in the history. If you want to change the number of elements in the history, you need

```
# The sum of the top 10 of the input data
module Top10Sum
in  x: Int      # input value
out y: Int      # sum

# Leftist Heap
type Heap = E               # terminal
     | T(Int,Int,Heap,Heap) # (rank,v,left,right)
......  # omitted
# sum of values in heap tree
func sumHeap(h: Heap[n]): Int where {n>0} =
  case h return Int of E -> 0
  | T(r,x,a:Heap[p],b:Heap[q])-> x+sumHeap(a)+sumHeap(b)

node h: Heap[21] init (E adj[21]) =  # heap tree node
  fit h@last to hl: Heap[19] -> insert(x, hl)
  | fail -> if x <= findMin(h@last) then h@last
            else insert(x, delMin(h@last))

node y: Int init 0 = sumHeap(h)      # sum of heap node
```

Listing 4: Ranking Module using Leftist Heap Tree

to rewrite the tuple decomposition in the element addition and detection processes. If the number of elements is large, it may be a heavy burden for programmers and may embed bugs with rewriting. Listing 3 shows the Emfrp$^{\mathrm{BCT}}$ code for the same module. The program uses the size limited list `L` defined by BCTs to represent the history of the input. The size of the list is 5 in order to keep 4 past input values and `N` at the end of the list. We define an recursive function to insert a value to the end of the list. The history node is updated by the function and **`fit`** expressions. The conversion of **`fit`** expression succeeds if the history length is less than or equal to three. If not, remove the head of the history node and insert the input value to the end. When changing the number of elements in the history, we can flexibly deal with it by appropriately rewriting the size parameters of the history node and **`fit`** expression.

This example shows how BCTs can improve the extensibility of the program. However, it should be noted that the amount of memory required at runtime is larger than that of Listing 2 because of `insert` function. In Listing 3, all type annotations of the case handlers are described, but they can be omitted if they are not needed.

3.3 Example 2: Ranking Module

The second example is `Top10Sum` module (Listing 4, full version is available Appendix A). This is the module that outputs the sum of up to the 10th largest values received as input. The module has up to 10 values using leftist heap [17], and update their contents in each iteration. Since the shape of the heap tree is determined dynamically, it is not possible to fix the data placement statically. In Emfrp, multiple values are represented by a tuple-based data type. Such a way of representing data cannot represent a structure in which the arrangement of contents is unspecified at runtime. By using a data structure that can represent the absence of a value, such as `Option` type, those data structure can be achieved but becomes an unnatural representation. Using BCTs, we can handle structures with unspecified data layout at execution time, although the size is bounded. This is an example of how BCTs improve the expressivity of a Emfrp program.

Size Constraints

$$k \in \mathbb{N} \quad \delta \in \mathit{SizeVar}$$
$$C \in \mathit{Constraint} ::= \top \mid \mathcal{A} \mid \forall\delta.C \mid C \wedge C' \mid C \to C'$$
$$\mathcal{A} \in \mathit{ArithCon} ::= \psi = \psi' \mid \psi < \psi' \mid \psi \le \psi'$$
$$\psi \in \mathit{SizeParam} ::= k \mid \delta \mid \psi + \psi' \mid \psi - \psi'$$

Types

$$\rho \in \mathit{TypeName}$$
$$\tau^\sharp \in \mathit{VarType} ::= \mathcal{B} \mid \rho^\delta$$
$$\mathcal{B} \in \mathit{BaseType} ::= \texttt{Bool} \mid \texttt{Int}$$
$$\tau^C \in \mathit{ConstType} ::= \mathcal{B} \mid \rho^k$$
$$T^\rho \in \mathit{ElemType} ::= \mathcal{B} \mid \rho \mid \rho'^k \quad (\rho' \neq \rho)$$
$$\tau \in \mathit{Type} ::= \mathcal{B} \mid \rho^\psi$$

Expressions

$$i \in \mathit{Integer} \quad x \in \mathit{Var} \quad N \in \mathit{Node} \quad f, g \in \mathit{Function}$$
$$c \in \mathit{Constant} ::= \texttt{true} \mid \texttt{false} \mid i \quad \chi \in \mathit{ConstructorLabel}$$
$$e \in \mathit{Expr} ::= c^{\mathcal{B}} \mid x \mid N \mid N\texttt{@last} \mid \texttt{let}\ x = e\ \texttt{in}\ e$$
$$\mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ op^{(\mathcal{B}_1,\mathcal{B}_2)\to\mathcal{B}}\ e$$
$$\mid f(\overrightarrow{e}) \mid \chi(\overrightarrow{e}) \mid \texttt{case}\ e\ \texttt{return}\ \tau\ \texttt{of}\ \overrightarrow{branch}$$
$$\mid e\ \texttt{adj}\ [\psi] \mid \texttt{fit}\ e\ \texttt{to}\ x{:}\rho^k \to e \mid \texttt{fail} \to e$$
$$branch \in \mathit{Branch} ::= \chi(\overrightarrow{x:\tau^\sharp}) \to e \qquad ce \in \mathit{ConstExpr} \subset \mathit{Expr}$$

Module Definitions

$$M \in \mathit{Module} ::= (\mathcal{T}, \mathcal{F}, \mathcal{I}, \mathcal{N}, \mathit{Init})$$
$$\mathcal{I} ::= \overrightarrow{N : \tau^C} \quad \text{(Input Nodes)}$$
$$\mathcal{T} ::= \cdot \mid \mathcal{T}, \rho \mapsto \chi(\overrightarrow{T^\rho}) \quad \text{(Type Definitions)}$$
$$\mathcal{N} ::= \cdot \mid \mathcal{N}, N \mapsto (\tau^C, e) \quad \text{(Node Types/Updates)}$$
$$\mathcal{F} ::= \cdot \mid \mathcal{F}, f \mapsto (\overrightarrow{x:\tau^\sharp}):\tau\ \texttt{where}\ \{\overrightarrow{\mathcal{A}}\}[\overrightarrow{\delta}] = e \quad \text{(Function Definitions)}$$
$$\mathit{Init} ::= \cdot \mid \mathit{Init}, N \mapsto ce \quad \text{(Initial Values)}$$

Recursion Indices

$$I(\chi) = \{i \mid i \in 1 \ldots n, T_i^\rho = \rho\} \quad \text{where} \quad \chi(T_1^\rho, \ldots, T_n^\rho) \in \mathcal{T}(\rho)$$

Figure 1: Syntax of Emfrp$^{\mathrm{BCT}}$

4 FORMALIZATION

4.1 Syntax

The syntax of Emfrp$^{\mathrm{BCT}}$ is shown in Fig. 1. The syntax defined here is almost the same as the concrete syntax used in Section 3, but there are some minor differences. For example, the type must be specified not only in the parameters of the function but also in all branches of case-expressions. Size parameters are written superscript of the type name, not in the [] (e.g., `L[n]` is written as $\texttt{L}^{\texttt{n}}$). The module in the concrete syntax has **`out`** nodes for declaring the outputs, but in the syntax here, the module does not include **`out`** nodes.

We make three kinds of limitations of the syntax. First, let $>_{\mathcal{N}}$ be an ordering on nodes such that for any node N, $N >_{\mathcal{N}} N'$ if N' occurs in the update expression of N (the second elements of $\mathcal{N}(N)$). The transitive closure $>^*_{\mathcal{N}}$ should be a strict partial order. Second, let $\geq_{\mathcal{F}}$ be an ordering on functions such that for any function f, $f \geq_{\mathcal{F}} g$ if g occurs in the body of f in $\mathcal{F}$. The transitive closure

Values

$l \in \mathit{Location}$

$v \in \mathit{Value} \quad ::= c \mid \chi[k](\overrightarrow{l})$

Environments

$\Gamma ::= \cdot \mid \Gamma, N : \tau \mid \Gamma, x : \tau$ (Type Env)

$E ::= \cdot \mid E, x \mapsto l$ (Evaluation Env)

$H ::= \cdot \mid H, l \mapsto v$ (Heap)

$\mathcal{L} ::= \cdot \mid \mathcal{L}, N \mapsto l \mid \mathcal{L}, N\texttt{@last} \mapsto l$ (Node Locations)

$\Delta ::= \cdot \mid \Delta, \delta \mapsto k$ (Size Parameter Env)

Figure 2: Values and Environments

$$\boxed{\tau \sim \tau' \qquad T^{\rho} \sim \tau'}$$

$$\begin{aligned} \mathcal{B} \sim \mathcal{B} &= \top \\ \rho^{\psi} \sim \rho^{\psi'} &= (\psi = \psi') \\ \rho \sim \rho^{\psi'} &= (\psi > 0) \end{aligned}$$

$$\boxed{\tau\downarrow}$$

$$\begin{aligned} \mathcal{B}\downarrow &= 0 \\ \rho^{\psi}\downarrow &= \psi \end{aligned}$$

$$\boxed{[\overrightarrow{\delta \mapsto \psi}]\psi \qquad [\overrightarrow{\delta \mapsto \psi}]\mathcal{A} \qquad [\overrightarrow{\delta \mapsto \psi}]\tau}$$

$$\begin{aligned} [\overrightarrow{\delta \mapsto \psi}]\mathcal{B} &= \mathcal{B} \\ [\overrightarrow{\delta \mapsto \psi}]\rho^{\psi'} &= \rho^{[\overrightarrow{\delta \mapsto \psi}]\psi'} \\ [\overrightarrow{\delta \mapsto \psi}](\psi \circ \psi') &= [\overrightarrow{\delta \mapsto \psi}]\psi \circ [\overrightarrow{\delta \mapsto \psi}]\psi' \\ &\quad (\text{if } \circ \in \{+, -, =, <, \leq\}) \\ [\overrightarrow{\delta \mapsto \psi}]k &= k \\ [\delta_1 \mapsto \psi_1, \ldots, \delta_n \mapsto \psi_n]\delta &= \delta \quad (\text{if } \delta \neq \delta_i \text{ for any } i) \\ [\delta_1 \mapsto \psi_1, \ldots, \delta_n \mapsto \psi_n]\delta &= \psi_i \quad (\text{if } \delta = \delta_i) \end{aligned}$$

$$\boxed{[\overrightarrow{\tau^{\sharp}} \mapsto \overrightarrow{\tau}]}$$

$$\begin{aligned} &[\tau_1^{\sharp}, \ldots, \tau_n^{\sharp} \mapsto \tau_1, \ldots, \tau_n] = \\ &\quad [\{\, \delta \mapsto \psi \mid 1 \leq i \leq n, \tau_i^{\sharp} \sim \tau_i = (\delta = \psi) \,\}] \\ &\quad (\forall i \in 1 \ldots n, \tau_i^{\sharp} \sim \tau_i \text{ is defined}) \end{aligned}$$

$$\boxed{ev_{\Delta}[[\psi]]}$$

$$\begin{aligned} ev_{\Delta}[[k]] &= k \\ ev_{\Delta}[[\delta]] &= \Delta(\delta) \\ ev_{\Delta}[[\psi + \psi']] &= ev_{\Delta}[[\psi]] + ev_{\Delta}[[\psi']] \\ ev_{\Delta}[[\psi - \psi']] &= ev_{\Delta}[[\psi]] - ev_{\Delta}[[\psi']] \\ ev_{\Delta}[[\overrightarrow{\delta}]] &= \textstyle\sum_{\delta \in \overrightarrow{\delta}} ev_{\Delta}[[\delta]] \end{aligned}$$

Figure 3: Auxiliary operators for types and constraints: unification for types ($\sim$), size parameter for types ($\downarrow$), size substitution (over types), and evaluation rules of size parameter.

$\geq^{*}_{\mathcal{F}}$ should be a partial order. Finally, any node should not occur in function bodies.

4.2 Operational Semantics

Here we define the operational semantics of the expression in Emfrp$^{\mathrm{BCT}}$. Since Emfrp$^{\mathrm{BCT}}$ is concerned with resources used in computation, its semantics specifies the amount used for operations that require various resources. Resources articulated in the semantics include references to local variables, heaps to store data and call stacks to memorize function call contexts.

The semantics is expressed as $[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e \Downarrow_u l; [t']H'$ (Fig. 4), where E is the local variable environment, s is the free space of the local variable, H is the heap at the start of the expression evaluation, t is the free space of the heap at this time, e is the expression to be evaluated, u is the rest of the call stack, l is the location where the evaluation result of e is stored, H' is the heap after the expression evaluation, and t' is the free space at the end. Also, we have declared types ($\mathcal{T}$) and functions ($\mathcal{F}$) in the module as resources that do not change during an evaluation, and a reference of nodes ($\mathcal{L}$) where the value of the node is stored. The domain of $\mathcal{L}$ is a union of the domain of $\mathcal{N}$ and I, i.e., nodes defined in the module and input nodes.

The result of the evaluation of an expression is not a direct value, but a *location* (a pointer) of the heap, and the evaluation stores a value in the heap. Since the heap changes according to the evaluation of the expression, the heaps before and after the evaluation are made explicit in the semantics. During the evaluation of the expression, the values stored in the heap are not changed or collected as garbages, so the use of the heap increases as the evaluation of the expression progresses.

Let-expressions and case-expressions create new variable bindings, so we have to store the location as a result of evaluating the variable. An environment for variables also consumes resources, so make it clear how much has been conserved. However, this environment, unlike the heap, is a resource that only consumes in the extents of the variables, so the changes are reverted each time the evaluation of the expression finishes. A function call switches the environments and consumes the call stack to preserve the context.

The free spaces of the variable environment, heap, and call stack used in semantics shall always be greater than or equal to 0. A negative value shall not be taken, and if it becomes negative, it shall be considered stuck. Also, these semantics use some partial functions as auxiliary functions: when a function is applied to an argument that is not in its domain, it is assumed that the semantics is stuck.

The size variables specific to Emfrp$^{\mathrm{BCT}}$ are not involved in the evaluation of the function at all, only the constants that represent the size of the constructor and the size of the fit-expression are used. Therefore, the size variable does not consume any resources when executing Emfrp$^{\mathrm{BCT}}$.

4.3 Type System

Here, we define a type check for an expression. Type judgement is described as $\Gamma \vdash^{\mathcal{T};\mathcal{F}}_{f} e : \tau \mid C$, where f is a function name, Γ is a type variable environment, and C is a constraint for size parameters and recursiveness of function calls. The inference rules of type judgment are defined as Fig. 5.

The type system focuses on the sizes of values, but unlike semantics, it does not take into account resources. Without constraints for

$\boxed{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e \Downarrow_u l; [t']H'}$

$$\frac{l \notin dom(H)}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} c^{\mathcal{B}} \Downarrow_n l; [t-1](H, l \mapsto c)} \text{(E-CONST)} \qquad \frac{E(x) = l}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} x \Downarrow_n l; [t]H} \text{(E-VAR)}$$

$$\frac{\mathcal{L}(N) = l}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} N \Downarrow_n l; [t]H} \text{(E-NODE)} \qquad \frac{\mathcal{L}(N\texttt{@last}) = l}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} N\texttt{@last} \Downarrow_n l; [t]H} \text{(E-ATLAST)}$$

$$\frac{\left[\ [s]E \mid [t_{i-1}]H_{i-1} \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_i \Downarrow_u l_i; [t_i]H_i\ \right]_{i\in 1\ldots n} \qquad \chi(T^{\rho}_1, \ldots, T^{\rho}_n) \in \mathcal{T}(\rho)}{[H_n(l_i) = \chi'[k_i](\ldots)]_{i\in I(\chi)} \qquad k = 1 + \sum_{i\in I(\chi)} k_i \qquad l \notin dom(H_n)} \text{(E-CTOR)}$$
$$[s]E \mid [t_0]H_0 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \chi(e_1, \ldots, e_n) \Downarrow_u l; [t_n - 1](H_n, l \mapsto \chi[k](l_1, \ldots, l_n))$$

$$\frac{\begin{array}{c}[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e \Downarrow_u l_1; [t_1]H_1 \qquad H_1(l_1) = \chi_a[k](l'_1, \ldots, l'_{ar_a}) \\ 1 \le a \le n \qquad [s-m](E, x_{a1} \mapsto l'_1, \ldots, x_{ar_a} \mapsto l'_{ar_a}) \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_a \Downarrow_u l_2; [t_2]H_2\end{array}}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{case}\ e\ \texttt{return}\ \tau\ \texttt{of}\ \{\chi_i(x_{i1}:\tau_{i1}, \ldots, x_{ir_i}:\tau_{ir_i}) \to e_i\}_{i\in 1\ldots n} \Downarrow_u l_2; [t_2]H_2} \text{(E-CASE)}$$

$$\frac{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \qquad [s-1](E, x \mapsto l_1) \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_2 \Downarrow_u l_2; [t_2]H_2}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow_u l_2; [t_2]H_2} \text{(E-LET)}$$

$$\frac{\begin{array}{c}\left[\ [s-n]E \mid [t_{i-1}]H_{i-1} \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_i \Downarrow_u l_i; [t_i]H_i\ \right]_{i\in 1\ldots n} \\ \mathcal{F}(f) = (x_1:\tau_1, \ldots, x_n:\tau_n):\tau\ \texttt{where}\ \{\overrightarrow{\mathcal{A}}\}[\overrightarrow{\delta}] = e \\ [s-n](x_1 \mapsto l_1, \ldots, x_n \mapsto l_n) \mid [t_n]H_n \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e \Downarrow_{u-1} l; [t']H'\end{array}}{[s]E \mid [t_0]H_0 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} f(e_1, \ldots, e_n) \Downarrow_u l; [t']H'} \text{(E-CALL)}$$

$$\frac{\begin{array}{c}[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \qquad H_1(l_1) = \texttt{true} \\ [s]E \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_2 \Downarrow_u l_2; [t_2]H_2\end{array}}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow_u l_2; [t_2]H_2} \text{(E-IF-THEN)}$$

$$\frac{\begin{array}{c}[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \qquad H_1(l_1) = \texttt{false} \\ [s]E \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_3 \Downarrow_u l_3; [t_3]H_3\end{array}}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow_u l_3; [t_3]H_3} \text{(E-IF-ELSE)}$$

$$\frac{\begin{array}{c}[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \qquad [s]E \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_2 \Downarrow_u l_2; [t_2]H_2 \\ H_2(l_1) = v_1 \qquad H_2(l_2) = v_2 \qquad v = op(v_1, v_2) \qquad l \notin dom(H_2)\end{array}}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1\ op^{(\mathcal{B}_1,\mathcal{B}_2)\to\mathcal{B}}\ e_2 \Downarrow_u l; [t_2 - 1](H_2, l \mapsto v)} \text{(E-OP)}$$

$$\frac{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e \Downarrow_u l; [t']H'}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e\ \texttt{adj}[\psi] \Downarrow_u l; [t']H'} \text{(E-ADJ)}$$

$$\frac{\begin{array}{c}[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \\ H_1(l_1) = \chi[k'](\overrightarrow{l'}) \qquad k' \le k \qquad [s-1](E, x \mapsto l_1) \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_2 \Downarrow_u l_2; [t_2]H_2\end{array}}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{fit}\ e_1\ \texttt{to}\ x:\rho^k \to e_2 \mid \texttt{fail} \to e_3 \Downarrow_u l_2; [t_2]H_2} \text{(E-FIT-SUCCESS)}$$

$$\frac{\begin{array}{c}[s]E \mid [s]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1 \\ H_1(l_1) = \chi[k'](\overrightarrow{l'}) \qquad k' > k \qquad [s]E \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_3 \Downarrow_u l_3; [t_3]H_3\end{array}}{[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} \texttt{fit}\ e_1\ \texttt{to}\ x:\rho^k \to e_2 \mid \texttt{fail} \to e_3 \Downarrow_u l_3; [t_3]H_3} \text{(E-FIT-FAIL)}$$

Figure 4: Operational semantics for expressions

sizes, the inference rules are similar to those in first-order functional languages.

Constraints constructed in the type judgment gives the conditions for size parameters, e.g., the sizes of then-clause and else-clause in if-expression are the same. Also, termination of recursive calls is guaranteed by decreasing measures of functions. The constraints include the condition that the measure of f decreases from the current context in each recursive call. The constraints are in the category of Presburger arithmetic, and the solver judges their validity.

The type checking for the module is defined as the type checking for each element in the module (Fig. 6). Hereafter, we assume that these three rules defined in Fig. 6 are satisfied and the constraints constructed by the rules are valid.

4.4 Memory Size Estimation

Type checking guarantees the data type of the Emfrp$^{\mathrm{BCT}}$, its size, and the termination of recursive functions. But there is another important element in the operational semantics, the amounts of usage of resources in the evaluation. Our aim is to prove that the evaluation of the expression is completed in a finite number of steps if it passes the type checking and resources are properly prepared. Prior to this proof, we define here the estimation algorithm as the function $\mathcal{M}$ to calculate the amount of concrete resources (Fig. 7). The $\sqcup$ used here is the operator that returns the maximum value.

The estimate function starts from a node update to estimate how many resources are necessary for calculating its result. In the process, the size parameters, which are declared in function arguments and branches of case expressions, are instantiated as constants. The branch for a case expression may distribute the size among multiple parameters, so we try all of the possible patterns and find the maximum value in $\overline{\mathcal{M}}$.

The return value of the estimation function is a quadruple, each representing the size of the evaluation result (on the type system), the number of local variables, the amount of heap requirements, and the amount of call stack requirements, respectively. These parameters are not exact values and return more than what would be required for any branch to be passed through. The next section will show that this will provide sufficient resources.

4.5 Soundness

Here, we show the termination of the estimation function $\mathcal{M}$ and the soundness of the type system. Due to the lack of space, we describe proofs for these theorems in Appendix B and Appendix C.

Definition 4.1. When Γ and Δ satisfy the condition $\bigcup\{\mathrm{FV}(\Gamma(x)\downarrow) \mid x \in \mathrm{dom}(\Gamma)\} \subseteq \mathrm{dom}(\Delta)$, we say (Γ, Δ) consistent.

For heap H and location l, when $H(l) = c$, we say that l denotes c on H. Also, when $H(l) = \chi[k](l_1, \ldots, l_n)$ such that l_i denotes e_i on H and $k = \Sigma_{i \in I(\chi)} k_i$ where $H(l_i) = \chi_i[k_i](\ldots)$ for $i \in I(\chi)$, we say that l denotes $\chi(e_1, \ldots, e_n)$ on H.

When E, H, Γ and Δ satisfy the condition, (Γ, Δ) consistent, for each variable $x \in \mathrm{dom}(E)$, $E(x)$ denotes ce_x on H, ce_x is typed as τ^C, and $\tau^C \downarrow \leq ev_\Delta[[\Gamma(x)\downarrow]]$, we say (E, H, Γ, Δ) consistent.

Theorem 4.2 (termination of the estimate function). *Assume that $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$ is satisfied, any function g occurring in e satisfies $f \geq_{\mathcal{F}}^* g$, Δ is a model of C, and (Γ, Δ) consistent. Then $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[[e]]$ terminates in finite steps, and when $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[[e]] = (k, s, t, u)$, $k = ev_\Delta[[\tau\downarrow]]$.*

Corollary 4.3. *For any node N, $\mathcal{M}_{\epsilon;\Gamma_\mathcal{N}}^{\mathcal{T};\mathcal{F}}[[\mathcal{N}(N)]]$ is terminated in finite steps where $\Gamma_\mathcal{N} = \mathcal{I}, N_1 : \tau_1^C, \ldots, N_n : \tau_n^C$.*

Theorem 4.4 (soundness). *Assume that $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$, (E, H, Γ, Δ) consistent, any function g occurring in e satisfies $f \geq_{\mathcal{F}}^* g$, Δ is a model of C, and $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[[e]] = (p_\mathcal{M}, s_\mathcal{M}, t_\mathcal{M}, u_\mathcal{M})$. For any $s \geq s_\mathcal{M}$, $t \geq t_\mathcal{M}$ and $u \geq u_\mathcal{M}$, there exist t', l, H', ce and τ^C such that $[s]E \mid [t]H \vdash_\mathcal{L}^{\mathcal{T};\mathcal{F}} e \Downarrow_u l; [t']H'$, $t' \geq t - t_\mathcal{M}$, l denotes ce on H', ce is typed as τ^C and $ev_\Delta[[\tau^C\downarrow]] \leq ev_\Delta[[\tau\downarrow]]$.*

Corollary 4.5. *Assume that $\Gamma_\mathcal{N}(N) = \tau_N$, $(E, H, \Gamma_\mathcal{N}, \epsilon)$ consistent, and $\mathcal{M}_{\epsilon;\Gamma_\mathcal{N}}^{\mathcal{T};\mathcal{F}}[[\mathcal{N}_N]] = (p_\mathcal{M}, s_\mathcal{M}, t_\mathcal{M}, u_\mathcal{M})$. Then, there exist t', l, H', ce and τ^C such that $[s_\mathcal{M}]E \mid [t_\mathcal{M}]H \vdash_\mathcal{L}^{\mathcal{T};\mathcal{F}} \mathcal{N}(N) \Downarrow_{u_\mathcal{M}} l; [t']H'$, l denotes ce on H', ce is typed as τ^C and $ev_\Delta[[\tau^C\downarrow]] \leq ev_\Delta[[\tau_N\downarrow]]$.*

4.6 Example

4.6.1 `insert` *function of* `DupCheck` *module.* We illustrate the size constraints and resource requirements of `insert` function (Listing 3). The size constraint $A \equiv \top \to 2 \leq m+1 \wedge m+1 = m+1$ is from `N` branch. And the size constraint $B \equiv m = 1+n \wedge n > 0 \to n > 0 \wedge (n+1)+1 = m+1$ is from `C` branch. The $n > 0$ to the right of the arrow is a constraint for a recursive call. Since `l` of the `case` input is a variable reference, there are no additional size constraints. Thus we obtain constraint $A \wedge B$ for the `case` expression. As a result, we have $\forall m.m > 0 \to A \wedge B$ for `insert` function declaration.

When function call with $m = 1$, size 1 list, the result of the estimation algorithm $\mathcal{M}$ is $(2, 0, 2, 0)$ because `C` branch is not executed. The amount of resources required to call `insert` function when $m = n$ is calculated using the information at $m = n - 1$. That is, the result of $\mathcal{M}$ for $m = 2$ is $(3, 2, 3, 1)$;

4.6.2 `sumHeap` *function of* `Top10Sum` *module.* We also show an example of `sumHeap` function (Listing 4). As in the first example, we find the constraints of the `case` expressions. The size constraint on `N` branch is $A \equiv \top \to 0 = 0$. The size constraint on `T` branch is $B \equiv n = 1 + p + q \wedge p > 0 \wedge q > 0 \to p > 0 \wedge q > 0 \wedge 0 = 0$ because the decomposition produces two tree structures. Thus we obtain the size constraint $\forall n.n > 0 \to A \wedge B$ for `sumHeap` function declaration.

`sumHeap` is called at node `y` with $n = 21$. In this case, Algorithm $\mathcal{M}$ finds the amount of resources required by the `T` branch by doing an exhaustive search for size p and q. That is, let (p, q) be $(1, 19) \ldots (10, 10) \ldots (19, 1)$ and search recursively for the branch expression. The result of $\mathcal{M}$ is the parameter for the most resource-intensive case.

5 RELATED WORKS

The notion of Sized Types [14], types enhanced with size information, has a strong influence on Bounded-Construction Types. Some essential properties of embedded systems, such as deadlock freedom and termination, can be guaranteed statically using Sized Types. Like Sized Types, our method attaches size information to

$\boxed{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C}$

$$\frac{}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} c^{\mathcal{B}} : \mathcal{B} \mid \top}\ \text{(T-CONST)} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} x : \tau \mid \top}\ \text{(T-VAR)} \qquad \frac{\Gamma(N) = \tau}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} N : \tau \mid \top}\ \text{(T-NODE)} \qquad \frac{\Gamma(N) = \tau}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} N\texttt{@last} : \tau \mid \top}\ \text{(T-ATLAST)}$$

$$\frac{\chi(T_1^\rho, \ldots, T_n^\rho) \in \mathcal{T}(\rho) \qquad \left[\ \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_i : \tau_i \mid C_i\ \right]_{i \in 1\ldots n} \qquad \psi = 1 + \textstyle\sum_{i \in I(\chi)} \tau_i \downarrow}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \chi(e_1, \ldots, e_n) : \rho^\psi \mid \bigwedge_{i \in 1\ldots n} (C_i \wedge T_i^\rho \sim \tau_i)}\ \text{(T-CTOR)}$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_0 : \rho^\psi \mid C_0 \qquad \left[\ \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} branch_i :\sim \tau \mid C_i\ \right]_{i \in 1\ldots n}}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{case}\ e_0\ \texttt{return}\ \tau\ \texttt{of}\ \{branch_i\}_{i \in 1\ldots n} : \tau \mid \bigwedge_{i \in 0\ldots n} C_i}\ \text{(T-CASE)}$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1 : \tau_1 \mid C \qquad \Gamma, (x : \tau_1) \vdash_f^{\mathcal{T};\mathcal{F}} e_2 : \tau_2 \mid C'}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \mid C \wedge C'}\ \text{(T-LET)}$$

$$\frac{\begin{array}{c}\mathcal{F}(g) = (x_1 : \tau_1^\sharp, \ldots, x_n : \tau_n^\sharp) : \tau\ \texttt{where}\ \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}[\ \overrightarrow{\delta}\ ] = e \\ \left[\ \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_i : \tau_i \mid C_i\ \right]_{i \in 1\ldots n} \qquad \mathcal{R} = \begin{cases} \sum_{\delta \in \overrightarrow{\delta}} (\theta\delta) < \sum_{\delta \in \overrightarrow{\delta}} \delta & (f = g) \\ \top & (\text{otherwise}) \end{cases} \\ \theta = [\tau_1^\sharp, \ldots, \tau_n^\sharp \mapsto \tau_1, \ldots, \tau_n] \end{array}}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} g(e_1, \ldots, e_n) : \theta\tau \mid \bigwedge_{i \in 1\ldots n} C_i \wedge \bigwedge_{i \in 1\ldots m} (\theta\mathcal{A}_i) \wedge \mathcal{R}}\ \text{(T-CALL)}$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1 : \texttt{Bool} \mid C_1 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_2 : \tau \mid C_2 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_3 : \tau' \mid C_3}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \mid \tau \sim \tau' \wedge C_1 \wedge C_2 \wedge C_3}\ \text{(T-IF)}$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1 : \mathcal{B}_1 \mid C_1 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_2 : \mathcal{B}_2 \mid C_2}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1\ op^{(\mathcal{B}_1, \mathcal{B}_2) \to \mathcal{B}}\ e_2 : \mathcal{B} \mid C_1 \wedge C_2}\ \text{(T-OP)} \qquad \frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \rho^{\psi'} \mid C}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e\ \texttt{adj}\ [\psi] : \rho^{\psi} \mid C \wedge \psi' \leq \psi}\ \text{(T-ADJ)}$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1 : \rho^\psi \mid C_1 \qquad \Gamma, (x : \rho^k) \vdash_f^{\mathcal{T};\mathcal{F}} e_2 : \tau \mid C_2 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_3 : \tau' \mid C_3}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{fit}\ e_1\ \texttt{to}\ x : \rho^k \to e_2\ \texttt{|}\ \texttt{fail} \to e_3 : \tau \mid C_1 \wedge C_2 \wedge C_3 \wedge \tau \sim \tau'}\ \text{(T-FIT)}$$

$\boxed{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} branch :\sim \tau \mid C}$

$$\frac{\begin{array}{c}\chi(T_1^\rho, \ldots, T_n^\rho) \in \mathcal{T}(\rho) \qquad \mathcal{R} = \begin{cases} \tau \downarrow = 1 + \sum_{i \in I(\chi)} \tau_i^\sharp \downarrow & (I(\chi) \neq \emptyset) \\ \top & (\text{otherwise}) \end{cases} \\ \overrightarrow{\delta} = \{\tau_i^\sharp \downarrow \mid i \in 1 \ldots n, \tau_i^\sharp \downarrow \in SizeVar\} \qquad C = \mathcal{R} \wedge \bigwedge_{i \in 1\ldots n} T_i^\rho \sim \tau_i^\sharp \\ \Gamma, x_1 : \tau_1^\sharp, \ldots, x_n : \tau_n^\sharp \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau' \mid C' \end{array}}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \chi(x_1 : \tau_1^\sharp, \ldots, x_n : \tau_n^\sharp) \to e :\sim \tau \mid \forall \overrightarrow{\delta}.\, C \to (C' \wedge \tau' \sim \tau)}\ \text{(T-BRANCH)}$$

Figure 5: Typing rules for expressions

types and guarantees that a program will continue to run with bounded memory space. The main difference between Bounded-Construction Types (BCTs) and Sized Types is fit-expression, which accesses the size information from a program. For example, the type of filter function is `(Int → Bool) → List[m] → List[m]` (in both BCTs and Sized Types) because we cannot know whether some elements are filtered out from a given list. In Sized Types, we cannot add another element to the returned list without size growing, even if the list is smaller. In contrast, in BCTs, we can check the returned list smaller than expected size (10, for example) by fit-expression (`fit l to l':List [9]`), and add another element in a type-safe manner. This expression is useful in functional reactive programming because the size of the values in the previous state may be less than the sizes that the types say. In Sized Types, such situations are avoided by filling the dummy data instead of filtering out. This is not natural. Our method is straightforward from the standard manner.

Juniper [8] is an FRP language designed for Arduino, which is a microcomputer board using ATmega328. Juniper's template mechanism allows us to set a parameter called *capacity variable* [7] in addition to the usual type parameter in C++ as well. By using this parameter, it is possible to set the size of the array in the record. However, this parameter is translated to a C++ template. Thus, it does not guarantee the maximum size of the recursive data types.

$\boxed{\mathcal{F} \Longrightarrow C}$

$$\frac{\left[\begin{array}{c} \mathcal{F}(f) = (x_1 : \tau_1^\sharp, \ldots, x_n : \tau_n^\sharp) : \tau \text{ where } \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}[\overrightarrow{\delta}] = e \\ x_1 : \tau_1^\sharp, \ldots, x_n : \tau_n^\sharp \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C_f \end{array}\right]_{f \in \mathrm{dom}(\mathcal{F})}}{\mathcal{F} \Longrightarrow \bigwedge_{f \in \mathrm{dom}(\mathcal{F})} \forall \overrightarrow{\tau_i^\sharp} \downarrow . (\bigwedge_{i \in 1 \ldots m} \mathcal{A}_i) \to C_f} \quad \text{(C-FUNC)}$$

$\boxed{\mathcal{N} \Longrightarrow C}$

$$\frac{\mathrm{dom}(\mathcal{N}) = \{N_1, \ldots, N_n\} \qquad \left[\begin{array}{c} \mathcal{N}(N_i) = (\tau_i^C, e_i) \\ \mathcal{I}, N_1 : \tau_1^C, \ldots, N_n : \tau_n^C \vdash_-^{\mathcal{T};\mathcal{F}} e : \tau_i^C \mid C_i \end{array}\right]_{i \in 1 \ldots n}}{\mathcal{N} \Longrightarrow \bigwedge_{i \in 1 \ldots n} C_i} \quad \text{(C-NODE)}$$

$\boxed{Init \Longrightarrow}$

$$\frac{\left[\begin{array}{c} \mathcal{N}(N) = (\tau^C, e) \\ \cdot \vdash_-^{\mathcal{T};\mathcal{F}} Init(N) : \tau^C \mid C_N \end{array}\right]_{N \in \mathrm{dom}(\mathcal{N})} \qquad \left[\ \cdot \vdash_-^{\mathcal{T};\mathcal{F}} Init(N) : \tau^C \mid C_N\ \right]_{(N,\tau^C) \in \mathcal{I}}}{Init \Longrightarrow \bigwedge_{N \in \mathrm{dom}(\mathcal{N})} C_N \wedge \bigwedge_{(N,\tau^C) \in \mathcal{I}} C_N} \quad \text{(C-INIT)}$$

Figure 6: Typing Constraints on Modules

RT-FRP [19] and E-FRP [20] by Wan et al. are FRP languages for real-time systems, which can statically check the upper bound of the resources (term size) used by a program. These languages allow static estimates of the total amount of resources required by the program, but do not allow the size of the data structure to be set in advance. In addition, these languages do not have any language features that correspond to the recursive data types. On the other hand, the proposed method can control the amount of resources in each data structure by using size parameters.

Futhark [10] is a purely functional language for GPUs. It can treat arrays as primitive data structures and is statically checked for size consistency by a type called Size Types [9]. It behave as a kind of dependent type, which can contain values passed as arguments to a function. On the contrary, all of our size parameters are processed at the type level. That is, the value of the size parameter cannot propagate to the value level or vice versa. This is a design choice to statically analyze the upper bound of memory usage at runtime.

Lucid Synchrone [2] and its advances, Lustre [6, 16] and Zélus [1], are synchronous data flow languages for real-time reactive systems. This language has a lot in common with Emfrp, such as combining time-varying values to describe a program, and being able to refer to values one time step earlier (`@last` in Emfrp, `pre` in Lustre). These languages, as well as Emfrp, prohibit loops (recursive calls) and recursive data structures such as lists in order to estimate the (worst) execution time and memory usage. Instead, they provide arrays and primitive operations (e.g., `map`, `reduce`, etc.) to manipulate them. The examples using the lists (Fig. 3) can also be implemented using arrays. However, if we want to use data structures such as trees, we represent it more naturally by using BCTs.

Krishnaswami et al. [15] controlled the memory allocation in FRP programs by using some kind of linear type. In their proposed language, we can control the special values representing memory resources by linear types to handle the list structure in a bounded memory space. While linear types can control the passing of memory resources, resource objects must be managed by programmers. In our method, we do not track memory resources, but determine the upper bounds of memory by statically evaluating the behavior of the program on size parameters.

Our memory estimation algorithm measures the worst-case memory consumption by scanning the program based on the size information. Automatic Amortized Resource Analysis (AARA), introduced by Hofmann *et al* [12], is a different method of this issue. Hoffman et al. [11] proposed an AARA method for OCaml and derived a multivariable polynomial bounds for some examples.

6 CONCLUSION

In this paper, we introduce a size-bounded recursive data type called Bounded-Construction-Types (BCTs) for Emfrp, an FRP language for resource-bounded embedded systems. After showing the effectiveness of BCTs with examples, we give a formal definition of Emfrp$^{\mathrm{BCT}}$, an extension of Emfrp with BCTs as well as the proof of the soundness of the language.

Several tasks remain for future work. One is to expand BCTs to have polymorphism with type and size. Another is to implement the extended language to show the benefit of BCTs through actual applications. We also plan to extend the memory estimation algorithm using different approaches (e.g., AARA).

REFERENCES

[1] Timothy Bourke and Marc Pouzet. 2013. Zélus: A Synchronous Language with ODEs. In *16th International Conference on Hybrid Systems: Computation and Control (HSCC'13)*. Philadelphia, USA, 113–118. http://www.di.ens.fr/~pouzet/bib/hscc13.pdf

[2] Paul Caspi, Grégoire Hamon, and Marc Pouzet. 2010. *Synchronous Functional Programming with Lucid Synchrone*. 207 – 247. https://doi.org/10.1002/9780470611012.ch7

[3] Evan Czaplicki. 2016. A Farewell to FRP: Making signals unnecessary with The Elm Architecture. http://elm-lang.org/blog/farewell-to-frp. http://elm-lang.org/blog/farewell-to-frp

[4] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. ACM, 411–422. https://doi.org/10.1145/2499370.2462161

[5] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*. ACM, 263–273. https://doi.org/10.1145/258949.258973

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[7] Caleb Helbling. 2016. Juniper Language Documentation (ver. 2.2.0). http://www.juniper-lang.org/language_docs.html. http://www.juniper-lang.org/index.html

[8] Caleb Helbling and Samuel Z Guyer. 2016. Juniper: A Functional Reactive Programming Language for the Arduino. In *4th International Workshop on Functional Art, Music, Modelling, and Design (FARM 2016)*. ACM, 8–16. https://doi.org/10.1145/2975980.2975982

[9] Troels Henriksen. 2019. Towards Size Types in Futhark. https://futhark-lang.org/blog/2019-08-03-towards-size-types.html. https://futhark-lang.org/blog/2019-08-03-towards-size-types.html

$$\boxed{\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e]\!] = (k, s, t, u)}$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![c^{\mathcal{B}}]\!] = (0,0,1,0) \qquad \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![x]\!] = (ev_{\Delta}[\![\Gamma(x)\downarrow]\!], 0, 0, 0)$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![N]\!] = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![N\texttt{@last}]\!] = (ev_{\Delta}[\![\Gamma(N)\downarrow]\!], 0, 0, 0)$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!] = (k_2, s_1 \sqcup (1 + s_2), t_1 + t_2, u_1 \sqcup u_2)$$
$$\text{where } (k_1, s_1, t_1, u_1) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_1]\!],\ (k_2, s_2, t_2, u_2) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{(\Delta,\tau^\sharp\downarrow\mapsto k_1);(\Gamma,x:\tau^\sharp)}[\![e_2]\!]$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!] = (k_2, s_1 \sqcup s_2 \sqcup s_3, t_1 + (t_2 \sqcup t_3), u_1 \sqcup u_2 \sqcup u_3)$$
$$\text{where } (k_i, s_i, t_i, u_i) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!]\ (1 \le i \le 3)$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_1\ op^{(\mathcal{B}_1,\mathcal{B}_2)\to\mathcal{B}}\ e_2]\!] = (0, s_1 \sqcup s_2, 1 + t_1 + t_2, u_1 \sqcup u_2)$$
$$\text{where } (k_i, s_i, t_i, u_i) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!]\ (1 \le i \le 2)$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![f(e_1, \ldots, e_n)]\!] = (k',\ n + (s' \sqcup \bigsqcup_{i\in 1\ldots n} s_i),\ t' + \textstyle\sum_{i\in 1\ldots n} t_i,\ (1 + u') \sqcup \bigsqcup_{i\in 1\ldots n} u_i)$$
$$\text{where } (k_i, s_i, t_i, u_i) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!]\ (1 \le i \le n),$$
$$\mathcal{F}(f) = (x_1 : \tau_1^\sharp, \ldots, x_n : \tau_n^\sharp) : \tau \texttt{ where } \{\overrightarrow{\mathcal{A}}\}[\ \overrightarrow{\delta}\ ] = e,$$
$$(k', s', t', u') = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\{\tau_i^\sharp\downarrow\mapsto k_i\}_{i\in 1\ldots n};\{x_i:\tau_i^\sharp\}_{i\in 1\ldots n}}[\![e]\!]$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\chi(e_1, \ldots, e_n)]\!] = (1 + \textstyle\sum_{i\in I(\chi)} k_i,\ \bigsqcup_{i\in 1\ldots n} s_n,\ 1 + \sum_{i\in 1\ldots n} t_i,\ \bigsqcup_{i\in 1\ldots n} u_i)$$
$$\text{where } (k_i, s_i, t_i, u_i) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!]\ (1 \le i \le n),\ \chi(\overrightarrow{T^\rho}) \in \mathcal{T}(\rho)$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{case } e_0 \texttt{ return } \tau \texttt{ of } \{branch_i\}_{i\in 1\ldots n}]\!] = (ev_{\Delta}[\![\tau\downarrow]\!],\ \bigsqcup_{i\in 0\ldots n} s_i,\ t_0 + \bigsqcup_{i\in 1\ldots n} t_i,\ \bigsqcup_{i\in 0\ldots n} u_i)$$
$$\text{where } (k_0, s_0, t_0, u_0) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_0]\!],\ (k_i, s_i, t_i, u_i) = \overline{\mathcal{M}}^{\mathcal{T};\mathcal{F}}_{k_0;\Delta;\Gamma}(branch_i)\ (1 \le i \le n)$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e\ \texttt{adj}\ [\psi]]\!] = (ev_{\Delta}[\![\psi]\!], s, t, u) \quad \text{where } (k, s, t, u) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e]\!]$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{fit } e_1 \texttt{ to } x : \rho^k \to e_2 \mid \texttt{fail} \to e_3]\!] = (k_2, s_1 \sqcup (1 + s_2) \sqcup s_3, t_1 + (t_2 \sqcup t_3), u_1 \sqcup u_2 \sqcup u_3)$$
$$\text{where } (k_1, s_1, t_1, u_1) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_1]\!],\ (k_2, s_2, t_2, u_2) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma,x:\rho^k}[\![e_2]\!],\ (k_3, s_3, t_3, u_3) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_3]\!]$$

$$\overline{\mathcal{M}}^{\mathcal{T};\mathcal{F}}_{m;\Delta;\Gamma}(\chi(x_1 : \tau_1^\sharp, \ldots, x_n : \tau_n^\sharp) \to e) =$$
$$\text{if } m \le |I(\chi)| \text{ then } (0,0,0,0) \text{ else } (\textstyle\bigsqcup_{a\in\mathbf{A}} k_a, n + \bigsqcup_{a\in\mathbf{A}} s_a, \bigsqcup_{a\in\mathbf{A}} t_a, \bigsqcup_{a\in\mathbf{A}} u_a)$$
$$\text{where } \chi(T_1^\rho, \ldots, T_n^\rho) \in \mathcal{T}(\rho),$$
$$\mathbf{A} = \{\ \{\tau_i^\sharp\downarrow\mapsto p_i\}_{i\in I(\chi)}\ \mid (\forall i \in 1\ldots n, p_i > 0) \wedge m = 1 + \textstyle\sum_{i\in I(\chi)} p_i \vee I(\chi) = \emptyset\},$$
$$b = \{\tau_j^\sharp\downarrow\mapsto k_j \mid j \in 1\ldots n, T_j^\rho = \rho'^{k_j}\},$$
$$(k_a, s_a, t_a, u_a) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{(\Delta,a,b);(\Gamma,x_1:\tau_1^\sharp,\ldots,x_n:\tau_n^\sharp)}[\![e]\!]\ (a \in \mathbf{A})$$

Figure 7: Estimation Algorithm

[10] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[11] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 359–373. https://doi.org/10.1145/3009837.3009842

[12] Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 185–197. https://doi.org/10.1145/604131.604148

[13] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*. Lecture Notes in Computer Science, Vol. 2638. Springer-Verlag, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[14] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, 410–423. https://doi.org/10.1145/237721.240882

[15] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. 2012. Higher-Order Functional Reactive Programming in Bounded Space. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 45–58. https://doi.org/10.1145/2103656.2103665

[16] LustreV6 2020. *Verimag Lustre V6*. Retrieved May 23, 2020 from http://www-verimag.imag.fr/Lustre-V6.html

[17] Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press.

[18] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems. In *MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity*. ACM, 36–44. https://doi.org/10.1145/2892664.2892670

[19] Zhanyong Wan, Walid Taha, and Paul Hudak. 2001. Real-Time FRP. In *International Conference on Functional programming (ICFP 2001)*. ACM SIGPLAN, 146–156. https://doi.org/10.1145/507635.507654

[20] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-Driven FRP. In *Practical Aspects of Declarative Langauges (Lecture Notes in Computer Science, Vol. 2257)*. Springer-Verlag, 155—172. https://doi.org/10.1007/3-540-45587-6_11

A SOURCE CODE OF TOP10SUM MODULE

```
# The sum of the top 10 of the input data
module Top10Sum
in  x: Int    # input value
out y: Int    # sum

# Leftist Heap
type Heap = E                # terminal
     | T(Int,Int,Heap,Heap) # (rank,v,left,right)

# get tree rank
func rank(e: Heap[n]) : Int where {n > 0} =
  case e return Int of E -> 0 | T(r, x, a, b) -> r

# make a heap tree
func make(x:Int,a:Heap[n],b:Heap[m]):Heap[1+n+m]
  where {n > 0, m > 0} =
  if ra > rb then T(rank(b)+1, x, a, b)
    else T(rank(a)+1, x, b, a)

# merge heap trees
func merge(h1:Heap[n],h2: Heap[m]):Heap[n+m-1]
  where {n > 0, m > 0} [n,m] =
  case h1 return Heap[n+m-1] of E -> h2 adj[n+m-1]
  | T(r1, x1, a1, b1) ->
      case h2 return Heap[n+m-1] of E -> h1 adj[n+m-1]
      | T(r2, x2, a2, b2) ->
          if x1 <= x2 then
            make(x1, a1, merge(b1, h2))
          else
            make(x2, a2, merge(h1, b2))

# insert value
func insert(x: Int, h: Heap[n]): Heap[n+2]
  where {n > 0} = merge(T(1, x, E, E), h)

# get min value
func findMin(h: Heap[n]): Int where {n>0} =
  case h return Int of E -> 0 | T(r, x, a, b) -> x

# delete min value
func delMin(h:Heap[n]):Heap[n-2] where {n>2} =
  case h return Heap[n-2] of E -> E adj[n-2]
  | T(r, x, a, b) -> merge(a, b)

# sum of values in heap tree
func sumHeap(h: Heap[n]): Int where {n>0} =
  case h return Int of E -> 0
  | T(r, x, a: Heap[p], b: Heap[q]) -> x+sumHeap(a)+
     sumHeap(b)

# heap tree node
node h: Heap[21] init (E adj[21]) =
  fit h@last to hl: Heap[19] -> insert(x, hl)
  | fail -> if x <= findMin(h@last) then h@last
              else insert(x, delMin(h@last))

# sum of heap node
node y: Int init 0 = sumHeap(h)
```

Listing 5: Duplication Checking Module using List

B PROOF OF TERMINATION OF THE ESTIMATE FUNCTION

Theorem B.1. *Assume that* $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$ *is satisfied, any function* g *occuring in* e *satisfies* $f \geq_{\mathcal{F}}^* g$, Δ *is a model of* C, *and* (Γ, Δ) *consistent. Then* $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!]$ *terminates in finite steps, and when* $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!] = (k, s, t, u)$, $k = ev_\Delta[\![\tau \downarrow]\!]$.

Proof. We prove this by induction on the lexicographic order of triple $(f, ev_\Delta[\![\overrightarrow{\delta}]\!], e)$ where $\overrightarrow{\delta}$ is the measure of f. The ordering of functions is defined as $\geq_{\mathcal{F}}^* \cup \{(-, f) | f \in \mathrm{dom}(\mathcal{F})\}$ ($-$ is a special symbol used for node typing rules). Also, for the term e, the subterms of e are considered to be smaller than e.

We divide the cases for the last rules used for deriving $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$. The cases of T-CONST, T-VAR, T-NODE and T-ATLAST are clear. The cases of T-CTOR, T-LET, T-IF, T-OP, T-ADJ and T-FIT are directly proven with induction hypothesis (decreasing the third of the triple).

The case of T-CASE is also proven with induction hypothesis, but we should check the conditions in the assertion of this theorem. The estimation function applied to a case-expression invokes itself with the body of each branch and size parameters of the constructor. The number of branches and the patterns of size parameters ($\mathbf{A}$ in $\overline{\mathcal{M}}$) are finite, the number of these recursive invocations of the estimate function are also finite. Each invocation terminates in finite steps (by induction hypothesis), and therefore the case of T-CASE also terminates. The assertion about k is straightforward.

The case of T-CALL is proven with induction hypothesis under three patterns of decrease of the triple. Let e be $g(e_1, \ldots, e_n)$. By induction hypothesis (decreasing the third of the triple), the estimation of each argument terminates and for i-th argument, $k_i = ev_\Delta[\![\tau_i \downarrow]\!]$. When $g = f$, because Δ is a model of $\mathcal{R}$ in T-CALL, $ev_\Delta[\![\Sigma_{\delta \in \overrightarrow{\delta}}(\theta\delta)]\!] < ev_\Delta[\![\overrightarrow{\delta}]\!]$. From the definition of θ and $\overrightarrow{\delta}$, $\theta\delta$ is equivalent to $\tau_i \downarrow$ for some i, i.e., $ev_\Delta[\![\theta\delta]\!] = k_i$. In the estimation of the function body of g, $ev_{\{\tau_i^\sharp \downarrow \mapsto k_i\}_{i \in 1\ldots n}}[\![\overrightarrow{\delta}]\!] = ev_{\{\tau_i^\sharp \downarrow \mapsto k_i\}_{i \in 1\ldots n}}[\![\Sigma_{\tau_i^\sharp \downarrow \in \overrightarrow{\delta}} \tau_i \downarrow]\!] = ev_{\{\tau_i^\sharp \downarrow \mapsto k_i\}_{i \in 1\ldots n}}[\![\Sigma_{\tau_i^\sharp \downarrow \in \overrightarrow{\delta}} k_i]\!] = ev_\Delta[\![\Sigma_{\delta \in \overrightarrow{\delta}}(\theta\delta)]\!]$. This means that, in the estimation of the function body of g, the second of the triple decreases. When $g \neq f$, from the assumption of occurrence of functions, g is less than f with respect to the function ordering. In both cases the triple decreases in the estimation of the function body, and type judgement of the function body is guaranteed by the typing conditions for $\mathcal{F}$. Therefore, from the induction hypothesis, the estimation terminates. □

C PROOF OF SOUNDNESS

Theorem C.1. *Assume that* $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$, (E, H, Γ, Δ) *consistent, any function* g *occurring in* e *satisfies* $f \geq_{\mathcal{F}}^* g$, Δ *is a model of* C, *and* $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!] = (p_\mathcal{M}, s_\mathcal{M}, t_\mathcal{M}, u_\mathcal{M})$. *For any* $s \geq s_\mathcal{M}$, $t \geq t_\mathcal{M}$ *and* $u \geq u_\mathcal{M}$, *there exist* t', l, H', ce *and* τ^C *such that* $[s]E \mid [t]H \vdash_\mathcal{L}^{\mathcal{T};\mathcal{F}} e \Downarrow_u l; [t']H'$, $t' \geq t - t_\mathcal{M}$, l *denotes* ce *on* H', ce *is typed as* τ^C *and* $ev_\Delta[\![\tau^C \downarrow]\!] \leq ev_\Delta[\![\tau \downarrow]\!]$.

Proof. We use the same lexicographic order of the triple $(f, ev_\Delta[\![\overrightarrow{\delta}]\!], e)$ as that in the proof of termination of the estimation function. We also use the same devision of the cases, i.e., the last rules used for deriving $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$. Since the assumptions of this theorem include those of the termination, it is the same way to show how the triple decreases.

The cases of T-CONST, T-VAR, T-NODE and T-ATLAST are clear ($H' = H, l \mapsto c$ and $t' = t - 1$ in T-CONST, $H' = H$ and $t' = t$ in the other cases).

The case of T-CTOR is proven as follows: Let $H = H_0$, $t = t_0$, $e = \chi(e_1, \ldots, e_n)$, and $\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!] = (p_{\mathcal{M}i}, s_{\mathcal{M}i}, t_{\mathcal{M}i}, u_{\mathcal{M}i})$ for $1 \leq i \leq n$. From the definition of $\mathcal{M}$, for any $1 \leq i \leq n$, $s \geq s_{\mathcal{M}} \geq s_{\mathcal{M}i}$ and $u \geq u_{\mathcal{M}} \geq u_{\mathcal{M}i}$. Since $t_0 \geq t_{\mathcal{M}} = 1 + \Sigma_{i \in 1 \ldots n} t_{\mathcal{M}i} \geq t_{\mathcal{M}1}$, there exist t_1, l_1, H_1, ce_1 and τ_1^C such that $[s]E \mid [t_0]H_0 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1, t_1 \geq t_0 - t_{\mathcal{M}1} \geq 1 + \Sigma_{i \in 2 \ldots n} t_{\mathcal{M}i}, l_1$ denotes ce_1 on H_1, ce is typed as τ_1^C and $ev_\Delta[\![\tau_1^C \downarrow]\!] \leq ev_\Delta[\![\tau_1 \downarrow]\!]$. Repeating the same process, we obtain $t_n \geq t_0 - \Sigma_{i \in 1 \ldots n} t_{\mathcal{M}i} \geq 1$ and H_n, and applying E-CTOR, we obtain $t' = t_n - 1 \geq t_0 - (1 + \Sigma_{i \in 1 \ldots n} t_{\mathcal{M}i}) = t - t_{\mathcal{M}}$, $H' = H_n, l \mapsto \chi[k](l_1, \ldots, l_n)$, $ce = \chi(ce_1, \ldots, ce_n)$.

The cases of T-IF, T-OP and T-ADJ are similar way as above. The cases of T-LET and T-FIT (more specifically, E-FIT-SUCCESS) are almost similar, and in addition, consistency about newly bound variables (x in the both cases) are derived from induction hypotheses for e_1. The case of T-CALL is similar for arguments to T-CTOR, and consistency about arguments is similar for T-LET.

In the case of T-CASE, induction hypothesis for the scrutinee e_0 says that there exist t_0', l_0, H_0', ce_0 and τ_0^C such that $[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_0 \Downarrow_u l; [t_0']H_0', t_0' \geq t - t_{\mathcal{M}0}, l$ denotes ce_0 on H', ce_0 is typed as τ_0^C and $ev_\Delta[\![\tau_0^C \downarrow]\!] \leq ev_\Delta[\![\tau_0 \downarrow]\!]$. Let $H_0'(l) = \chi[k](l_1, \ldots, l_n)$. Since ce_0 (denoted by l) is typed as τ_0^C, so the subterm $ce_1, \ldots, ce_n$ (which are denoted by $l_1, \ldots, l_n$, respectively) are also typed. From this fact, we can show the consistency of the environments at the body of branch for χ. The rest of the proof can be done in the same way with induction hypothesis. □