Electronic Preprint for Journal of Information Processing Vol.29

Regular Paper

A Functional Reactive Programming Language for Small-Scale Embedded Systems with Recursive Data Types

Akihiko Yokoyama[1,a)] Sosuke Moriguchi[1,b)] Takuo Watanabe[1,c)]

Received: February 16, 2021, Accepted: May 27, 2021

Abstract: We introduce a new type system to Emfrp, a functional reactive programming (FRP) language designed for resource-constrained embedded systems. Functional reactive programming is a programming paradigm that allows concise descriptions of reactive systems such as GUIs by combining time-varying values that express values changing over time. Emfrp is a domain-specific language based on FRP, designed and developed for small-scale embedded systems. Because the language can statically determine the amount of runtime memory and guarantee the termination of reactive actions, a program written in Emfrp can safely continue reactive behaviors in resource-constrained environments. To ensure these properties, Emfrp disallows the use of recursive data types and functions. However, such restrictions often impose unnatural representations of data structures like lists or trees. The declarative characteristic of FRP and these restrictions impel us to write poorly maintainable redundant codes or deter us from writing certain types of programs. In this paper, we propose Emfrp$^{\mathrm{BCT}}$, an extended Emfrp with size-annotated recursive data types, to overcome this problem. The proposed system is more expressive than Emfrp, yet, it retains the aforementioned static properties. After explaining that through examples, we describe the features of Emfrp$^{\mathrm{BCT}}$, formalize the language, present an algorithm for statically computing the runtime memory bounds, and prove its soundness. Moreover, we implemented a compiler from Emfrp$^{\mathrm{BCT}}$ to C, measured the translation time, and evaluated runtime overhead.

Keywords: functional reactive programming, embedded systems, type systems

1. Introduction

A reactive system responds to external inputs by continuously producing outputs while changing its internal state. Graphical user interfaces (GUIs) and programs that control home appliances and robots are familiar examples of reactive systems. A program for a reactive system is often written using polling and/or interrupts (callbacks) to describe the responses to asynchronous inputs. However, such programs tend to be complex [2].

Functional reactive programming (*FRP*) [6] is a programming paradigm that simplifies the description of reactive systems by declaratively combining objects called *time-varying values*. A time-varying value is an abstraction of a value that changes continuously over time. By describing the data flow from input to output using time-varying values, programmers can concisely describe a process that continues responding to the input.

Since its invention [6], FRP has mainly evolved as a series of domain-specific languages (DSLs) or libraries for the functional programming language Haskell. For this reason, FRP libraries (e.g., Refs. [1], [19], [28]) and language processors (e.g., Ref. [4]) developed thus far often require a certain amount of computational resources for their execution. The FRP language Emfrp [27] is targeted at small-scale embedded environments such as microcontrollers, where CPU performance and memory size are limited. Although FRP is useful in small-scale embedded environments, we should be aware of runtime errors due to resource exhaustion. Emfrp is designed to have a small memory footprint for compiled executables. The amount of memory used at the runtime is determined statically (see Section 3.1), and dynamic allocation of extra runtime memory is not required. Furthermore, Emfrp guarantees the termination of the update process of time-varying values (nodes); see Section 2.3. Thus, a program written with Emfrp can safely continue its reactive behavior without running out of memory resources, even in a resource-constrained environment. Several restrictions are imposed on the language to guarantee these properties, such as not treating time-varying values as first-class data, and disallowing higher-order functions as well as recursion in data types and functions. However, in particular, *disallowing recursion* in data types and functions makes it difficult to naturally express dynamic data structures such as lists and trees, which are also valuable for embedded systems. In addition, because FRP is based on declarative descriptions, introducing these restrictions may result in programs that are difficult to maintain or modify. However, the unrestricted use of recursion leads to programs that consume unlimited memory or those that do not terminate the updating process of time-varying values.

In this study, to relax the restriction on recursive definitions while maintaining the aforementioned properties of Emfrp, we propose Emfrp$^{\mathrm{BCT}}$, an extension of Emfrp with recursive data types that contain information about the maximum sizes of constructible structures. Furthermore, the language allows us to define restricted recursive functions (primitive recursive functions)

[1] Department of Computer Science, School of Computing, Tokyo Institute of Technology, Tokyo 152–8552, Japan
[a)] akihiko@psg.c.titech.ac.jp
[b)] chiguri@acm.org
[c)] takuo@acm.org

© 2021 Information Processing Society of Japan

whose arguments definitely decrease with recursive calls. The recursive data types and the associated language mechanisms introduced in Emfrp^{BCT} can be regarded as an adaptation of Sized Types [21] to the programming style in Emfrp, which actively utilizes the previous values of time-varying values. A detailed comparison is given in Section 7.1.

In Emfrp^{BCT}, the size of the data is checked statically as part of type checking, so programs using unlimited memory will not be compiled and executed. The termination of the update process of an arbitrary node is also guaranteed. Thus, a program written in Emfrp^{BCT} can safely continue its reactive behavior without running out of memory resources. Programs with recursive data, such as lists and heap trees, must be written in an unnatural form using tuples in Emfrp. In contrast, using Emfrp^{BCT}, these programs can be written in a concise and maintainable manner. We show this using examples in Section 4.

The main contribution of this research is *the design of an FRP language with recursive data types containing information about the amount of memory to be used, along with an algorithm for estimating the memory usage of a program written in the language*. The specific contributions are as follows.

- We formally define the syntax, operational semantics, and type system of Emfrp^{BCT} and present an algorithm for estimating the amount of memory required to evaluate expressions.
- We prove that for an arbitrary node, the amount of resources obtained by the algorithm is sufficient for updating if the node is well-typed and passes the size validation.
- We implement a compiler from Emfrp^{BCT} to C and measure the time required for type checking and estimating the amount of memory, as well as the time and space overhead of using recursive data types.

The structure of this paper is as follows. After briefly describing Emfrp in Section 2, we present the restrictions enforced by the language and the problems caused by these restrictions in Section 3, using some motivating examples. Section 4 introduces recursive data types with size information and provides an informal description of Emfrp^{BCT}, an extension of Emfrp with the types. In Section 5, we present the formalization of Emfrp^{BCT} by providing the operational semantics, type system, and an algorithm for estimating memory usage. We also demonstrate the soundness of the algorithm. Section 6 describes the implementation of Emfrp^{BCT} and examines the results of overhead measurements. In Section 7 we discuss related work and Section 8 concludes the paper with future directions.

2. Emfrp

This section provides an overview of Emfrp [27], an FRP language for small-scale embedded systems, followed by descriptions of its execution model and memory management.

2.1 Overview

Emfrp is a statically typed pure FRP language used to describe reactive programs running on small-scale embedded systems. Programs written in the language are intended to run on environments such as microcontrollers with low-power CPUs and a few kilobytes of memory (RAM or Flash) or bare-metal systems without an operating system (OS). Emfrp is designed to avoid FRP-specific problems called space leaks (an increase in memory consumption due to the retention of past values of time-varying values) and time leaks (an increase in computation time due to the growth of the history required to update time-varying values) by imposing several restrictions on the language. The restrictions are as follows. Among the past time-varying values, only the last value can be referenced. Time-varying values are not first-class values and are always referred to by names. Higher-order time-varying values are not allowed. Moreover, time-varying values cannot be referenced within a function.

An Emfrp program is compiled into a pair of C source files. One contains a single loop that implements continuous reactive processing, and the other contains function skeletons for input and output. By filling the latter with C code that implements input and output to and from the external environment, the user can complete an executable Emfrp module. The implementation of the input/output code generally depends on the runtime environment. As the C code generated by the Emfrp compiler is platform-independent, we can run Emfrp programs on various platforms by preparing the input/output code.

2.2 Example Program

Figure 1 shows an Emfrp program that calculates the position of a two-wheeled differential drive robot, which is an Emfrp version of an example in Ref. [19]. The inputs of the program are the velocities v_l and v_r of the left and right wheels, respectively, the angle θ between the robot direction and the x-axis, and the elapsed time t. The output x is the x-coordinate of the robot position. The following equation expresses the relationship between the inputs and output:

$$x(t) = \frac{1}{2}\int_0^t (v_l(u) + v_r(u))\cos(\theta(u))du \qquad (1)$$

An Emfrp program is written as a module. Line 1 in Fig. 1 specifies the module name. In Emfrp, time-varying values are called *nodes*. Nodes are categorized into *input nodes*, *output nodes*, and *intermediate nodes*. An input node takes values from the external environment, and an output node provides its values to the external environment.

Lines 2–5 in Fig. 1 are the declarations of the input nodes (`vl`, `vr`, `theta`, `t`), and line 6 is the declaration of the output node (`x`), which correspond to v_l, v_r, θ, t, and x in Equation (1), respectively. The external environment provides the values to the input nodes. In this example, we assume that the rotary encoders on the robot provide the values of `vl` and `vr` (the speeds of the wheels), the orientation sensor provides the value of `theta` (the angle from the x-axis), and the CPU calculates the value of `t` (elapsed time since system startup). The values of the intermediate and output nodes are defined using the keyword **`node`**. Lines 10 and 13 are definitions of the intermediate node `dt` and output node `x`, respectively. The definition of a node is expressed in the form **`node`** n = e or **`node init`**`[`c`]` n = e, where n is the name of the node, and e is an expression to update the value of the node, called an *update expression*. The optional **`init`**`[`c`]` specifies the initial value of

© 2021 Information Processing Society of Japan

```
 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 the x-axis (rad)
 5      t(0) : Int     # elapsed time (msec)
 6  out x : Float      # x-coordinate 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-coordinate of the robot (m)
13  node init[0.0] x =
14    x@last + (vr+vl) * cos(theta) * dt/2
```

Fig. 1 An Emfrp program for calculating the *x*-coordinate of a differential drive robot.

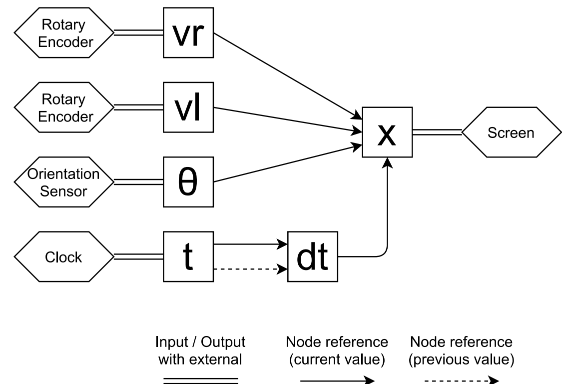


Fig. 2 Dependency graph of the `RobotPos` module.

the node, where c is a constant. A node name with `@last` is an expression that represents the *previous value* of the node. For example, `t - t@last` in line 10 provides the elapsed time since the last node update. In lines 13–14, the *x*-coordinate of the current position (derived from the integration over time) is calculated by adding its small changes to `x@last`. Nodes whose previous values are referenced in the program must have their initial values specified. In this example, the initial value of the input node `t` is specified as `0` in line 5, and the initial value of the output node `x` is specified as `0.0` in line 13.

2.3 Execution Model

If a node n' is used in the definition of a node n (i.e., n' occurs in the update expression of n without `@last`), n is said to depend on n'. Emfrp requires that the dependencies between nodes in a program exclude cycles. In other words, the graph constructed with the nodes as vertices and the dependencies of the nodes as directed edges must be a directed acyclic graph (DAG). This graph is called a dependency graph. Note that the occurrences of node names with `@last` are not considered as dependencies. **Figure 2** shows the dependency graph for Fig. 1. In this graph, the dependencies are indicated by solid arrows, and reference relations with `@last` are shown as dotted arrows.

The Emfrp compiler statically schedules the order of node updates by topologically sorting the dependency graph. For example, from Fig. 2, the sequence (`t`, `dt`, `vl`, `vr`, `theta`, `x`) can be obtained as the node update order. The runtime system of the language first receives input from the external environment, then performs node updates along with the obtained update order, outputs the values to the external environment, and finally performs memory management. This sequence of operations is called an *iteration*. The reactive behavior of an Emfrp program is realized by repeated iterations. The FRP execution model described previously, in which node updates are performed along the order obtained from the dependency graph, is called the *push-based* model [2]. This model may cause unnecessary node updates, but it can simplify the scheduling and runtime mechanisms. Therefore, it is used in Emfrp, which is designed for small-scale environments.

2.4 Memory Management

In the Emfrp runtime system, an array is statically allocated for each tuple or variant type in the program. Objects of the type are stored in the array (i.e., the array serves as a heap area for the type). Such a heap-allocated object is passed by the reference when a function is called. Primitive types such as `Int` are represented as their C counterparts, and no heap areas are required for data of these types.

To release objects allocated in heaps but no longer needed, a variant of mark-sweep garbage collection (GC) is performed for every iteration. Immediately after each node update, if the value of the updated node is an object in a heap area, it is marked to extend its lifetime. The necessity of a heap-allocated object is determined according to the lifetime set at the time of allocation. Objects that are temporarily used during node updates are not explicitly released. At the end of each iteration, the area in the heap that stores the previous values is released. Then, for each node, its current value is switched to the previous value for the next iteration.

The total amount of runtime memory required by an Emfrp program is the sum of the amount of memory used by the current and previous values of each node, the maximum amount of memory used during node updates, and the amount of memory used by the language runtime. Because these amounts of memory can be determined statically, the Emfrp compiler can fix the sizes of the heap area for objects and the stack area for local variables at compile time. The heap area need not be expanded at runtime, making it possible to execute a program with a fixed amount of statically allocated memory.

3. Motivation

Emfrp has several language restrictions to ensure safe program execution on embedded systems. This section describes the effect of these restrictions on the expressiveness of Emfrp and the problems they cause.

3.1 Restrictions in Emfrp

Programs written in Emfrp are statically guaranteed to continue reactive behavior safely in resource-constrained environments because of the following two properties.

The first is the termination of the updating of nodes in a program. The updating process propagates with the dependency relation between nodes. The process proceeds without stalling because the updating of each node always terminates. In other words, we can statically guarantee that *the iterations for reactive behavior will be repeated without stalling*.

© 2021 Information Processing Society of Japan

Second, the amount of memory required at runtime can be determined statically, making it possible to determine in advance whether the resources required to execute the program are available in the target environment. For example, in a bare-metal environment, because we cannot usually expect an OS functionality that kills processes causing memory shortage, we occasionally need to reset the hardware. Such behavior is generally difficult to debug and can remain as a serious defect. A program written with Emfrp allocates sufficient memory in advance based on the amount of memory required at runtime. We can then guarantee that the reactive behavior of the program will not be stuck due to errors caused by an insufficient amount of memory. Consequently, programs written in Emfrp can run safely in a bare-metal environment.

However, to achieve these properties, Emfrp prohibits recursive definitions when defining data types and functions. By prohibiting recursive functions, it is easier to guarantee the termination of node updates. In addition, prohibiting recursive data types makes it simpler to calculate the amount of memory required for program execution. Emfrp allows only primitive data types such as integer values and Boolean values, direct product types for tuples, and direct sum types whose contents can be determined by tags.

The problem with the current Emfrp is that it is difficult to naturally express data structures where the number of elements changes at runtime. Using the aforementioned direct product type, defining a data structure with a fixed number of elements is a straightforward process. However, defining a data structure that changes the number of elements at runtime, such as lists or trees, using the data types available in Emfrp, should be performed redundantly or abandoned. In general, in functional programming languages, where declarative descriptions are common as in FRP languages, such data structures are usually defined by recursive data types and are frequently used.

3.2 Motivating Examples

This subsection presents motivating examples to illustrate the usefulness of lists or trees even in small-scale embedded systems, and explains that such data structures require redundant descriptions in the current Emfrp.

DupCheck Module

`DupCheck` is an example Emfrp module that checks whether a value that is the same as the current input exists among the inputs obtained within the last certain period. In fact, the module maintains the history of the integer values obtained in the last four iterations. It checks whether the current input exists in the history and outputs the result as a Boolean value. **Figure 3** shows the Emfrp source code of `DupCheck`.

This module has two input time-varying values: the reset signal `reset` and an input integer value `v`. An optional integer type `OptI` is defined at the top of the module body. The input history is represented by node `history` as a 4-tuple of `OptI` data. The reason for using the option type is that in the first four iterations, `history` has elements without valid data. If `reset` is true, `history` is cleared, and only the latest input is retained. If `reset` is false, the update of `history` proceeds as follows. The first

```
1  module DupCheck
2  in  reset(False) : Bool, # reset signal
3      v(0)         : Int  # input value
4  out detect : Bool
5  use Std
6
7  type OptI = NoneI | SomeI(Int)
8
9  func check(x : Int, a : OptI): Bool =
10   a of NoneI -> False, SomeI(y) -> x == y
11
12 node init [(NoneI, NoneI, NoneI, NoneI)]
13   history: (OptI,OptI,OptI,OptI) =
14     if reset then (NoneI, NoneI, NoneI, SomeI(v))
15     else
16       history@last of (a1, a2, a3, a4) ->
17         (a2, a3, a4, SomeI(v))
18
19 node init [False] detect: Bool =
20   history@last of (a1, a2, a3, a4) ->
21     check(v,a1) || check(v,a2) || check(v,a3)
22     || check(v,a4)
```

Fig. 3 Implementation of the `DupCheck` module in Emfrp.

(oldest) element in `history` is deleted and shifted by one before the latest input is added to the end. The node `detect` determines whether the latest input value exists in the history. In the update expression of `detect`, the history is decomposed by pattern matching of tuples, and function `check` is used to find the desired value. Note that `history@last` is decomposed to refer to the past history.

In this program, because the history of inputs is represented as a tuple, many code pieces need to be modified to change the history size. Specifically, we should modify the type, initial value, and pattern match clause in the update expression of `history`, and the pattern match clause in the update expression of `detect`. After the code is modified thus, the variable names in the patterns must be rewritten appropriately. If all of them have the same type, as in `history`, it is difficult to detect human errors such as misplaced variables. Thus, programs that use tuples to represent variable-sized data are less maintainable and redundant, leading to the occurrence of errors.

Top10Sum Module

`Top10Sum` is an Emfrp module that outputs the sum of the 10 largest input values obtained so far. In this example, as in `DupCheck`, the input history is maintained to compute the result. **Figure 4** shows the Emfrp source code of `Top10Sum` that is also defined using tuples and an optional type.

Node `h` holds the 10 largest input values in a descending order. It is represented as a tuple of optional type data. When updating node `h`, it decides whether `v` should be inserted in the manner of an insertion sort by comparing the value of the elements in `h` with the input value `v`. The output node `sum` calculates the sum of the elements in `h`. Similar to `DupCheck`, this module has a redundant description.

In the aforementioned example, the time complexity of finding the insertion point of the input value `v` is O(Number of elements in the history). If we can use a tree structure (heap tree), the time required to find the minimum value in a set can be reduced. Thus, when the capacity of history increases, it is useful to manage it as a tree. However, it is difficult to represent such a tree structure using the data types (tuples and direct sums) available in the current Emfrp.

© 2021 Information Processing Society of Japan

```
1  module Top10Sum
2  in  reset(False) : Bool,
3      v(0) : Int           # input value
4  out sum : Int            # sum
5  use Std
6
7  type OptI = NoneI | SomeI(Int)
8  type TI10 = T10(OptI, OptI, OptI, OptI, OptI, OptI, OptI, OptI, OptI, OptI)
9
10 func le_opt(x : Int, a : OptI): Bool =
11   a of NoneI -> True, SomeI(y) -> x > y
12
13 func get_opt(x : OptI): Int =
14   x of NoneI -> 0, SomeI(y) -> y
15
16 node init [T10(NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI)]
17   h: TI10 = if reset then T10(NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI, NoneI)
18     else
19       h@last of T10(a0, a1, a2, a3, a4, a5, a6, a7 ,a8, a9) ->
20            if gt_opt(min, a0) then T10(SomeI(v), a0, a1, a2, a3, a4, a5, a6, a7, a8)
21       else if gt_opt(min, a1) then T10(a0, SomeI(v), a1, a2, a3, a4, a5, a6, a7, a8)
22       else if gt_opt(min, a2) then T10(a0, a1, SomeI(v), a2, a3, a4, a5, a6, a7, a8)
23       else if gt_opt(min, a3) then T10(a0, a1, a2, SomeI(v), a3, a4, a5, a6, a7, a8)
24       else if gt_opt(min, a4) then T10(a0, a1, a2, a3, SomeI(v), a4, a5, a6, a7, a8)
25       else if gt_opt(min, a5) then T10(a0, a1, a2, a3, a4, SomeI(v), a5, a6, a7, a8)
26       else if gt_opt(min, a6) then T10(a0, a1, a2, a3, a4, a5, SomeI(v), a6, a7, a8)
27       else if gt_opt(min, a7) then T10(a0, a1, a2, a3, a4, a5, a6, SomeI(v), a7, a8)
28       else if gt_opt(min, a8) then T10(a0, a1, a2, a3, a4, a5, a6, a7, SomeI(v), a8)
29       else if gt_opt(min, a9) then T10(a0, a1, a2, a3, a4, a5, a6, a7, a8, SomeI(v))
30       else h@last
31
32 node init [0] sum: Int = h of T10(a0, a1, a2, a3, a4, a5, a6, a7 ,a8, a9) ->
33   get_opt(a0) + get_opt(a1) + get_opt(a2) + get_opt(a3) + get_opt(a4)
34   + get_opt(a5) + get_opt(a6) + get_opt(a7) + get_opt(a8) + get_opt(a9)
```

Fig. 4 Implementation of the `Top10Sum` module in Emfrp.

These examples illustrate the drawback of Emfrp due to language restrictions, which makes it difficult to handle data structures whose size can change at runtime. This not only leads to redundancy in the source code, but may also adversely affect the performance of node update calculations.

4. Method

4.1 Approach

The introduction of recursive data types to Emfrp solves the problems described in the previous section. However, we cannot naïvely introduce recursive data types and recursive function definitions in the same way as in other functional languages. Assuming that we may use recursive data types and recursive functions without any restrictions, we can define the following nodes and functions:

```
1 # Node with recursive data type (lists for nats)
2 node init[Nil] r : IntList = Cons(x, r@last)
3 # Infinite recursion
4 func infloop(x : Int) = 1 + infloop(x)
5 node init[0] i : Int = infloop(0)
```

Node `r` is initialized to `Nil`, which is the end of the list, and it repeatedly adds the input value `v` to the head of the list each time `r` is updated. Therefore, the memory usage increases over time, resulting in memory shortage. As in node `i`, invoking a non-terminating function such as `infloop` prevents the termination of node updating. In these examples, the properties of Emfrp, such as statically determining the amount of runtime memory and guaranteeing the termination of node updating, are lost. Consequently, the naïve introduction of recursive definitions that cause these problems is unacceptable.

To address this issue, we introduce a new method for recursive definitions to Emfrp that can limit the size of constructible structures and the depth of recursive calls and propose a mechanism that statically detects the violation of the limitation. Using the proposed method to limit the size of constructible structures, we can statically detect a code that builds indefinitely expanding lists, as in the above example. In addition, by limiting the depth of recursive calls using this method, we can guarantee the termination of recursive functions and statically determine the amount of the call stack and other memory usages required at runtime.

4.2 Bounded Construction Types

In the proposed method, the maximum size of constructible data structures of a type is specified in the type. We define the size of a data structure as the number of constructors that form the structure and belong to its type. We use the following OCaml code to exemplify the definition. The sizes of `l` and `n` of type `ilist` are 4 and 1, respectively, and the size of `t` of type `otree` is 5:

```
1 type ilist = Nil | Cons of int * ilist
2 let l = Cons (1, Cons (2, Cons (3, Nil)))
3 let n = Nil
4
5 type otree = L | N of otree * int option * otree
6 let t = N (N (L, Some 2, L), Some 1, L)
```

Note that size is a positive integer, and only the number of constructors that belong to the target type is counted (e.g., we do not count `Some` in `t`).

We introduce new recursive data types augmented with size information, called *Bounded Construction Types* (BCTs). The size information associated with a BCT, called the *size parameter* of the type, indicates the *maximum size of constructible structures*

© 2021 Information Processing Society of Japan

of the type.

The size parameter of a BCT is expressed as a superscript on the type. For example, $\texttt{ilist}^4$ denotes the type of integer list up to four in length. Both `l` and `n` in the above example are of type $\texttt{ilist}^4$. As the sizes of `l` and `n` are 4 and 1, respectively, `n` is also of type $\texttt{ilist}^2$, whereas `l` is not. Similarly, `t` is of type $\texttt{otree}^8$ but not of type $\texttt{otree}^4$ because its size is five.

As described previously, for data structures of a type whose size can change at runtime, we introduce a limit on the number of constructors and express it as a part of the type. This allows us to statically detect objects whose size can exceed the given limit at runtime by performing type checking. In addition, by focusing on the size parameter of an argument of a recursive function, we can statically determine during type checking whether the size of the argument is strictly decreasing during the recursive call. If the size of the argument is guaranteed to decrease with each recursive call, then the depth of the recursive call is at most that size. This allows us to guarantee the termination of recursive functions statically. In other words, well-foundedness is required for recursive function calls.

The type checking of BCTs includes the extraction of constraints for checking the consistency of the possible sizes of the structure (size parameters), in addition to the generally performed type checking. By examining the validity of the constraints, we can check whether a given program violates the size of the data structure or the limit on recursive calls.

4.3 Emfrp$^{\text{BCT}}$

We propose Emfrp$^{\text{BCT}}$, which is an extension of a subset of Emfrp with BCTs. The basic execution model of Emfrp$^{\text{BCT}}$ is the same as that of Emfrp, although there are differences in syntax and runtime data representation. Functions in Emfrp may have polymorphic types, but we do not deal with them in this study because we focus on the size parameter. As higher-order functions are not available in Emfrp$^{\text{BCT}}$, the polymorphic functions do not generate size constraints on the data denoted by the type variables. Therefore, even if we were to introduce a polymorphism similar to Emfrp, it would be sufficient to use the same level of type checking for polymorphic functions as in the conventional method, and its implementation would not be difficult.

In Emfrp$^{\text{BCT}}$, if a program passes the type checking, the size information of the values in the program is guaranteed to be consistent. Using this information, we can statically estimate the amount of memory that the program will need at runtime by exhaustively traversing its syntax tree. Thus, although Emfrp$^{\text{BCT}}$ can use recursive data types (BCTs), it preserves the properties of Emfrp, such as the termination of the update process of time-varying values and the static estimation of memory usage at runtime. In the following sections, we explain Emfrp$^{\text{BCT}}$ for each feature.

Module Definition

The module definition includes the definition of data types, functions, input nodes, output nodes, and intermediate nodes, as in Emfrp. Although some type annotations can be omitted in Emfrp because of the type inference, all variables must be annotated in Emfrp$^{\text{BCT}}$.

Type Definition

We can use BCTs as recursive data types in Emfrp$^{\text{BCT}}$. Tuple types, direct sum types, and variant types that are available in Emfrp are emulated as non-recursive BCTs, hence they are not introduced directly into the language. The type definition for the BCT can be written as follows:

$$\texttt{type}\ \rho = \chi_1(T^\rho, \ldots, T^\rho) \mid \ldots \mid \chi_n(T^\rho, \ldots, T^\rho)$$

Here ρ is the type name, and χ is the constructor name. The type name and constructor name are unique in the program. We use T^ρ to represent the constructor parameter, which can be a basic type, ρ, or a BCT of another size. Here ρ plays a role of a placeholder to indicate which position in the parameter is recursive. Mutual recursions in BCT definitions are not allowed. In other words, when defining a recursive type, its own type name will always appear in the parameter of one of the constructors.

For example, an integer binary tree in BCT can be defined as follows:

```
type TreeI = LeafI | NodeI(TreeI, Int, TreeI)
```

When defining a BCT, other BCTs are allowed to appear in the constructor parameters. In such a case, the constant size parameter must be specified. As an example, the type definition of a binary tree, whose elements are list types up to size 10, is as follows:

```
type TreeL = LeafL | NodeL(TreeL, ListI[10], TreeL)
```

When the name of a BCT appears in its own type definition, it does not specify a size parameter, but when it appears in other type definitions or as a type name in function arguments, it specifies a size parameter by adding $[\psi]$ to the name. Here ψ is a meta-variable that indicates the size parameter. The size parameters are integer constants greater than or equal to zero, the sum and difference of the size parameters, and the size variable.

When the BCT constructor is applied to a value, the value has a type whose size is the sum of the sizes of the applied values plus one. For example the value `LeafI` in the above example has the type `TreeI[1]`, and `NodeI(NodeI(LeafI, 2, LeafI), 1, LeafI)` and `NodeI(LeafI, 3, NodeI(LeafI, 2, LeafI))` have the type `TreeI[5]`.

Function Definition

In addition to the function definitions allowed in Emfrp, Emfrp$^{\text{BCT}}$ allows recursive function definitions. Note that mutually recursive definitions are not allowed. Furthermore, there is a restriction that the arguments of recursive functions reduce in size during the recursive call.

The following defines a recursive function:

$$\texttt{func}\ f(\ x_1 : \pi_1, \ldots, x_n : \pi_n\ ) : \tau\ \texttt{where}\ \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$$
$$[\delta_1, \ldots, \delta_l] = e$$

where f is the function name, x_i is the parameter name, and π_i is a type that can be specified as a parameter of the function, which is either a basic type or a BCT with a size variable as a size parameter. The size variable introduced here is used in the

© 2021 Information Processing Society of Japan

adj expression (see below) of the function body. Here τ is the result type, $\mathcal{A}$ is a logical formula for the size variable and represents a precondition that must be statically checked to describe the function call, and δ is a size variable that appears in the parameters. The summation of size variable specified in $[\delta_1, \ldots, \delta_l]$ is the measure function of f and is used for verifying that the size of parameters passed to the recursive calls decreases. We use e to denote an expression of the function body. Reference to a node or to the previous value of a node is not allowed in the function definition.

The function definition is validated at type checking by judging of the implication of the precondition specified in the definition as the antecedent and the size constraint obtained from the body expression as the consequent. Here, the size decrease constraint on the recursive call is included in the constraint obtained from the body expression.

Expression

This section explains the expression for Emfrp$^{\mathrm{BCT}}$. The primitive binary operations are the same as in Emfrp. The local variable definitions are also similar, even though the syntax is different. We explain the expressions dealing with BCT, which are unique features of Emfrp$^{\mathrm{BCT}}$, and some of the constraints extracted from them.

From the function call $f(e, \ldots, e)$, the extracted constraint is the precondition of the function call as a constraint in addition to the usual type checking for the consistency of the argument types. If f is a recursive call, the constraint that the argument size is decreasing monotonically according to the measure function specified in the function definition is also extracted.

In the **if** expression whose result type is BCT, the result types of **then** and **else** clauses are checked, and the constraint that their sizes be equal is extracted. That is, for example, we must use **adj** expression (see below) to adjust the size when the result value of **then** clause is `Nil` and the result value of **else** clause is `Cons (1, Nil)`.

The decomposition expression **case** e **return** τ **of** $branch_1$ $|\ldots|$ $branch_n$ decomposes the BCT for each constructor. Here $branch_1, \ldots, branch_n$ must cover all the constructors of the target type. The τ denotes the type of the result. From the **case** expression, the constraint that the size of the result for all branches is equal to τ is extracted. Each *branch* is of the form $\chi(x_1{:}\pi, \ldots, x_n{:}\pi)$ `->` e. In each branch expression, the value of the constructor is bound to a variable name with a type annotation. The type annotations are the same as for function arguments, and new size variables can be introduced, and unused type annotations can be omitted. This new size variable has a constraint related to the size of the expression to be decomposed. We illustrate the size constraints extracted from the branch using the `TreeL` type described previously. When decomposing the values of `TreeL[5]` by **case** expressions, the branch of `NodeL` is of the form `NodeL(l: TreeL[a],v: ListI[b], r: TreeL[c]) ->` e. In this case, the newly introduced size variable requires $a > 0 \wedge b > 0 \wedge c > 0 \wedge 1 + a + c = 5 \wedge b = 10$. Therefore, from this branch the constraint $\forall a, b, c. a > 0 \wedge b > 0 \wedge c > 0 \wedge 1 + a + c = 5 \wedge b = 10 \rightarrow C \wedge$ (constraint on the size of the result) is extracted, where C is the size constraint obtained from e. When the BCT values are produced, the information about the size of the component BCTs is absent, so the size constraint of the **case** expression reconstructs this information.

The size expansion expression e **adj**`[`ψ`]` adjusts the size of the BCT of e to ψ. As the size of a BCT indicates the maximum number of the elements of the type, it is safe to consider a small size value as a large size value. From this expression, the constraint (the size of e) $\leq \psi$ is extracted.

A BCT value in Emfrp$^{\mathrm{BCT}}$ contains runtime size information. The size-cast expression **fit** e_0 **to** $x{:}\rho^k$ `->` e_1 `|` **fail** `->` e_2 branches the computation depending on the runtime size of e_0. If the runtime size of e_0 is less than or equal to the constant size k, the size-cast expression casts e_0 to the type ρ^k and bind the cast value to x which can be used in e_1. If not, the expression e_2 in the **fail** clause is executed. This expression is mainly designed for use with the previous value of a node in a node update expression.

Node Definition

We can define nodes in the same way as in Emfrp. Node definitions cannot introduce new size variables as function definitions because node definitions do not have arguments. Therefore, only the size variables introduced in the **case** branch can be used.

Memory Usage Estimation

In Emfrp, memory usage at runtime of a program is estimated by traversing the syntax tree of the program code and counting the expressions that produce values such as constant values and constructor applications. As there are no recursive calls in Emfrp, all possible node updates can be covered by expanding all function calls in the syntax tree.

Similar to Emfrp, Emfrp$^{\mathrm{BCT}}$ estimates memory usage by traversing the syntax tree and tracking the production of values. Memory usage is estimated for modules that pass the type and constraint checks. When traversing the syntax tree of an expression, the concrete values are assigned to the size variables that appear in the expression. For each function call, it computes the concrete value of the introduced size variable, assigns it to the size variable, and expands the syntax tree. The same function may be expanded by recursive calls, but the expansion of the syntax tree stops after a finite number of times because a monotonic decrease in size is guaranteed.

In **case** expression, each branch is traversed exhaustively for all possible sizes. In a branch of **case**, a size variable is introduced under the constraint on the size of the type to be decomposed. The branch expression is traversed by assigning all combinations of valuations that satisfy the constraints of the size variables introduced. When decomposing the values of BCTs, the information about the sizes of the component BCTs is absent, so the information is reconstructed by enumerating all the size tuples that satisfy the constraints. In addition, if there is no assignment that can satisfy the constraint, the branch is not traversed.

Finally, the memory usage for a node update is estimated by traversing the syntax tree starting from the node update expression and assigning concrete values to the size variables. Three categories of memory are required to continue executing the module: the memory region that holds the current and previous values of the nodes, the largest memory region required to update a single node, and the memory region required for the language

© 2021 Information Processing Society of Japan

runtime.

4.4 Examples

We implemented the two modules presented in Section 3 in Emfrp^{BCT}. Using these examples, we explain the extracted size constraints and traversal of the syntax tree to estimate memory usage.

DupCheck Module

Figure 5 shows the `DupCheck` module in Emfrp^{BCT}. The input of the module is the same as in the Emfrp example. Line 6 defines a BCT of type list with `Int` as an element. Lines 8–24 define the auxiliary functions for the list. Lines 26–31 define a node `history` of type `List[5]` representing the history of inputs. Note that the size is set to five to hold four histories, including `Nil` at the end of the list. In the node update expression of `history`, the runtime size of the list is obtained using the **`fit`** expression, with the branch by its size. If there is enough space in `history`, the input values are simply added to it. Otherwise, the first value of `history` is removed, and the input value is added to the end. Line 33 defines node `detect`, which checks whether the current input value exists in the previous history.

By representing the history using a list, Fig. 5 is more maintainable than Fig. 3 because the code is independent from the specific history length. In addition, because the consistency of the size is statically checked when the history length is modified, errors related to size caused by the modification can be detected.

We use the `insert` function as an example to explain the constraints that are extracted when validating a recursive function definition. First, a new size variable `m` is introduced from the argument of the `insert` function. A precondition, $m > 0$, is required when calling this function.

Constraints for function definition expressions are extracted bottom-up. In the `Nil` branch, $\top \rightarrow 2 \leq m + 1 \wedge m + 1 = m + 1$ is extracted from the constraints for the constructor application, **`adj`** expression, and the size of the result. Let X be this extracted constraint. Next, in the `Cons` branch, a new size variable `n` is introduced under the condition $n > 0 \wedge m = 1 + n$ to decompose the variable `l` of size m. Focusing on the recursive call `insert(x, t)` in the following expression, the conjunction of the precondition for the function call ($n > 0$) and the constraint on the recursive call ($n < m$) is extracted. The constraint on the recursive call is specified by the measure function `[m]` at the end of line 9. This implies that the total size of the actual arguments of BCT is less than `m`. In other words, $n > 0 \wedge n < m$ is extracted from this recursive call. Noting the size of the constructor-applied result of the recursive call and the size of the branch result, we obtain the constraint $1 + (n + 1) = m + 1$. Thus, the constraint extracted from the entire `Cons` branch is $\forall n.(n > 0 \wedge m = 1 + n) \rightarrow (n > 0 \wedge n < m \wedge 1 + (n + 1) = m + 1)$. Let Y be this constraint.

Given the above constraints and the requirement that the size of the function result be $m + 1$, the size constraint to be checked in function validation is $\forall m.(m > 0) \rightarrow (X \wedge Y \wedge m + 1 = m + 1)$. Note that the constraints shown here are not the actual constraints extracted during the real type checking. The actual constraints are more complicated than those shown here because of the addition of tautological constraints ($\top$) for variable references, but they

```
 1  module DupCheck
 2  in  reset  : Bool init (False), # reset signal
 3      v      : Int  init (0)      # input value
 4  out detect : Bool
 5
 6  type List = Nil | Cons(Int, List)
 7
 8  func insert(x: Int, l: List[m]): List[m+1]
 9    where {m > 0} [m] =
10    case l return List[m+1] of
11    | Nil -> Cons(x, Nil) adj[m+1]
12    | Cons(h:Int,t:List[n]) -> Cons(h,insert(x,t))
13
14  func search(x: Int, l: List[m]): Bool
15    where {m > 0} [m] =
16    case l return Bool of
17    | Nil -> False
18    | Cons(h: Int, t: List[n]) ->
19        if x = h then True else search(x,t)
20
21  func tail(l: List[m]): List[m-1] where {m-1 > 0} =
22    case l return List[m-1] of
23    | Nil -> Nil adj[m-1]
24    | Cons(h: Int, t: List[n]) -> t
25
26  node history: List[5] init (Nil adj[5]) =
27    if reset then Cons(v, Nil) adj[5]
28    else
29      fit history@last to
30      | hl: List[4] -> insert(v,hl)
31      | fail -> insert(v,tail(history@last))
32
33  node detect: Bool init (False) =
34    search(v, history@last)
```

Fig. 5 Implementation of the `DupCheck` module in Emfrp^{BCT}.

are essentially the same. For other function and node definitions, similar type checking and constraint extraction are performed to validate the definitions.

When estimating the memory usage for node updates, the value of the size variable newly introduced by decomposing list structures is determined uniquely. Let m be the length of a list and n be the length of the tail of the list satisfying the condition $m = 1 + n$. When m is given, n is determined uniquely, and thus each branch of **`case`** is traversed only once. In other words, no backtracking is required, in contrast to the following example. For example, in the case of the `insert` function, each time the body of the function is traversed recursively, the size variable m is assigned $4, 3, 2, 1$ in that order.

Top10Sum Module

Figure 6 shows `Top10Sum` module in Emfrp^{BCT}. In the previous example (Fig. 4), we used tuples to manage the values, but in this example we use a leftist heap [25]. The input and output of the module are the same as those in the Emfrp example. Lines 8–9 define the type of the heap tree. The third and fourth parameters of the constructor `T` are recursive. Lines 11–36 define the auxiliary functions of the leftist heap. The node `h` defined in lines 39–43 is a heap tree of size 21. This size can accommodate 10 values. When updating the `h` node, if there is space for additional elements, the input value is inserted into the heap. If not, the input value is compared with the minimum value in the heap. If the input value is larger than the minimum value, the input value is included in the top 10, so the input value replaces the minimum value; otherwise, the current state is retained. The insertion into the heap and the deletion of the minimum can be performed with a time complexity of $O(\log(\text{number of elements in the heap}))$, and the minimum of the

© 2021 Information Processing Society of Japan

```
 1  # The sum of the top 10 of the input data
 2  module Top10Sum
 3  in  reset: Bool init (False),
 4      x:     Int init (0)       # input value
 5  out sum: Int    # sum
 6
 7  # Leftist Heap
 8  type Heap = E                  # terminal
 9       | T(Int,Int,Heap,Heap) # (rank,v,left,right)
10
11  func rank(e: Heap[n]) : Int where {n > 0} =
12    case e return Int of E -> 0 | T(r, x, a, b) -> r
13
14  func make(x:Int,a:Heap[n],b:Heap[m]): Heap[1+n+m]
15    where {n > 0, m > 0} =
16    let ra = rank(a) in let rb = rank(b) in
17    if ra > rb then T(rb+1,x,a,b) else T(ra+1,x,b,a)
18
19  func merge(h1:Heap[n],h2: Heap[m]): Heap[n+m-1]
20    where {n > 0, m > 0} [n,m] =
21    case h1 return Heap[n+m-1] of E -> h2 adj[n+m-1]
22    | T(r1, x1, a1, b1) ->
23      case h2 return Heap[n+m-1] of E->h1 adj[n+m-1]
24      | T(r2, x2, a2, b2) ->
25        if x1 <= x2 then make(x1, a1, merge(b1, h2))
26          else make(x2, a2, merge(h1, b2))
27
28  func insert(x: Int, h: Heap[n]): Heap[n+2]
29    where {n > 0} = merge(T(1, x, E, E), h)
30
31  func findMin(h: Heap[n]): Int where {n>0} =
32    case h return Int of E -> 0 | T(r, x, a, b) -> x
33
34  func delMin(h:Heap[n]):Heap[n-2] where {n>2} =
35    case h return Heap[n-2] of E -> E adj[n-2]
36    | T(r, x, a, b) -> merge(a, b)
37
38
39  node h: Heap[21] init (E adj[21]) =
40    if reset then E adj[21] else
41    fit h@last to hl: Heap[19] -> insert(x, hl)
42    | fail -> if x <= findMin(h@last) then h@last
43              else insert(x, delMin(h@last))
44
45  node sum : Int init (0) = if reset then 0 else
46    let diff = fit h@last to
47        | hl : Heap[19] -> x
48        | fail ->
49          if x <= findMin(h@last) then 0
50            else x - findMin(h@last))
51    in sum@last + diff
```

Fig. 6 Implementation of the Top10Sum module in Emfrp^{BCT}.

heap can be obtained with a time complexity of $O(1)$. Lines 45–51 define node sum, which calculates the sum. When a value is inserted into the heap, it calculates the difference between the input value and the minimum value in the heap, and updates the total value by adding it to its own previous value. This update process can be done with complexity $O(1)$. Overall, the module requires a time complexity of $O(\log(\text{number of elements in the heap}))$ to update the nodes. Therefore, this program is more time efficient than the Emfrp example, which takes linear time to update and even if the number of elements is increased, the update process is less time-consuming.

The recursive function in this program is the merge function defined in lines 19–26. When this function is invoked, only one of the arguments decreases in size. Thus, we set the measure function as [n,m] and check that the sum of the sizes of the two arguments is decreasing monotonically. This check guarantees the termination of the merge function.

When estimating the memory usage of a node update, an exhaustive search for the shape of the tree is performed in the T branch of the heap tree decomposition by the **case** expression. To decompose a tree of size 21, (m, n), the third and fourth size parameters of the constructor T, have the constraint $21 = 1+m+n$. Therefore, $(m, n) = (1, 19), (2, 18), (3, 17), \ldots, (18, 2), (19, 1)$ are assigned as the sizes in order, and then the syntax tree is traversed.

The proposed memory usage estimation algorithm enumerates all possible structures of the data type (size assignment) while searching, which may lead to a combinatorial explosion. As Emfrp^{BCT} is designed for small embedded systems, the proposed algorithm is expected to be applied to data types with relatively small sizes. The execution time of the proposed algorithm is expected to be practical in most cases, but considering larger data types is a topic for future work.

5. Formalization

In this section, we formalize the syntax, operational semantics, and type system of Emfrp^{BCT}. We then propose an estimation algorithm for memory usage and demonstrate its soundness, that is, the estimated amount of memory allows us to update all the nodes.

5.1 Syntax

We define the syntax as **Fig. 7**. Most of the syntax is similar to the concrete syntax of Emfrp^{BCT} described in Section 4. It differs from the concrete syntax in the following ways. First, the position of the size parameter is a superscript of the type name. Second, a type annotation must be added whenever a variable is introduced in a branch of a **case** expression. In addition, output nodes are not declared explicitly in the module.

In addition to the syntactic rules shown in the figure, there are several other syntax requirements. First, the branches of the **case** expression must cover all the constructors of the type to be decomposed. Second, the function definitions should not contain node references or previous value references. Third, the names of the types, constructors, functions, variables, and size variables must be unique. Finally, the node and function definitions should be sorted according to the orders defined in the following.

Definition 1 (Dependency order on node definitions)**.** For nodes N and N', we define $N >_N N'$ if the update expression of a node N (the second element of $\mathcal{N}(N)$) contains a reference to a node N', i.e., N depends on N'. As there is no circular dependency of nodes, the transitive closure $>_N^*$ of $>_N$ is a strict partial order.

Definition 2 (Dependency order on function definitions)**.** For functions $f, g \in dom(\mathcal{F})$, we define $f \geq_{\mathcal{F}} g$ if the body of f contains a call to g, i.e., f depends on g, or $g = f$. As mutual recursions are not allowed, the transitive closure $\geq_{\mathcal{F}}^*$ of $\geq_{\mathcal{F}}$ is a partial order.

The syntax of size constraints is as follows. We use k and δ to denote a size constant and a size variable, respectively. A size variable is a variable of a size parameter that takes a positive integer. A size constraint C is extracted by type checking and can be the true literal, an arithmetic constraint, a universal quantification over size variables, a conjunction of constraints, or an implication of constraints. Here $\mathcal{A}$ is a precondition for the function call specified in its definition, and ψ is a size parameter for BCTs.

Next, we explain types in Emfrp^{BCT}. We use $\mathcal{B}$ to denote a base type, which is either an integer or Boolean type. Here ρ denotes a name of a BCT. There are several variants of types because the

© 2021 Information Processing Society of Japan

Size Constraints

$$\begin{array}{llll}
k & \in & \mathbb{N} \quad \delta \in \mathit{SizeVar} & \\
\mathcal{C} & \in & \mathit{Constraint} & ::= \top \mid \mathcal{A} \mid \forall\delta.\mathcal{C} \mid \mathcal{C} \wedge \mathcal{C}' \mid \mathcal{C} \rightarrow \mathcal{C}' \\
\mathcal{A} & \in & \mathit{ArithCon} & ::= \psi = \psi' \mid \psi < \psi' \mid \psi \leq \psi' \\
\psi & \in & \mathit{SizeParam} & ::= k \mid \delta \mid \psi + \psi' \mid \psi - \psi'
\end{array}$$

Types

$$\begin{array}{llll}
\mathcal{B} & \in & \mathit{BaseType} & ::= \texttt{Bool} \mid \texttt{Int} \\
\rho & \in & \mathit{TypeName} & \\
\pi & \in & \mathit{VarType} & ::= \mathcal{B} \mid \rho^{\delta} \\
\sigma & \in & \mathit{ConstType} & ::= \mathcal{B} \mid \rho^{k} \\
T^{\rho} & \in & \mathit{ElemType} & ::= \mathcal{B} \mid \rho \mid \rho'^{k} \quad (\rho' \neq \rho) \\
\tau & \in & \mathit{Type} & ::= \mathcal{B} \mid \rho^{\psi}
\end{array}$$

Expressions

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

Module Definitions

$$\begin{array}{llll}
M & \in & \mathit{Module} ::= (\mathcal{T}, \mathcal{F}, \mathcal{I}, \mathcal{N}, \mathit{Init}) & \\
\mathcal{I} & ::= & \overrightarrow{N : \sigma} & \text{(Input Nodes)} \\
\mathcal{T} & ::= & \cdot \mid \mathcal{T}, \rho \mapsto \overrightarrow{\chi(\overrightarrow{T^{\rho}})} & \text{(Type Definitions)} \\
\mathcal{N} & ::= & \cdot \mid \mathcal{N}, N \mapsto (\sigma, e) & \text{(Node Definitions)} \\
\mathcal{F} & ::= & \cdot \mid \mathcal{F}, f \mapsto (\overrightarrow{x{:}\pi}){:}\tau\ \texttt{where}\ \{\overrightarrow{\mathcal{A}}\}[\overrightarrow{\delta}] = e & \\
 & & & \text{(Function Definitions)} \\
\mathit{Init} & ::= & \cdot \mid \mathit{Init}, N \mapsto ce & \text{(Initial Values)}
\end{array}$$

Fig. 7 Syntax of $\text{Emfrp}^{\text{BCT}}$.

allowed forms differ depending on the context. In particular, T^{ρ} is a type of constructors of ρ, and ρ without a size parameter is used to declare ρ as the recursive type.

For expressions, **case** expressions should be annotated with the result types. Nested pattern matching is not handled. Here *ce* represents a constant expression, typically used for specifying the initial value of a node. Although not declared explicitly here, only integer values and constructor calls are allowed in constant expressions. A module consists of five maps: type definitions, function definitions, input nodes, intermediate or output nodes, and initial values for these nodes. BCTs declared in the concrete syntax with **type** are stored as the entries from type names to their constructors.

5.2 Operational Semantics

Figure 8 shows the definition of the values, the environments, and the auxiliary functions for the recursive indices. Here l represents the location of the heap area, and v represents a value that can be an integer, a Boolean constant, or a BCT value. The BCT value has the tag of the constructor, size information at runtime, and a list of positions representing parameters. We use E to denote the environment from the local variable names to positions, H to denote the environment from positions to values in the heap area, and $\mathcal{L}$ to denote the environment from node names to positions in the heap. The previous-value reference of a node is also referenced from $\mathcal{L}$. Here Γ is the type environment and is used for type checking, and Δ is the environment from size variables to size constants and is used to estimate memory usage. The auxiliary function $I(\chi)$ returns the set of indices that appear in the recursive type on the parameters of constructor χ.

Values

$$\begin{array}{llll}
l & \in & \mathit{Location} & \\
v & \in & \mathit{Value} & ::= c^{\mathcal{B}} \mid \chi[k](\overrightarrow{l})
\end{array}$$

Environments

$$\begin{array}{llll}
E & ::= & \cdot \mid E, x \mapsto l & \text{(Local Var. Env.)} \\
H & ::= & \cdot \mid H, l \mapsto v & \text{(Heap)} \\
\mathcal{L} & ::= & \cdot \mid \mathcal{L}, N \mapsto l \mid \mathcal{L}, N\texttt{@last} \mapsto l & \text{(Node Locations)} \\
\Gamma & ::= & \cdot \mid \Gamma, N : \tau \mid \Gamma, x : \tau & \text{(Type Env.)} \\
\Delta & ::= & \cdot \mid \Delta, \delta \mapsto k & \text{(Size Var. Env.)}
\end{array}$$

Recursive Indices

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

Fig. 8 Values, environments, and recursive indices.

The operational semantics of the expressions are shown in **Fig. 9**. As $\text{Emfrp}^{\text{BCT}}$ is a language that focuses on resource usage during computation, some of the resource requirements are made explicit in the semantics. The explicit resources are the number of local variables to be referenced, the number of data to be stored in the heap, and the number of function calls. The heap, local variable environment and call stack have free spaces. A free space is an integer which means the number of their unused entries. Here $[s]E$ denotes the local variable environment E with the free space s. For example, we can add at most 5 entries to $[5]E$. The operational semantics are indicated by $[s]E \mid [t]H \vdash_{\mathcal{L}}^{\mathcal{T};\mathcal{F}} e \Downarrow_u l'; [t']H'$. This is read as "with type definitions $\mathcal{T}$, function definitions $\mathcal{F}$, and a node location environment $\mathcal{L}$, evaluating an expression e under a local variable environment E with free space s, a heap H with free space t, and a call stack with free space u results in a heap H' with free space t' which stores the result of evaluation at l'." Note that the evaluation result of the expression does not directly represent a value, but a position in the heap after evaluation. The domain of $\mathcal{L}$ used in the evaluation of a node is the union of the domains of $\mathcal{N}$ and $\mathcal{I}$. Note that Emfrp has a restriction that the current or previous value of a node cannot be referenced inside the function. To reflect this restriction in $\text{Emfrp}^{\text{BCT}}$, the node location environment $\mathcal{L}$ is changed to $\cdot$, which represents an empty environment, in the evaluation of the function body (rule E-CALL).

Here E corresponds to a stack frame for local variables, and its free space is represented by s. Similarly, H and H' are the heap areas for storing values, and their free spaces are represented by t and t', respectively. In addition, u indicates the free space of the call stack, that is, the possible number of function calls. Each free space s, t, t', and u is a non-negative integer. If any free space becomes negative during the evaluation, the evaluation is stuck (i.e., abnormal termination). Furthermore, the evaluation fails to proceed even when a partial function used as an auxiliary function, such as a reference to the environment, is applied to a value that is not in its domain.

During the evaluation of an expression, values are stored in the heap, but once stored, the values and their locations are not changed. In other words, free space on the heap cannot be increased in the absence of GC or other operations during the evaluation of an expression. Therefore, in the evaluation of an expression, the heap H' after the evaluation includes the heap H before the evaluation ($H \subseteq H'$) and the free space t' of the heap after the evaluation is the same size or smaller than the free space t of the

© 2021 Information Processing Society of Japan

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 9 Operational semantics for expressions.

heap before the evaluation ($t \ge t'$).

The environment E maintains the locations that must be stored locally. Rules E-CASE, E-LET, and E-FIT-SUCCESS introduce a new local variable, so binding is added to E by reducing the free space during the evaluation of the subsequent expression. In the rules E-CALL and E-OP, functions and binary operations are applied after the arguments are evaluated, but it is necessary to temporarily store the position of the result of the argument evaluation. This process can be understood as introducing local variables that cannot be referenced by other expressions. Therefore, when evaluating the second and subsequent arguments, the local variable environment is set to $[s-(i-1)]E$ in E-CALL and $[s-1]E$ in E-OP to reduce the free space. Meanwhile, rule E-CTOR evaluates the parameters of the constructor assuming that the location where the result in the heap is stored is allocated as an implicit local variable. As the locations of the evaluation results of these parameters are described sequentially in the allocated locations in the heap, the evaluation is performed without allocating local variables for the number of parameters as in a function call.

The rest of the rules are similar to those of a typical call-by-value functional language. The size parameters described in the program are not handled during evaluation, except in the **fit** expression. In the **fit** expression, a branch of evaluation is performed using the runtime size information of the object, but the comparison is made only with a constant, and no size variable appears.

5.3 Type System

The definition of the auxiliary operators used in the typing rules is given in **Fig. 10**. Here $\cdot \sim \cdot$ is an operator that generates the size constraints required to unify the types. It generates constraints that are constant true if both sides are of the same basic type, and constraints that the size parameters are equal if both sides are BCTs. The last pattern generates constraints for recursive types that appear in the constructor (as placeholders). The operator $\tau \downarrow$ extracts the size parameter from the type. The third set of rules is

© 2021 Information Processing Society of Japan

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

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

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

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

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

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

$$\boxed{[\overrightarrow{\pi} \mapsto \overrightarrow{\tau}]}$$

$$\begin{array}{l} [\pi_1, \ldots, \pi_n \mapsto \tau_1, \ldots, \tau_n] \quad = \\ \quad [\{\, \delta \mapsto \psi \mid 1 \leq i \leq n, \pi_i \sim \tau_i = (\delta = \psi) \,\}] \\ \quad\quad (\forall i \in 1 \ldots n, \pi_i \sim \tau_i \text{ is defined}) \end{array}$$

Fig. 10 Auxiliary operators for types and constraints: unification for types ($\sim$), size parameter for types ($\downarrow$), and size substitution (over types).

the assignment rules for the size parameters. The last set of rules generates a size parameter assignment from a pair of sequences of types. It unifies the corresponding types with respect to size and generates a size parameter assignment $\delta \mapsto \psi$ if the resulting constraint has the form $\delta = \psi$. Note that the constraint obtained by unification may not have the form $\delta = \psi$ because the types of the form π include primitive types.

The typing rules for expressions are shown in **Fig. 11**. Typing rules are described with type judgments $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$. This is read as "with type definitions $\mathcal{T}$ and function definitions $\mathcal{F}$, an expression e in a function f under a type environment Γ has type τ and a size constraint C is extracted." The typing rule for a branch of a **case** expression is represented by the type judgment $\Gamma \vdash_{f,\psi}^{\mathcal{T};\mathcal{F}}$ *branch* $:\sim \tau \mid C$. This is read as "with type definitions $\mathcal{T}$ and function definitions $\mathcal{F}$, a branch *branch* that decomposes an expression of size ψ in a function f under a type environment Γ has type τ and a size constraint C is extracted." In each type judgement, a function name f is passed to indicate the function being type-checked. For update expressions and initial values of nodes, a special name $-$ representing an empty is passed as the function name.

The size constraint C extracted along with type checking consists of addition, subtraction, and comparison of positive integers, universal quantification over size variables, conjunctions, and implications, as presented in Section 5.1. This is included in the theory known as Presburger arithmetic. The validity of the Presburger arithmetic expression is known to be decidable. Therefore, the validity of the constraint is also decidable.

Here we explain some typing rules. In rule T-CTOR, each type of constructor parameter is typed, and then the size of the result is computed using the information about the recursive type.

In rule T-CASE, the constraint of the expression is a conjunction of that obtained from the decomposed expression and those obtained from the branches. Each branch is examined by the rule T-BRANCH. The size parameters are extracted from each of the constructor parameters, and the constraint for these parameters is introduced as C. The constraint C is composed of a constraint $\mathcal{R}$ derived from the size of the type of the decomposed expression and a series of constraints for the sizes of other BCTs. These constraints, with the constraint C' derived from the body of the branch and the constraint $\tau' \sim \tau$ on the size of the result, are used to generate the final constraint.

Rule T-CALL performs type checking and constraint extraction for function calls. For type checking, it performs type checking of function calls of the general first-order functions. For constraint extraction, we first generate a size variable assignment θ from the size variables of the parameters of the function definition obtained from $\mathcal{F}$ and the size parameters of the arguments. The precondition $\mathcal{A}$ of the function call is appropriately replaced by θ. If the call is recursive ($f = g$), we generate the constraint $\mathcal{R}$ that the result of the measure function substituted by θ (the size at the recursive call) is less than the measure of the function parameters (the size at the beginning of the function). This constraint guarantees the termination of recursive calls.

The rule T-IF adds a constraint that the size of the result of the **then** and **else** clauses must match ($\tau \sim \tau'$).

Figure 12 shows the size constraint for the validity of a module definition. These rules generate constraints to be checked for function definitions, node definitions, and node initial values based on the size constraints obtained from their defining expressions. For example, in the case of function definitions, the generated constraints are implications universally quantified over size variables introduced with parameters of functions, whose assumptions are the precondition of the function calls, and whose consequences are the constraints obtained from their bodies. The validity of these constraints implies that the function definitions are valid in the module. The validity of the entire module can be checked by checking similarly the validity of the node definitions and the initial values of nodes.

5.4 Memory Usage Estimation Algorithm

In **Fig. 13**, we define the auxiliary function used in the algorithm to estimate the amount of memory used. It computes the value of the size parameter under the size variable environment Δ.

Next, **Fig. 14** demonstrates the algorithm for estimating the amount of memory used in the evaluation of an expression. The algorithm comprises an algorithm $\mathcal{M}$ for traversing expressions and an algorithm $\overline{\mathcal{M}}$ for searching parameters in the **case** branch. In the algorithm, the numerical max function is described by the binary operator $\sqcup$.

The algorithm $\mathcal{M}$ takes as inputs a type definition $\mathcal{T}$, a function definition $\mathcal{F}$, a size environment Δ, and a type environment Γ, and outputs the type τ of e, the free space s of the local variable environment, free space t of the heap, and free space u of the call stack required for the evaluation of e. It computes the amount of memory that can be used by subexpressions of the target expression by scanning the syntax tree, accompanied by the size environment, to obtain the memory required by the target ex-

© 2021 Information Processing Society of Japan

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

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

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

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_0 : \rho^\psi \mid \mathcal{C}_0 \qquad \left[\ \Gamma \vdash_{f,\psi}^{\mathcal{T};\mathcal{F}} \mathit{branch}_i :\sim \tau \mid \mathcal{C}_i\ \right]_{i \in 1\ldots n}}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{case}\ e_0\ \texttt{return}\ \tau\ \texttt{of}\ \{\mathit{branch}_i\}_{i \in 1\ldots n} : \tau \mid \bigwedge_{i \in 0\ldots n} \mathcal{C}_i}\ (\text{T-CASE})$$

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

$$\frac{\begin{array}{c} \mathcal{F}(g) = (x_1 {:} \pi_1, \ldots, x_n {:} \pi_n) {:} \tau\ \texttt{where}\ \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}\texttt{[}\ \overrightarrow{\delta}\ \texttt{]} = e \\ \begin{array}{c} \left[\ \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_i : \tau_i \mid \mathcal{C}_i\ \right]_{i \in 1\ldots n} \\ \theta = [\pi_1, \ldots, \pi_n \mapsto \tau_1, \ldots, \tau_n] \end{array} \qquad \mathcal{R} = \begin{cases} \sum_{\delta \in \overrightarrow{\delta}} (\theta\delta) < \sum_{\delta \in \overrightarrow{\delta}} \delta & (f = g) \\ \top & (\text{otherwise}) \end{cases} \end{array}}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} g(e_1, \ldots, e_n) : \theta\tau \mid \bigwedge_{i \in 1\ldots n} \mathcal{C}_i \wedge \bigwedge_{i \in 1\ldots m} (\theta\mathcal{A}_i) \wedge \mathcal{R}}\ (\text{T-CALL})$$

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1 : \texttt{Bool} \mid \mathcal{C}_1 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_2 : \tau \mid \mathcal{C}_2 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_3 : \tau' \mid \mathcal{C}_3}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \mid \tau \sim \tau' \wedge \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3}\ (\text{T-IF})$$

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

$$\frac{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_1 : \rho^\psi \mid \mathcal{C}_1 \qquad \Gamma, (x : \rho^k) \vdash_f^{\mathcal{T};\mathcal{F}} e_2 : \tau \mid \mathcal{C}_2 \qquad \Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e_3 : \tau' \mid \mathcal{C}_3}{\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} \texttt{fit}\ e_1\ \texttt{to}\ x {:} \rho^k \to e_2 \mid \texttt{fail} \to e_3 : \tau \mid \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3 \wedge \tau \sim \tau'}\ (\text{T-FIT})$$

$$\boxed{\Gamma \vdash_{f,\psi}^{\mathcal{T};\mathcal{F}} \mathit{branch} :\sim \tau \mid \mathcal{C}}$$

$$\frac{\begin{array}{c} \chi(T_1^\rho, \ldots, T_n^\rho) \in \mathcal{T}(\rho) \\ \overrightarrow{\delta} = \{\pi_i \downarrow\ \mid i \in 1\ldots n, \pi_i \downarrow \in \mathit{SizeVar}\} \\ \Gamma, x_1 {:} \pi_1, \ldots, x_n {:} \pi_n \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau' \mid \mathcal{C}' \end{array} \qquad \begin{array}{c} \mathcal{R} = \begin{cases} \psi = 1 + \sum_{i \in I(\chi)} \pi_i \downarrow & (I(\chi) \neq \emptyset) \\ \top & (\text{otherwise}) \end{cases} \\ \mathcal{C} = \mathcal{R} \wedge \bigwedge_{i \in 1\ldots n} T_i^\rho \sim \pi_i \end{array}}{\Gamma \vdash_{f,\psi}^{\mathcal{T};\mathcal{F}} \chi(x_1 {:} \pi_1, \ldots, x_n {:} \pi_n) \to e :\sim \tau \mid \forall \overrightarrow{\delta}. \mathcal{C} \to (\mathcal{C}' \wedge \tau' \sim \tau)}\ (\text{T-BRANCH})$$

Fig. 11 Typing rules for expressions.

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

$$\frac{\left[\begin{array}{c} \mathcal{F}(f) = (x_1 {:} \pi_1, \ldots, x_n {:} \pi_n) {:} \tau\ \texttt{where}\ \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}\texttt{[}\ \overrightarrow{\delta}\ \texttt{]} = e \\ x_1 {:} \pi_1, \ldots, x_n {:} \pi_n \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau' \mid \mathcal{C}'_f \\ \mathcal{C}_f = \forall \overrightarrow{\pi_i \downarrow}. (\bigwedge_{i \in 1\ldots m} \mathcal{A}_i) \to (\mathcal{C}'_f \wedge \tau \sim \tau') \end{array}\right]_{f \in \mathrm{dom}(\mathcal{F})}}{\mathcal{F} \Longrightarrow \bigwedge_{f \in \mathrm{dom}(\mathcal{F})} \mathcal{C}_f}\ (\text{C-FUNC})$$

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

$$\frac{\mathrm{dom}(\mathcal{N}) = \{N_1, \ldots, N_n\} \qquad \left[\begin{array}{c} \mathcal{N}(N_i) = (\sigma_i, e_i) \\ \mathcal{I}, N_1 {:} \sigma_1, \ldots, N_n {:} \sigma_n \vdash_{_}^{\mathcal{T};\mathcal{F}} e : \tau_i \mid \mathcal{C}'_i \\ \mathcal{C}_i = \mathcal{C}'_i \wedge \tau_i \sim \sigma_i \end{array}\right]_{i \in 1\ldots n}}{\mathcal{N} \Longrightarrow \bigwedge_{i \in 1\ldots n} \mathcal{C}_i}\ (\text{C-NODE})$$

$$\boxed{\mathit{Init} \Longrightarrow \mathcal{C}}$$

$$\frac{\left[\begin{array}{c} \mathcal{N}(N) = (\sigma, e) \\ \cdot \vdash_{_}^{\mathcal{T};\mathcal{F}} \mathit{Init}(N) : \tau \mid \mathcal{C}'_N \\ \mathcal{C}_N = \mathcal{C}'_N \wedge \tau \sim \sigma \end{array}\right]_{N \in \mathrm{dom}(\mathcal{N})} \qquad \left[\begin{array}{c} \cdot \vdash_{_}^{\mathcal{T};\mathcal{F}} \mathit{Init}(N) : \tau \mid \mathcal{C}'_N \\ \mathcal{C}_N = \mathcal{C}'_N \wedge \tau \sim \sigma \end{array}\right]_{(N,\sigma) \in \mathcal{I}}}{\mathit{Init} \Longrightarrow \bigwedge_{N \in \mathrm{dom}(\mathcal{N})} \mathcal{C}_N \wedge \bigwedge_{(N,\sigma) \in \mathcal{I}} \mathcal{C}_N}\ (\text{C-INIT})$$

Fig. 12 Validation rules for module definitions.

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

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

Fig. 13 Evaluation rules of the size parameter.

pression. The required memory is calculated backward from the rules of operational semantics. For example, in the case of the expression **`let`**, the free space in the local variable environment is required for e_1 to be evaluated, and for e_2 to be evaluated with x bound to the environment, which is expressed in the second output $s_1 \sqcup (1 + s_2)$.

In the case of a function call, we build a new size environment that binds the actual sizes of the arguments calculated using the

© 2021 Information Processing Society of Japan

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

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

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

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!] = (\tau_2, s_1 \sqcup (1 + s_2), t_1 + t_2, u_1 \sqcup u_2)$$
$$\text{where } (\tau_1, s_1, t_1, u_1) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_1]\!],\ (\tau_2, s_2, t_2, u_2) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;(\Gamma, x:\tau_1)}[\![e_2]\!]$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] = (\tau_2, s_1 \sqcup s_2 \sqcup s_3, t_1 + (t_2 \sqcup t_3), u_1 \sqcup u_2 \sqcup u_3)$$
$$\text{where } (\tau_i, s_i, t_i, u_i) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!]\ (1 \le i \le 3)$$

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

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![f(e_1, \ldots, e_n)]\!] = (\theta\tau,\ (n + s') \sqcup \textstyle\bigsqcup_{i \in 1 \ldots n}(n - 1 + s_i),\ t' + \sum_{i \in 1 \ldots n} t_i, (1 + u') \sqcup \bigsqcup_{i \in 1 \ldots n} u_i)$$
$$\text{where } (\tau_i, s_i, t_i, u_i) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_i]\!]\ (1 \le i \le n),$$
$$\mathcal{F}(f) = (x_1 : \pi_1, \ldots, x_n : \pi_n) : \tau\ \texttt{where}\ \{\overrightarrow{\mathcal{A}}\}\texttt{[}\ \overrightarrow{\delta}\ \texttt{]}\ \texttt{=}\ e,$$
$$\theta = [\pi_1, \ldots, \pi_n \mapsto \tau_1, \ldots, \tau_n],$$
$$(\tau', s', t', u') = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\{\pi_i\downarrow \mapsto ev_\Delta[\![\tau_i\downarrow]\!]\}_{i \in 1 \ldots n}; \{x_i : \pi_i\}_{i \in 1 \ldots n}}[\![e]\!]$$

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

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

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e\ \texttt{adj}\ \texttt{[}\psi\texttt{]}]\!] = (\rho^{\psi}, s, t, u) \quad \text{where } (\rho^{\psi'}, s, t, u) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e]\!]$$

$$\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![\texttt{fit}\ e_1\ \texttt{to}\ x : \rho^k \to e_2\ \texttt{|}\ \texttt{fail} \to e_3]\!] = (\tau_2, s_1 \sqcup (1 + s_2) \sqcup s_3, t_1 + (t_2 \sqcup t_3), u_1 \sqcup u_2 \sqcup u_3)$$
$$\text{where } (\tau_1, s_1, t_1, u_1) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_1]\!],\ (\tau_2, s_2, t_2, u_2) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma, x:\rho^k}[\![e_2]\!],\ (\tau_3, s_3, t_3, u_3) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_3]\!]$$

$$\boxed{\overline{\mathcal{M}}^{\mathcal{T};\mathcal{F}}_{m;\Delta;\Gamma}(\chi(x_1 : \pi_1, \ldots, x_n : \pi_n) \to e) = (s, t, u)}$$

$$\overline{\mathcal{M}}^{\mathcal{T};\mathcal{F}}_{m;\Delta;\Gamma}(\chi(x_1 : \pi_1, \ldots, x_n : \pi_n) \to e) =$$
$$\text{if } m < 1 + |I(\chi)| \text{ then } (0, 0, 0) \text{ else } (n + \textstyle\bigsqcup_{a \in \mathbf{A}} s_a, \bigsqcup_{a \in \mathbf{A}} t_a, \bigsqcup_{a \in \mathbf{A}} u_a)$$
$$\text{where } \chi(T^{\rho}_1, \ldots, T^{\rho}_n) \in \mathcal{T}(\rho),$$
$$\mathbf{A} = \{\ \{\pi_i \downarrow \mapsto p_i\}_{i \in I(\chi)}\ \mid ((\forall i \in 1 \ldots n, p_i > 0) \wedge m = 1 + \textstyle\sum_{i \in I(\chi)} p_i) \vee I(\chi) = \emptyset\},$$
$$b = \{\pi_j \downarrow \mapsto k_j \mid j \in 1 \ldots n, T^{\rho}_j = \rho'^{k_j}\},$$
$$(\tau_a, s_a, t_a, u_a) = \mathcal{M}^{\mathcal{T};\mathcal{F}}_{(\Delta, a, b);(\Gamma, x_1:\pi_1, \ldots, x_n:\pi_n)}[\![e]\!]\ (a \in \mathbf{A})$$

Fig. 14 Memory usage estimation algorithm.

auxiliary function *ev* to the size variables declared in the callee function. The environment is used in traversing the body of the callee function.

In $\overline{\mathcal{M}}$, the size m of the type to be decomposed in the **case** is also used as the input, and the branch is targeted instead of the expression e. The outputs are the free space in the local variable environment s, free space in the heap t, and free space in the call stack u needed when this branch is evaluated. The algorithm $\overline{\mathcal{M}}$ finds the memory required in the evaluation of the expression e of the branch in all cases in a set of size-parameter assignments $\mathbf{A}$ that satisfy the conditions of the constructor decomposition. If the conditions of the constructor decomposition cannot be satisfied, that is, if it is statically known that the expression of the branch will not be executed, s, t, and u are all set to 0. As the algorithm $\mathcal{M}$ for the **case** expression uses the maximum values of all of the results of the branches, if $\overline{\mathcal{M}}$ estimates resources as 0, the result is just ignored.

As the algorithm expands the expression while calculating the concrete values of the size parameters, the execution time varies depending on the size of the expression type. The algorithm searches for parameters in the constructor, so it is also affected by the structure of the type. In particular, the search for the branch requires the search for overlapping combinations of parameters in the constructors, which may cause a computational explosion. However, considering that Emfrp$^{\mathrm{BCT}}$ is a language for small-scale embedded systems, and assuming that the size of the code to be handled is relatively small and the type structure is not complex, this algorithm can still be practical.

We have demonstrated an algorithm for estimating the memory usage of an expression. As described in Section 4.3, to estimate the memory usage of the entire module, it is also necessary to consider the memory used by the node, the node's previous value, and the runtime.

© 2021 Information Processing Society of Japan

$$\boxed{H, \mathcal{T} \vdash l : \sigma}$$

$$\frac{H(l) = c^{\mathcal{B}}}{H, \mathcal{T} \vdash l : \mathcal{B}} \text{(H-CONST)} \qquad \frac{\begin{array}{l} H(l) = \chi[k](l_1, \ldots, l_n) \\ \chi(T_1^{\rho}, \ldots, T_n^{\rho}) \in \mathcal{T}(\rho) \\ k = 1 + \sum_{i \in I(\chi)} k_i \end{array} \quad \left[\begin{cases} H, \mathcal{T} \vdash l_i : \rho^{k_i} & (i \in I(\chi)) \\ H, \mathcal{T} \vdash l_i : \sigma_i \quad \wedge \quad \sigma_i \downarrow \leq T_i^{\rho} \downarrow & (\text{otherwise}) \end{cases} \right]_{i \in 1 \ldots n}}{H, \mathcal{T} \vdash l : \rho^k} \text{(H-CTOR)}$$

Fig. 15 Typing rules for values on the heap.

5.5 Soundness

We demonstrate the soundness of typing expressions with the operational semantics, the type system, and the memory usage estimation algorithm defined so far. First, we show the termination of the algorithm for estimating the amount of memory used. We define the consistency between the type environment Γ and the size environment Δ, and the model of the size constraint as follows.

Definition 3 (Consistency between the type environment and the size environment)**.**
When $\bigcup\{\mathrm{FV}(\Gamma(x)\downarrow) \mid x \in \mathrm{dom}(\Gamma)\} \subseteq \mathrm{dom}(\Delta)$ is satisfied, we say "type environment Γ and Δ are consistent," or (Γ, Δ)-consistent.

Definition 4 (Model of size constraint)**.**
We assume that $\mathrm{FV}(C) \subseteq \mathrm{dom}(\Delta)$ for the size environment Δ and the size constraint C. If the constraint that every free variable in C is replaced by Δ is valid, "Δ is a model of C."

Using these definitions, we show Theorem 5 for the algorithm to estimate the amount of memory used.

Theorem 5 (Termination of the memory usage estimation algorithm)**.**
Assume that:

- $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$;
- (Γ, Δ)-consistent;
- for any function name g in the expression e, $f \geq_{\mathcal{F}}^{*} g$ is satisfied;
- Δ is a model of C.

Then, the following hold:

- $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!]$ terminates in a finite number of steps;
- when $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!] = (\tau_{\mathcal{M}}, s, t, u)$, $\tau_{\mathcal{M}} = \tau$.

Proof Sketch. By induction on the lexicographic order of the triple $(f, ev_{\Delta}[\![\overrightarrow{\delta}]\!], e)$, where f is the function name, $\overrightarrow{\delta}$ is the measure function of f, and e is an expression. See Appendix A.1 for details. □

For the node update expression, Corollary 1 holds with the type environment for all the nodes on which the target node depends.

Corollary 1 (Termination of the memory usage estimation algorithm for node update expression)**.**
For any node N in $\mathcal{N}$, $\mathcal{M}_{\cdot;\Gamma_{\mathcal{N}}}^{\mathcal{T};\mathcal{F}}[\![\mathcal{N}(N)]\!]$ terminates in a finite number of steps, where $\Gamma_{\mathcal{N}} = \mathcal{I}, N_1 : \sigma_1, \ldots, N_n : \sigma_n$.

We show that the estimation algorithm terminates in a finite number of steps. In the following, we implicitly use the termination property of the algorithm.

Next, we define the consistency between types represented by the values in the heap and each environment. In **Fig. 15**, we show the typing rules for location l on the heap H. This expresses the value traced from l on the heap H would be typed. The definition of consistency for the heap and each environment is as follows.

Definition 6 (Consistency between node location environment, local variable environment, heap, type environment, and size environment)**.**
When:

- (Γ, Δ)-consistent;
- for any $x \in \mathrm{dom}(E)$, $H, \mathcal{T} \vdash E(x) : \sigma$ and $\sigma \downarrow \leq ev_{\Delta}[\![\Gamma(x)\downarrow]\!]$;
- for any $n \in \mathrm{dom}(\mathcal{L})$, $H, \mathcal{T} \vdash L(n) : \sigma'$ and $\sigma' \downarrow \leq ev_{\Delta}[\![\Gamma(\mathrm{node_name}(n))\downarrow]\!]$;

are satisfied, we say "node location environment $\mathcal{L}$, local variable environment E, heap H, type environment H and size environment Δ are consistent," or $(\mathcal{L}, E, H, \Gamma, \Delta)$-consistent, where $\mathrm{node_name}(n)$ denotes a name of n [*1].

Finally, we show soundness for the typing of expressions in Theorem 7.

Theorem 7 (Soundness of typing of expressions)**.**
Assume that:

- $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$;
- $(\mathcal{L}, E, H, \Gamma, \Delta)$-consistent;
- for any function name g in the expression e, $f \geq_{\mathcal{F}}^{*} g$ is satisfied;
- Δ is a model of C;
- $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!] = (\tau_{\mathcal{M}}, s_{\mathcal{M}}, t_{\mathcal{M}}, u_{\mathcal{M}})$.

For any $s \geq s_{\mathcal{M}}$, $t \geq t_{\mathcal{M}}$, and $u \geq u_{\mathcal{M}}$, there exist l, t', H', and σ such that the following are satisfied:

- $[s]E \mid [t]H \vdash_{\mathcal{L}}^{\mathcal{T};\mathcal{F}} e \Downarrow_u l; [t']H'$;
- $t' \geq t - t_{\mathcal{M}}$;
- $H', \mathcal{T} \vdash l : \sigma$;
- $\sigma \downarrow \leq ev_{\Delta}[\![\tau\downarrow]\!]$.

Proof Sketch. This proved by induction on the lexicographic ordering of the triple $(f, ev_{\Delta}[\![\overrightarrow{\delta}]\!], e)$, as used in the proof of Theorem 5. See Appendix A.2 for details. □

Noting that there are no local variables and that Δ, C are empty because there are no size variables at the beginning of a node update, we obtain Corollary 2 for typing of nodes.

Corollary 2 (Soundness of typing of node)**.**
Assume that:

- $\Gamma_{\mathcal{N}}(N) = \tau_N$;
- $(\mathcal{L}, \cdot, H, \Gamma_{\mathcal{N}}, \cdot)$-consistent;
- $\mathcal{M}_{\cdot;\Gamma_{\mathcal{N}}}^{\mathcal{T};\mathcal{F}}[\![\mathcal{N}(N)]\!] = (\tau_{\mathcal{M}}, s_{\mathcal{M}}, t_{\mathcal{M}}, u_{\mathcal{M}})$.

There exist l, t', H', and σ such that the following are satisfied:

- $[s_{\mathcal{M}}]\cdot \mid [t_{\mathcal{M}}]H \vdash_{\mathcal{L}}^{\mathcal{T};\mathcal{F}} \mathcal{N}(N) \Downarrow_{u_{\mathcal{M}}} l; [t']H'$;
- $H', \mathcal{T} \vdash l : \sigma$;
- $\sigma \downarrow \leq \tau_N \downarrow$.

Based on these theorems and corollaries, the operational semantics, type system, and memory usage estimation algorithm of Emfrp$^{\mathrm{BCT}}$ are sound. That is, for nodes that are typed correctly and whose size constraints are valid (satisfiable), node updates can be performed with the amount of memory obtained by the algorithm without running out of resources.

[*1] When n is a reference of the node, it returns n. When n is a previous-value reference, it returns the name of the referred node.

© 2021 Information Processing Society of Japan

6. Implementation and Evaluation

We implemented the Emfrp^{BCT} compiler. In this section, we explain the details of the implementation. Then, the results of various overhead measurements are discussed.

6.1 Emfrp^{BCT} Compiler

The implemented compiler translates Emfrp^{BCT} code into C code. The executable binary (program) is obtained by compiling the translated C code with a C compiler. The compiled program is executed according to the operational semantics shown in Section 5.2. To prevent unintended optimization by the C compiler, the program manages its own local variable allocation and stack pointers during function calls. It also follows Emfrp in the format of external input and output, and in the code that needs to be filled by users.

BCTs are compiled as variant type values in Emfrp with additional runtime size information. The language runtime includes some kind of mark-sweep GC to manage BCTs, which is an extension of the memory management for variant types in Emfrp. The memory usage estimation algorithm in the compiler also considers the memory space used by the GC.

There is a difference between the formal definition and the implementation of the runtime representation of the values of basic types. In the formal definition, the values of basic types are placed on the heap, similar to the values of BCTs, and are referenced via pointers from the variable stack. In the implementation, however, the values are copied and assigned directly to the variable stack.

To improve performance, the memory usage estimation algorithm is implemented to avoid redundant traversals with the same size parameter.

6.2 Evaluation Experiments

We performed evaluation experiments using the implemented compiler. The compiler was implemented using OCaml (version 4.11.1), with satisfiability modulo theories (SMT) solver Z3 (version 4.8.9.0) [3], [5] as an external library. The output C code was compiled by clang (version 12.0.0). In compiling the code, because the target system is a small-scale embedded system, the -Os option was set to optimize the program size. All measurements were performed on a MacBook Air (Intel Core i7 2.2 GHz CPU, 8 GB RAM, macOS Catalina 10.15.6).

6.2.1 Compilation Time

The execution time of the algorithm proposed in Section 5.4 depends on the size of BCTs in the code and the number of recursive types used inside the types. For `DupCheck` and `Top10Sum` (see Section 4.4), we measured the time required to compile the code by changing the size of the history nodes. The results are listed in **Tables 1** and **2**. Each program code uses `List[`$N+1$`]` type and `Heap[`$2N+1$`]` type.

Table 1 Comparison of the size of `List` and the compilation time in the `DupCheck` module.

N	Type check [sec]	Memory estimate [sec]	Total [sec]
4	0.015466	0.000049	0.015783
10	0.015334	0.000068	0.015680
20	0.015473	0.000128	0.015876
50	0.015486	0.000233	0.015993
100	0.015504	0.000530	0.016371
200	0.015121	0.000830	0.016227
300	0.015402	0.001312	0.016992
400	0.014178	0.001841	0.016312
500	0.015623	0.005209	0.021105

Table 2 Comparison of the size of `Heap` and the compilation time in the `Top10Sum` module.

N	Type check [sec]	Memory estimate [sec]	Total [sec]
10	0.018962	0.013779	0.033102
20	0.018160	0.192269	0.210782
30	0.019086	1.028855	1.048550
40	0.021169	3.467531	3.489383
50	0.016734	8.922994	8.940415
60	0.019698	19.372921	19.393433
70	0.028259	36.680560	36.710777
80	0.017129	64.181296	64.200058
90	0.017927	104.388627	104.408470
100	0.019529	163.357405	163.379870

`DupCheck` shows a slow change in memory usage estimation time. When decomposing the `List` type by the **`case`** expression, the size to be searched is uniquely determined, and the function expansion is performed for at most the length of the list. Therefore, it is expected that the computation time of the algorithm increases slowly as the size increases, and this tendency was confirmed in actual measurements.

`Top10Sum` takes a long time to estimate the memory usage even for relatively small sizes. This is because `Heap` has constructor `T` that contains two `Heap`. When decomposing a tree in the **`case`** expression, it is expected that the computation time will increase drastically because the size variables are assigned and searched exhaustively to satisfy the size constraint. This tendency was confirmed by actual measurements.

In both programs, the type checking time is almost constant regardless of N. The form of extracted constraints during type checking does not depend on the size of BCTs, so the satisfiability checking of the constraints is performed in a constant time.

6.2.2 Execution Time and Program Size

We measured the effect of using BCTs on program size and runtime overhead. We measured the program size (text, data, and block starting symbol (bss)) and execution time for 10^7 iterations of `DupCheck` and `Top10Sum` with random input for each of three programs: one converted from the code shown in Section 3 using the current Emfrp compiler [*2], one implemented using tuples with Emfrp^{BCT}, and one written using BCTs.

For `DupCheck`, we measured the modules whose history node types are `List[5]` ($N = 4$) and `List[31]` ($N = 30$). The results are listed in **Tables 3** and **4**, respectively. In both cases, the implementation using tuples was faster than that using lists. In the case of $N = 4$, there is no significant difference in the text area where the program code is stored. However, when $N = 30$, the text area of the program using tuples is larger. This is because constructor applications are inline expanded, and the search for duplicate elements is implemented as a series of **`if`** expressions. Meanwhile, when a list is used, the size of the text area remains almost the same as the number of histories changes, but the bss area increases significantly. This is because the number of function calls and the stack space used increase with the number of histories managed. In terms of whole program size, the example

[*2] https://github.com/sawaken/emfrp

© 2021 Information Processing Society of Japan

Table 3 Comparison of the execution time and the program size of `DupCheck` ($N = 4$).

$N = 4$	Exec [sec]	text [byte]	data [byte]	bss [byte]
Emfrp	0.358	1,374	56	240
Tuple	0.840	2,419	48	384
List	1.702	2,395	56	640

Table 4 Comparison of the execution time and the program size of `DupCheck` ($N = 30$).

$N = 30$	Exec [sec]	text [byte]	data [byte]	bss [byte]
Emfrp	1.467	6,312	64	1,280
Tuple	5.045	8,149	56	1,840
List	12.063	2,417	56	3,552

Table 5 Comparison of the execution time and the program size of `Top10Sum` ($N = 10$).

$N = 10$	Exec [sec]	text [byte]	data [byte]	bss [byte]
Emfrp	0.484	3,397	56	472
Insertion	2.339	5,547	48	720
Heap tree	2.090	3,582	56	3,024

Table 6 Comparison of the execution time and the program size of `Top10Sum` ($N = 30$).

$N = 30$	Exec [sec]	text [byte]	data [byte]	bss [byte]
Emfrp	1.409	11,585	56	1,272
Insertion	6.692	22,976	56	1,840
Heap tree	5.533	3,573	56	8,784

using lists saves more memory.

For `Top10Sum`, we measured the modules whose history node `h` has the types `Heap[21]` ($N = 10$) and `Heap[61]` ($N = 30$). The results are listed in **Tables 5** and **6**, respectively. In terms of execution time, the example using a heap tree runs faster even when the number of elements changes. The program size shows the same trend as that of `DupCheck`. When $N = 30$, the implementation using insertion sorting of tuples enlarges the text area. This is because the constructor is applied to every element number in order so that the value is inserted at the appropriate position in the tuple.

A common point among the measurement results of both the `DupCheck` and `Top10Sum` modules is that the existing Emfrp program is fast and relatively memory-saving. In the existing Emfrp, variables and function calls on Emfrp are transformed to correspond to those in the C language. Therefore, it is easy to optimize local variables by assigning them to registers, and the computation is fast. Meanwhile, the current Emfrp^{BCT} compiler has the overhead of managing its own stack to adapt its behavior to its formal semantics. In addition, the existing Emfrp compiler analyzes the live information of the nodes and releases the memory occupied by the nodes at an early stage, so that the computation can be performed with less memory.

6.2.3 Execution Time per Iteration Phase

Tables 7 and **8** list the results of measuring the execution time of the program for each phase in the iteration. This is a cumulative measurement of the time spent on node update (update), mark phase for GC (mark), sweep phase or memory refresh (refresh), and total program execution time (total) for each implementation of examples at $N = 30$. The unit of time is seconds. The time measurement was performed by inserting the measurement C code (using the `clock` function) into the iteration loop. Owing to the overhead caused by the measurement code, the overall time

Table 7 Execution time of each phase in the program `DupCheck` ($N = 30$).

$N = 30$	update [sec]	mark [sec]	refresh [sec]	total [sec]
Tuple	14.174	5.293	5.980	45.962
List	19.612	7.087	5.959	53.227

Table 8 Execution time of each phase in the program `Top10Sum` ($N = 30$).

$N = 30$	update [sec]	mark [sec]	refresh [sec]	total [sec]
Insertion	15.850	5.300	5.975	47.651
Heap tree	11.376	7.743	6.600	46.210

increased compared with the results in Tables 4 and 6.

For the `DupCheck` module, the time to refresh the memory was approximately the same because the number of history nodes allocated in the heap area was approximately the same for the list and tuple cases. However, we could measure the overhead of marking the list, and the node update took longer to execute when using a list. By comparing the programs, the tuple implementation requires only one pattern match to update the history, whereas the list implementation requires a number of pattern matches in proportion to the length of the history. Therefore, from the results, it can be observed that the node update time increases in the list implementation compared with the tuple implementation because of the increased number of function calls and pattern matches.

For the `Top10Sum` module, the number of objects allocated in the heap area is N in the case of the insertion sort implementation using tuples, but $2N + 1$ in the heap tree version. The heap tree implementation requires more mark and refresh time for the objects on the heap than the insertion sort implementation. Comparing the two implementations, the insertion sort implementation requires pattern matching proportional to the history length, whereas the heap tree version requires it to be proportional to the height of the tree. As the height of the tree is smaller than the history length in many cases, the node update time is shorter in the heap tree implementation.

6.3 Discussion

From the measurements, it can be observed that the code size of the program using BCT does not increase because the repetition of the process is expressed as a recursive function and that there is some overhead in marking the objects of BCTs. We cannot make a simple comparison of the node update time because the calculation method (algorithm) is different in each case. In the case of the `DupCheck` module, the implementation using lists is a program that is robust to the changes in the number of histories, but the access to the elements is done by using a recursive function and pattern matching, which is more time-consuming than the tuple implementation. In the case of the `Top10Sum` module, we used a heap tree, which was difficult to represent in the existing Emfrp; thus, we were able to write a more time-efficient process. To improve the responsiveness of reactive systems, it is useful to use more efficient data structures to improve the performance.

Compared with tuple-based naïve data management, BCT-based programs are expected to use many recursive function calls and pattern matching. To improve the performance, it is necessary to accelerate the function calls and pattern matching. In addition, because the upper bound of recursive function calls can be

© 2021 Information Processing Society of Japan

statically determined using the size parameter, inline expansion is expected to enhance the performance. However, inline expansion can lead to elaborate program code, so it should be optimized for each environment on a case-by-case basis.

7. Related Work

7.1 Sized Types

The proposed method, BCTs, is strongly influenced by Sized Types [21]. In Sized Types, in addition to the maximum object size similar to BCTs, a minimum size is included in the type. Using the Sized Type system, we can guarantee that streams always produce values and other properties, allowing the safe use of functional languages in embedded systems.

The most significant difference between Sized Types and our proposed method is that BCTs can handle the size information at runtime using a **fit** expression. In Emfrp, a node often refers to its own previous value by `@last`. When the node type is BCT, the current and previous values have the same type, so it is necessary to match the type and size well when defining nodes. In Emfrp^{BCT}, the size can be cast in the direction of size reduction using the size information at runtime in the **fit** expression, so that processes such as adding elements can be described concisely. In a Sized Type system, such casts require dummy data as elements so that the size is always the same, resulting in unnatural handling of the data.

Simplified code fragments of the `h` node in the `Top10Sum` module are listed in the following as an example:

```
1 node h : Heap[21] init (E adj[21]) =
2   fit h@last to
3   | hl:Tree[19] -> insert(input, hl)
4   | fail        -> insert(input, delMin(h@last))
```

If we simply introduce Sized Types to Emfrp, that is, if we describe node `h` without using **fit** expression, there are several possible ways to describe it. The first is to set a dummy data structure in advance, as follows, and always reduce the size by `delMin`:

```
1 node h : Heap[21] init (DUMMY_HEAP) =
2   insert(input, delMin(h@last))
```

This method requires `DUMMY_HEAP` to be set with the correct structure and size and, in some cases, it is difficult to assign it as an initial value. In addition, for some applications, it may not be appropriate to provide dummy data of type `Int`; therefore, an `Int Option` type may be used, which may increase the complexity of `delMin`.

The second method uses the function `size` to calculate the size of the heap tree at runtime and branch the process with **if** expression:

```
1 node h : Tree[21] init (E adj[21]) =
2   if size(t@last) <= 19 then
3     insert(input, CAST_SIZE(h@last))
4   else insert(input, delMin(h@last))
```

When there is enough space in the heap tree, it is necessary to match the type (size) of the arguments to call the `insert` function on `h@last`. In this code fragment, the `CAST_SIZE` function represents the size parameter decrementing operation while preserving the structure, but it is not possible to write such an operation as a function of existing Sized Types. This can be represented as an expression by repeating decomposition using **case** expressions until the required size is obtained. By decomposing the value of BCTs using the **case** expression, the size parameter can be reduced. Therefore, for a simple data structure such as a list, a list with the required size parameter can be obtained by nesting the **case** expression as many times as necessary. However, when the tree structure is decomposed to obtain the required size, as in the case of node `h`, a complex expression that considers all possible forms of the tree structure is required instead of a simple nesting of **case** expressions. Although expressions can be generated using macros, for example, the large number of branches given by the **case** expression duplicates the code for both cases where the required size is obtained and where it is not obtained, resulting in a bloated program. In this study, which targets small-scale embedded systems, this bloating of the program by code duplication must be avoided.

Therefore, the introduction of **fit** expressions is useful for adapting to the programming style using `@last` in Emfrp, and the actual size information is stored in the runtime representation of BCTs for use in **fit** expressions.

Combining Sized Types and region-based memory management [29], Hughes et al. proposed a method to determine the amount of resources consumed at runtime [20]. By allocating regions with their size information and considering the number of stores on the stack or heap as effects, they showed that a type-checked program can be computed in a fixed memory area. In their language, when an object is allocated to the heap, it is always indicated explicitly in which region it will be allocated, so there is no need to search for an upper bound on resource usage in **case** expression, as in this study. However, it is necessary for the user to specify the proper size of the region in advance.

7.2 Data Representation by Arrays

In procedural languages, data structures whose number of elements change at runtime are often represented using arrays. Data structures such as trees can be represented by treating the index number of the array as a pointer, but the array needs to be mutable for such usage to be memory efficient. We did not introduce arrays in our extension to preserve Emfrp as a pure FRP language. If we were to introduce arrays, we expected some restrictions to be introduced that are difficult to use, such as limiting to a constant of loop iterations to guarantee termination of node updates. This led to the introduction of recursive data types with size information and primitive recursive functions.

Futhark [7], [16], a functional language for graphics processing units (GPUs), is an attempt to handle arrays in a memory-efficient manner on a pure functional language. In Futhark, arrays are typed by linear types, which allows for in-place updates. Applying this feature to matrices allows us to write efficient programs for GPUs in functional languages. Recently, a new mechanism called Size Types [14], [15] was introduced to describe the type of a function with matrix size information, which can statically determine the size of some objects. Although its purpose is different from that of BCTs in this study, a similar notation was used.

© 2021 Information Processing Society of Japan

7.3 FRP Languages and Dataflow Languages

Hume [10] is a DSL for real-time embedded systems, which consists of event-driven state machines called Boxes and descriptions of their cooperative behavior. Hume has several levels for each language feature supported, and Emfrp corresponds to the level called FSM-Hume. Emfrp^{BCT} corresponds to the level of PR-Hume and supports primitive recursive functions and recursive data structures. FSM-Hume and PR-Hume have a cost model [11], which is defined by an operational semantics with the remaining capacity and remaining time of the resource area [9], [30], similar to Emfrp^{BCT}. To analyze the amount of resources, the Sized Types approach and Automatic Amortized Resource Analysis (AARA) approach (described in the following) have been proposed [22]. Unlike Emfrp^{BCT}, Hume does not comprise the actual size of the data structure in the language, so it is not possible to directly express `fit` to check if the size is within the upper limit. Therefore, this approach needs to be handled differently, such as by using a function to measure the size of the data structure. Hume combines many static analyses to analyze the amount of resources; however, it is difficult to accurately determine that a function *represents the size of the data structure* by static analysis, and it is also difficult to accurately inherit the size condition in the analysis for each branch.

Juniper [13] is an FRP language for small-scale embedded systems designed especially for Arduino, a board using a microcontroller (ATmega328). In addition to the usual type parameters, a capacity variable can be specified in Juniper's mechanism, which is equivalent to C++ templates [12]. Using this capacity variable, it is possible to specify the size of the arrays within the records (structures in C++). Juniper can also define recursive data types, but it does not provide a mechanism to specify the data size statically for them.

The FRP language proposed by Krishnaswami et al. [23] is another approach to resource management. In their language, special values representing resources are prepared before the start of the program, and these values are used to perform calculations with a fixed amount of memory. If the programmer manages the resources (the special values) properly, it can handle list structures. In addition, linearly typed resource values can handle higher-order functions and higher-order time-varying values without space leaks, enabling expressive descriptions of FRP.

Lustre [8], [24] is a synchronous dataflow language for real-time reactive systems. This language has many points in common with Emfrp, such as describing processes using combinations of time-varying values and features to obtain the value one step earlier (`@last` in Emfrp, `pre` in Lustre). The worst-case computation time and memory usage can be estimated by prohibiting loops such as recursive calls and the use of recursive data types, as in Emfrp. Lustre provides arrays and special instructions (`map`, `reduce`, etc.) for manipulating arrays to perform similar actions on multiple data. In Lustre, the example of `DupCheck` can be written concisely using these instructions, but it is not suitable for programs dealing with tree structures. In contrast, in Emfrp^{BCT}, by defining a list type using BCTs, we can use the same features as arrays in Lustre.

7.4 Other Resource Estimation Methods

Dependent types and Indexed Types [32] can include size information of objects in types. Indexed Types is a type system that allows size annotations and constraints to be specified for algebraic data types. The difference between Indexed Types and our BCTs is that flexible constraints can be described in type definitions, but in our research, we focus on the amount of resources required at runtime rather than consistency regarding size, so we let the user describe how much resource a type occupies at runtime. Dependent type systems allow the description of types that depend on values and are known to correspond to predicate logic. Using dependent types, it is possible to define and use types with size information, but these types require proofs about the size information as well as program code, which is a burden on the users. To reduce the burden, it is possible to solve the constraints obtained by dependent types automatically using SMT solvers [26], [31]. When taking such an approach, it is important to choose the class of constraints. Otherwise, type checking will be undecidable. In our method, constraints to be solved are in the category of Presburger arithmetic, so that the type checking is decidable. As the type checking is decidable, when the solution of a constraint is not found (type checking fails), the constraint is unsatisfiable, and an error can be reported properly.

AARA [18] is a statistical analysis method for program resource usage introduced by Hofmann et al. AARA statically estimates the amount of required resources by a type system with numerical values called a potential and a linear programming solver, rather than using object size information, as in our study or Sized Types. Hoffmann et al. recently applied this method to OCaml [17]. AARA assigns potentials to types, which makes it difficult for users to intuitively set the amount of resources. Therefore, in our study, we adopted the approach of assigning direct size information of the data structure to types. The proposed method traverses the syntax tree exhaustively to determine the resource usage, depending on the size and type structure, which may take a long time to analyze. As AARA took a different approach, solving the constraints obtained by typing using linear programming to estimate resource usage, it is expected to accelerate our algorithm by integrating this method.

8. Conclusion

In this study we proposed a new FRP language, Emfrp^{BCT}, which employs a type system called BCTs that statically specify the *maximum size of constructible structures*. The language is an extension of the existing FRP language Emfrp, designed for small-scale embedded systems. With the BCTs, programs that required redundant descriptions in Emfrp can now be written concisely, and data management can be performed efficiently. Despite the availability of recursive data types, Emfrp^{BCT} retains the properties of Emfrp, such as the termination of node updates and the ability to statically estimate the amount of memory used at runtime. We proved these properties by formalizing the operational semantics, type system, and memory usage estimation algorithm of Emfrp^{BCT} and demonstrated their soundness. Furthermore, we implemented a compiler for Emfrp^{BCT} and measured the overhead of compilation and execution times.

© 2021 Information Processing Society of Japan

For future work, we plan to enhance Emfrp^{BCT} by introducing: (1) type polymorphism and size polymorphism, (2) mutually recursive definitions of functions and types, (3) flexible measure functions other than size for defining recursive functions, and (4) a code optimizer for Emfrp^{BCT} that incorporates existing optimization techniques. Presently, the size parameters are constants. By statically parameterizing them on a per-module basis, we can increase the reusability of the modules. The current algorithm traverses the syntax tree of a program with size information, which increases the time required for estimation when the program size is large or when dealing with complex BCTs. If a fast but inaccurate algorithm is available, its use during development will help reduce the development time. To improve the quality of the final product, the (accurate but time-consuming) algorithm proposed in this work can then be used for deployment in the production environment. The development of different memory usage estimation algorithms is also considered a future challenge.

Acknowledgments The authors wish to thank the anonymous reviewers for their helpful suggestions and comments. This work was supported in part by JSPS KAKENHI Grant Numbers 18K11236 and 19K20245.

References

[1] Apfelmus, H.: Reactive Banana, available from ⟨https://wiki.haskell.org/Reactive-banana⟩ (2016).

[2] Bainomugisha, E., Carreton, A.L., Van Cutsem, T., Mostinckx, S. and De Meuter, W.: A Survey on Reactive Programming, *ACM Computing Surveys*, Vol.45, No.4, pp.52:1–52:34 (online), DOI: 10.1145/2501654.2501666 (2013).

[3] Bjørner, N. and Janota, M.: Playing with Quantified Satisfaction, *LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations*, Fehnker, A., McIver, A., Sutcliffe, G. and Voronkov, A. (Eds.), EPiC Series in Computing, Vol.35, pp.15–27, EasyChair (online), DOI: 10.29007/vv21 (2015).

[4] Czaplicki, E. and Chong, S.: Synchronous Functional Reactive Programming for GUIs, *34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2013*), pp.411–422, ACM (online), DOI: 10.1145/2499370.2462161 (2013).

[5] De Moura, L. and Bjørner, N.: Z3: An Efficient SMT Solver, *Proc. Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp.337–340, Springer-Verlag (2008).

[6] Elliott, C. and Hudak, P.: Functional Reactive Animation, *2nd ACM SIGPLAN International Conference on Functional Programming* (*ICFP 1997*), pp.263–273, ACM (online), DOI: 10.1145/258949.258973 (1997).

[7] The Futhark Programming Language: High-performance purely functional data-parallel array programming on the GPU (2020), available from ⟨https://futhark-lang.org/⟩.

[8] Halbwachs, N., Caspi, P., Raymond, P. and Pilaud, D.: The synchronous data flow programming language LUSTRE, *Proc. IEEE*, Vol.79, No.9, pp.1305–1320 (1991).

[9] Hammond, K., Ferdinand, C. and Heckmann, R.: Towards Formally Verifiable Resource Bounds for Real-Time Embedded Systems, *SIGBED Rev.*, Vol.3, No.4, pp.27–36 (online), DOI: 10.1145/1183088.1183093 (2006).

[10] Hammond, K. and Michaelson, G.: Hume: A Domain-Specific Language for Real-Time Embedded Systems, *2nd International Conference on Generative Programming and Component Engineering* (*GPCE 2003*), Lecture Notes in Computer Science, Vol.2830, pp.37–56, Springer-Verlag (online), DOI: 10.1007/978-3-540-39815-8_3 (2003).

[11] Hammond, K. and Michaelson, G.: Predictable Space Behaviour in FSM-Hume, *Implementation of Functional Languages* (*IFL 2002*), Lecture Notes in Computer Science, Vol.2670, pp.1–16, Springer (2003).

[12] Helbling, C.: Juniper Language Documentation (ver. 2.2.0) (2016), available from ⟨http://www.juniper-lang.org/language_docs.html⟩.

[13] Helbling, C. and Guyer, S.Z.: Juniper: A Functional Reactive Programming Language for the Arduino, *4th International Workshop on Functional Art, Music, Modelling, and Design* (*FARM 2016*), pp.8–16, ACM (online), DOI: 10.1145/2975980.2975982 (2016).

[14] Henriksen, T.: Towards Size Types in Futhark (2019), available from ⟨https://futhark-lang.org/blog/2019-08-03-towards-size-types.html⟩.

[15] Henriksen, T.: Futhark 0.15.1 released - now with size types! (2020), available from ⟨https://futhark-lang.org/blog/2020-03-15-futhark-0.15.1-released.html⟩.

[16] Henriksen, T., Serup, N.G.W., Elsman, M., Henglein, F. and Oancea, C.E.: Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates, *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.556–571, ACM (online), DOI: 10.1145/3062341.3062354 (2017).

[17] Hoffmann, J., Das, A. and Weng, S.-C.: Towards Automatic Resource Bound Analysis for OCaml, *SIGPLAN Not.*, Vol.52, No.1, pp.359–373 (online), DOI: 10.1145/3093333.3009842 (2017).

[18] Hofmann, M. and Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs, *SIGPLAN Not.*, Vol.38, No.1, pp.185–197 (online), DOI: 10.1145/640128.604148 (2003).

[19] Hudak, P., Courtney, A., Nilsson, H. and Peterson, J.: Arrows, Robots, and Functional Reactive Programming, *Advanced Functional Programming*, Lecture Notes in Computer Science, Vol.2638, pp.159–187, Springer-Verlag (online), DOI: 10.1007/978-3-540-44833-4_6 (2003).

[20] Hughes, J. and Pareto, L.: Recursion and Dynamic Data-Structures in Bounded Space: Towards Embedded ML Programming, *SIGPLAN Not.*, Vol.34, No.9, pp.70–81 (online), DOI: 10.1145/317765.317785 (1999).

[21] Hughes, J., Pareto, L. and Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types, *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL '96*), pp.410–423, ACM (online), DOI: 10.1145/237721.240882 (1996).

[22] Jost, S., Loidl, H.-W., Hammond, K., Scaife, N. and Hofmann, M.: "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis, *FM 2009: Formal Methods*, Cavalcanti, A. and Dams, D.R. (Eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp.354–369 (2009).

[23] Krishnaswami, N.R., Benton, N. and Hoffmann, J.: Higher-Order Functional Reactive Programming in Bounded Space, *SIGPLAN Not.*, Vol.47, No.1, pp.45–58 (online), DOI: 10.1145/2103621.2103665 (2012).

[24] Verimag Lustre V6, available from ⟨http://www-verimag.imag.fr/Lustre-V6.html⟩ (2020).

[25] Okasaki, C.: *Purely Functional Data Structures*, Cambridge University Press (1999).

[26] Rondon, P.M., Kawaguchi, M. and Jhala, R.: Liquid Types, *Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08*, pp.159–169, ACM (online), DOI: 10.1145/1375581.1375602 (2008).

[27] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, *MODULARITY Companion 2016: Companion Proc. 15th International Conference on Modularity*, pp.36–44, ACM (online), DOI: 10.1145/2892664.2892670 (2016).

[28] SodiumFRP, available from ⟨https://github.com/SodiumFRP⟩.

[29] Tofte, M. and Talpin, J.-P.: Region-Based Memory Management, *Inf. Comput.*, Vol.132, No.2, pp.109–176 (online), DOI: 10.1006/inco.1996.2613 (1997).

[30] Vasconcelos, P.B. and Hammond, K.: Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs, *Proc. 15th International Conference on Implementation of Functional Languages, IFL'03*, pp.86–101, Springer-Verlag (online), DOI: 10.1007/978-3-540-27861-0_6 (2003).

[31] Xi, H. and Pfenning, F.: Eliminating Array Bound Checking through Dependent Types, *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI '98*, pp.249–257, ACM (online), DOI: 10.1145/277650.277732 (1998).

[32] Zenger, C.: Indexed types, *Theoretical Computer Science*, Vol.187, No.1, pp.147–165 (online), DOI: 10.1016/S0304-3975(97)00062-5 (1997).

Appendix

A.1 Proof of the Termination of the Memory Usage Estimation Algorithm

Theorem 5 (Termination of the memory usage estimation algorithm)**.**

© 2021 Information Processing Society of Japan

Assume that:
- $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$;
- (Γ, Δ)-consistent;
- for any function name g in the expression e, $f \geq_{\mathcal{F}}^* g$ is satisfied;
- Δ is a model of C.

Then, the following hold:
- $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!]$ terminates in a finite number of steps;
- when $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!] = (\tau_{\mathcal{M}}, s, t, u)$, $\tau_{\mathcal{M}} = \tau$.

Proof. Define the triple $(f, ev_\Delta[\![\overrightarrow{\delta}]\!], e)$. Here f is a function name, and the order of the functions is defined as $\geq_{\mathcal{F}}^* \cup \{(-, f) \mid f \in \mathrm{dom}(\mathcal{F})\}$. Here, $-$ is a placeholder to check the node update expression. Next, $\overrightarrow{\delta}$ is the measurement function of f. Finally, for expression e, subexpressions of e are ordered as less than e.

The proof of termination proceeds by induction of the lexicographic ordering of the triple. The case is divided according to the rule applied at the end of the type derivation ($\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$).

If the last applied typing rule is T-CONST, T-VAR, T-NODE, or T-ATLAST, then the statement clearly holds.

In the case of T-CTOR, T-LET, T-IF, T-OP, T-ADJ, or T-FIT, the third element of the triple decreases in each assumption of the rules, and the statement follows immediately from the induction hypothesis.

In the case of T-CASE, let e be the entire **case** expression and let e_i be the expression in each branch. For the expression to be decomposed in e, the third element of the triple is decreasing, so the induction hypothesis is applicable to the expression. Each branch is processed by $\overline{\mathcal{M}}$. Here $\overline{\mathcal{M}}$ generates a set of size-variable assignments $\mathbf{A}$ for the size variables contained in the constructor. The union of Δ and the assignment in $\mathbf{A}$ is a model of the constraint extracted during type checking of the branch expression e_i. As $\mathbf{A}$ is a finite set, $\mathcal{M}$ is called a finite number of times inside $\overline{\mathcal{M}}$ for the expression e_i of the branch. Here e_i is a subexpression of e, so the third element of the triple is reduced. From the above, the induction hypothesis is applicable to e_i. Hence, the execution of $\overline{\mathcal{M}}$ also terminates in a finite number of steps. In addition, the type of the entire e obtained by typing and the type obtained as a result of $\mathcal{M}(e)$ are identical. Therefore, the statement holds for the expression to be checked in e, as well as for e as a whole because the execution of each branch also terminates.

In the case of T-CALL, the expression is $g(e_1, \ldots, e_n)$. For the actual arguments $e_1, \ldots, e_n$ of the function call, each one is a subexpression of $g(e_1, \ldots, e_n)$, so the third element of the triple is reduced, and the induction hypothesis is applicable to these arguments. From this, the type of the expression is identical for the typing rule and $\mathcal{M}$. In the case of $g \neq f$, because $f \geq_{\mathcal{F}}^* g$ by the assumption of the statement, the first element of the triple is reduced in the call to $\mathcal{M}$ for the body of g. Therefore, the induction hypothesis is applicable for the body of g. If $g = f$, then this function call is a recursive call. Here, we show that the second element of the triple, the measure of the recursive call, decreases. Based on this assumption, Δ is a model of $\bigwedge_{i \in 1 \ldots n} C_i \wedge \bigwedge_{i \in 1 \ldots m} (\theta \mathcal{A}_i) \wedge \mathcal{R}$. Thus, it is also a model of $\mathcal{R}$. Therefore, from the constraint $\mathcal{R}$, $ev_\Delta[\![\Sigma_{\delta \in \overrightarrow{\delta}} (\theta\delta)]\!] < ev_\Delta[\![\overrightarrow{\delta}]\!]$. From the definitions of θ and $\overrightarrow{\delta}$, it follows that $\theta\delta = \tau_i \downarrow$ for some i. The value of the measure in $\mathcal{M}$ for the body of g is $ev_{\{\pi_i \downarrow \mapsto ev_\Delta[\![\tau_i \downarrow]\!]\}_{i \in 1 \ldots n}}[\![\overrightarrow{\delta}]\!]$:

$$
\begin{aligned}
& ev_{\{\pi_i \downarrow \mapsto ev_\Delta[\![\tau_i \downarrow]\!]\}_{i \in 1 \ldots n}}[\![\overrightarrow{\delta}]\!] \\
&= ev_{\{\pi_i \downarrow \mapsto ev_\Delta[\![\tau_i \downarrow]\!]\}_{i \in 1 \ldots n}}[\![\Sigma_{\pi_i \downarrow \in \overrightarrow{\delta}} \pi_i \downarrow]\!] \\
&= \Sigma_{\pi_i \downarrow \in \overrightarrow{\delta}} (ev_{\{\pi_i \downarrow \mapsto ev_\Delta[\![\tau_i \downarrow]\!]\}_{i \in 1 \ldots n}}[\![\pi_i \downarrow]\!]) \\
&= \Sigma_{\pi_i \downarrow \in \overrightarrow{\delta}} (ev_{\{\pi_i \downarrow \mapsto ev_\Delta[\![\tau_i \downarrow]\!]\}_{i \in 1 \ldots n}}[\![ev_\Delta[\![\tau_i \downarrow]\!]]\!]) \\
&= \Sigma_{\pi_i \downarrow \in \overrightarrow{\delta}} (ev_\Delta[\![\tau_i \downarrow]\!]) \\
&= \Sigma_{\delta \in \overrightarrow{\delta}} (ev_\Delta[\![\theta\delta]\!]) \\
&= ev_\Delta[\![\Sigma_{\delta \in \overrightarrow{\delta}} (\theta\delta)]\!] \\
&< ev_\Delta[\![\overrightarrow{\delta}]\!]
\end{aligned}
$$

With the above transformation, the measure in $\mathcal{M}$ for the function body is reduced compared with the measure in the original expression. As the second element of the triple is reduced, the induction hypothesis is applicable to the function body. In both cases, the execution of $\mathcal{M}$ for the function body terminates. Therefore, the statement holds. □

A.2 Proof of Soundness of Typing

Theorem 7 (Soundness of typing of expression)**.**
Assume that:
- $\Gamma \vdash_f^{\mathcal{T};\mathcal{F}} e : \tau \mid C$;
- $(\mathcal{L}, E, H, \Gamma, \Delta)$-consistent;
- for any function name g in the expression e, $f \geq_{\mathcal{F}}^* g$ is satisfied;
- Δ is a model of C;
- $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!] = (\tau_{\mathcal{M}}, s_{\mathcal{M}}, t_{\mathcal{M}}, u_{\mathcal{M}})$.

For any $s \geq s_{\mathcal{M}}$, $t \geq t_{\mathcal{M}}$, and $u \geq u_{\mathcal{M}}$, there exist l, t', H', and σ such that the following are satisfied:
- $[s]E \mid [t]H \vdash_{\mathcal{L}}^{\mathcal{T};\mathcal{F}} e \Downarrow_u l; [t']H'$;
- $t' \geq t - t_{\mathcal{M}}$;
- $H', \mathcal{T} \vdash l : \sigma$;
- $\sigma \downarrow \leq ev_\Delta[\![\tau \downarrow]\!]$.

Proof. The proof is given by induction on the triple lexicographic ordering used in the proof of Theorem 5. The cases are divided according to the rule applied at the end of the type derivation.

If the last rule applied is T-CONST, we can set $H' = H$, $l \mapsto c^{\mathcal{B}}$, $t' = t - 1$, and $\sigma = \mathcal{B}$.

For T-VAR, T-NODE, and T-ATLAST, $H' = H$, $t' = t$, and σ should be the corresponding types.

For T-CTOR, let $H = H_0$, $t = t_0$, and $e = \chi(e_1, \ldots, e_n)$. For $1 \leq i \leq n$, let $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e]\!] = (\tau_{\mathcal{M}i}, s_{\mathcal{M}i}, t_{\mathcal{M}i}, u_{\mathcal{M}i})$. From the definition of $\mathcal{M}$, for $1 \leq i \leq n$, $s \geq s_{\mathcal{M}} = 1 + \bigsqcup s_{\mathcal{M}i} > s_{\mathcal{M}i}$, $u \geq u_{\mathcal{M}} \geq u_{\mathcal{M}i}$ holds. From $t_0 \geq t_{\mathcal{M}} = 1 + \Sigma_{i \in 1 \ldots n} t_{\mathcal{M}i} \geq t_{\mathcal{M}1}$ and the induction hypothesis, there exist t_1, l_1, H_1, and σ_1 such that $[s-1]E \mid [t_0]H_0 \vdash_{\mathcal{L}}^{\mathcal{T};\mathcal{F}} e_1 \Downarrow_u l_1; [t_1]H_1$, $t_1 \geq t_0 - t_{\mathcal{M}1} \geq 1 + \Sigma_{i \in 2 \ldots n} t_{\mathcal{M}i}$, $H_1, \mathcal{T} \vdash l_1 : \sigma_1$, and $\sigma_1 \downarrow = ev_\Delta[\![\tau_1 \downarrow]\!]$ are satisfied. Repeating this for the arguments of the constructor, we obtain $t_n \geq t_0 - \Sigma_{i \in 1 \ldots n} t_{\mathcal{M}i} \geq 1$ and H_n. Applying the rule of operational semantics E-CTOR, we obtain $t' = t_n - 1 \geq t_0 - (1 + \Sigma_{i \in 1 \ldots n} t_{\mathcal{M}i}) = t - t_{\mathcal{M}}$ and $H' = H_n, l \mapsto \chi[k](l_1, \ldots, l_n)$.

For T-LET, the form of the expression is `let` $x = e_1$ `in` $e2$. Let $\mathcal{M}_{\Delta;\Gamma}^{\mathcal{T};\mathcal{F}}[\![e_1]\!] = (\tau_{\mathcal{M}1}, s_{\mathcal{M}1}, t_{\mathcal{M}1}, u_{\mathcal{M}1})$ and $\mathcal{M}_{\Delta;\Gamma,x:\tau_1}^{\mathcal{T};\mathcal{F}}[\![e_2]\!] =$

© 2021 Information Processing Society of Japan

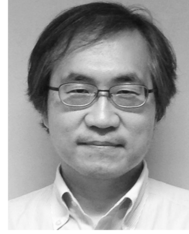
$(\tau_{\mathcal{M}2}, s_{\mathcal{M}2}, t_{\mathcal{M}2}, u_{\mathcal{M}2})$. Then, $s_{\mathcal{M}} \geq s_{\mathcal{M}1}, u_{\mathcal{M}} \geq u_{\mathcal{M}1}$, $s_{\mathcal{M}} \geq 1 + s_{\mathcal{M}2}, u_{\mathcal{M}} \geq u_{\mathcal{M}2}$, and $t_{\mathcal{M}} = t_{\mathcal{M}1} + t_{\mathcal{M}2}$. Let $s \geq s_{\mathcal{M}}$, $t \geq t_{\mathcal{M}}$, and $u \geq u_{\mathcal{M}}$. From the induction hypothesis with e_1, we obtain l_1, t_1, H_1, and σ_1 such that $[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_1 \Downarrow_u l_1; [t_1]H_1, t_1 \geq t - t_{\mathcal{M}1}$, $H_1, \mathcal{T} \vdash l_1 : \sigma_1$, and $\sigma_1 = ev_\Delta[\![\tau_1 \downarrow]\!]$ are satisfied. In addition, because $H \subseteq H_1$, $(\mathcal{L}, (E, x \mapsto l_1), H_1, (\Gamma, x : \tau_1), \Delta)$-consistent holds. Thus, from the induction hypothesis with e_2, we obtain l_2, t_2, H_2, and σ_2 such that $[s-1](E, x \mapsto l_1) \mid [t_1]H_1 \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_2 \Downarrow_u l_2; [t_2]H_2, t_2 \geq t_1 - t_{\mathcal{M}2}$, $H_2, \mathcal{T} \vdash l_2 : \sigma_2$, and $\sigma_2 = ev_\Delta[\![\tau_2 \downarrow]\!]$ are satisfied. Thus, we have $t' = t_2 \geq t - (t_{\mathcal{M}1} + t_{\mathcal{M}2})$, $l' = l_2$, $H' = H_2$, and $\sigma = \sigma_2$.

For T-IF, T-OP, and T-ADJ, the proofs proceed in the same way as that for T-CTOR. The case of T-FIT is the same as T-LET. In the case of T-CALL, the actual arguments are treated in the same manner as T-CTOR. The consistency of the size variables for the arguments is treated as in T-LET. Finally, the reduction of the triple for the body of the function is shown as the function call in Theorem 5. From these results, the statement of the entire function call expression is shown.

In the case of T-CASE, let the decomposed expression be e_0. Let $\mathcal{M}^{\mathcal{T};\mathcal{F}}_{\Delta;\Gamma}[\![e_0]\!] = (\tau_{\mathcal{M}0}, s_{\mathcal{M}0}, t_{\mathcal{M}0}, u_{\mathcal{M}0})$. Let the branches be $branch_i$ for $1 \leq i \leq n$ and $\overline{\mathcal{M}}^{\mathcal{T};\mathcal{F}}_{m;\Delta;\Gamma}(branch_i) = (s_{\overline{\mathcal{M}i}}, t_{\overline{\mathcal{M}i}}, u_{\overline{\mathcal{M}i}})$. In this case, $s_{\mathcal{M}} \geq s_{\mathcal{M}0}$ and $s_{\mathcal{M}} \geq s_{\overline{\mathcal{M}i}}$ for $1 \leq i \leq n$. In addition, for $u_{\mathcal{M}} \geq u_{\mathcal{M}0}$ and $u_{\mathcal{M}} \geq u_{\overline{\mathcal{M}i}}$ for $1 \leq i \leq n$. Furthermore, $t_{\mathcal{M}} = t_{\mathcal{M}0} + \Sigma_{i \in 1 \ldots n} t_{\overline{\mathcal{M}i}}$. Let $s \geq s_{\mathcal{M}}$, $t \geq t_{\mathcal{M}}$, and $u \geq u_{\mathcal{M}}$. Here, from the induction hypothesis with e_0, there exists t_0, l_0, H_0, σ_0 such that $[s]E \mid [t]H \vdash^{\mathcal{T};\mathcal{F}}_{\mathcal{L}} e_0 \Downarrow_u l_0; [t_0]H_0$, $t_0 \geq t - t_{\mathcal{M}0}$, $H_0, \mathcal{T} \vdash l_0 : \sigma_0$, $\sigma_0 \downarrow = ev_\Delta[\![\rho^\psi \downarrow]\!]$, and $H_0(l_0) = \chi[k](l_1, \ldots, l_m)$. In $\mathcal{M}$, branches are searched by $\overline{\mathcal{M}}$. Here $\overline{\mathcal{M}}$ generates a set of assignments $\mathbf{A}$ to the size variables that appear in the constructor. It also generates assignments b for fixed-size size variables. Let C be the size constraint on the size variables introduced in the constructor, and C' be the size constraint obtained by typing the expression of the branch; then, $\{(\Delta, a, b) \mid a \in \mathbf{A}\}$ is a set of models of $C \wedge C'$. Thus, for $d \in \{(\Delta, a, b) \mid a \in \mathbf{A}\}$, $(\mathcal{L}, (E, \{x_i \mapsto l_i\}_{i \in 1 \ldots m}), H_0, (\Gamma, \{x_i : \pi_i\}_{i \in 1 \ldots m}), d)$-consistent holds. Hence, the induction hypothesis is applicable with the expression in the branch, and we can prove the statement using the same procedure as that for T-LET. □

Akihiko Yokoyama is a doctoral course student in the Department of Computer Science at Tokyo Institute of Technology. He received his Master of Engineering from Tokyo Institute of Technology in 2021. He is interested in programming language theory, type system, compiler, software verification, and resource-aware programming language.

Sosuke Moriguchi is an assistant professor in the Department of Computer Science at Tokyo Institute of Technology. He received his Doctor of Engineering from Tokyo Institute of Technology in 2013. He is interested in formal methods for software engineering. He is a member of IPSJ, JSSST, and ACM.



Takuo Watanabe is a professor in the Department of Computer Science at the Tokyo Institute of Technology. He received his Ph.D. in Science from the Tokyo Institute of Technology in 1991. His research is mainly situated in the field of programming models and languages, formal methods, and embedded systems. His recent interest is to understand the nature of reactive programming models for embedded/cyber-physical systems.

© 2021 Information Processing Society of Japan