PARTIAL EVALUATION FOR CONCURRENT SOFTWARE FINITE STATE VERIFICATION

By

MUHAMMAD WAHYU NANDA

Sarjana Teknik

Institut Teknologi Bandung

Bandung, Indonesia

1994

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1999

PARTIAL EVALUATION FOR CONCURRENT SOFTWARE FINITE STATE VERIFICATION

Thesis Approved:

J Chandler
Thesis Adviser

D. E. Hedrick

Dean of The Graduate College

Wayne B. Powell

ACKNOWLEDGMENTS

This thesis is a learning experience for me. I thank my adviser, Dr. John Hatcliff, who guide me with his knowledge and encouragement so I am finally able to finish this thesis. I also appreciate the opportunity and trust he gave to me as his research assistant.

My special thank to Dr. John P. Chandler for serving as my final adviser. His help and kindness make this thesis possible. And I also appreciate the opportunity and trust he gave to me as a teaching assistant during my study in Oklahoma State University.

I offer my gratitude to Dr. G. E. Hedrick for serving as my committee and providing critical reviews on this manuscript.

I also thanks Dr. H. K. Dai for serving as my committee in such a short notice. His expertise and insight were very helpful to this thesis and to me personally.

I would like to acknowledge The Department of Computer Science, especially to Beau Turner and Anna Ventris, who always there when I needed their help.

I extend my acknowledgment to my friend, Fajar Setiawan, only with his help that I can complete all the requirements to finish this thesis.

My personal thanks to my lovely wife, Novita Sari, for her constant companion, support, and encouragement.

Finally, I thank my beloved mother, Hayatun Nismah, and my beloved father, Rumzi Bey Rasjad, for their unconditional support not only during my education, but throughout my entire life.

TABLE OF CONTENTS

1 INTRODUCTION 1
- 1.1 Safety-critical high assurance concurrent software verification 1
- 1.2 Limitation of current FSV tools 2
- 1.3 Pre-processing using partial evaluation as an extension of current FSV tools 2
- 1.4 Overview 3

2 PARTIAL EVALUATION 4
- 2.1 What is partial evaluation? 4
- 2.2 How does partial evaluation work? 5
- 2.3 Offline and online partial evaluation 7
  - 2.3.1 Offline partial evaluation 7
  - 2.3.2 Online partial evaluation 9
  - 2.3.3 Assessments 9
- 2.4 Previous works on partial evaluation 10
  - 2.4.1 Similix 10
  - 2.4.2 Schism 10
  - 2.4.3 Fuse 10
  - 2.4.4 C-mix 10
  - 2.4.5 Tempo 11
  - 2.4.6 Fortran specializer F-Spec 11
  - 2.4.7 Assessments 11

3 FINITE STATE VERIFICATION 12
- 3.1 Introduction 12
  - 3.1.1 Program verification approaches 12
  - 3.1.2 Finite state verification 12
- 3.2 The SMV model checking system 13
  - 3.2.1 SMV transition system 13
  - 3.2.2 Computational Tree Logic (CTL) 14
  - 3.2.3 Model checking 16

4 PROBLEM FORMULATION 17
- 4.1 State of the art 17
  - 4.1.1 Configurable systems 19
  - 4.1.2 Replicated Workers Framework (RWF) 20
- 4.2 Goal 24

5 INTERPRETER 26
- 5.1 Syntax of the FCL 26
  - 5.1.1 Formal definition of FCL syntax 27
- 5.2 Type checking 27
  - 5.2.1 Type 28

5.2.2 Type compatibility . . . 29
5.2.3 Type checking domains and values . . . 29
5.2.4 Type checking of type definitions . . . 31
5.2.5 Type checking of variable declarations . . . 31
5.2.6 Type checking of types . . . 32
5.2.7 Type checking of expressions . . . 33
5.2.8 Type checking of operation types equivalence . . . 35
5.2.9 Type checking of assignments . . . 36
5.2.10 Type checking of assignment types equivalence . . . 37
5.2.11 Type checking of jumps . . . 37
5.2.12 Type checking of blocks . . . 38
5.2.13 Type checking of programs . . . 39
5.3 Evaluation of the FCL interpreter . . . 39
5.3.1 Execution traces . . . 39
5.3.2 Evaluation domains and values . . . 40
5.3.3 Evaluation of variable declarations . . . 41
5.3.4 Evaluation of array members initialization . . . 42
5.3.5 Evaluation of initializations . . . 43
5.3.6 Evaluation of left-expressions . . . 44
5.3.7 Evaluation of expressions . . . 45
5.3.8 Evaluation of operations . . . 47
5.3.9 Evaluation of assignments . . . 48
5.3.10 Evaluation of assignment-updates . . . 49
5.3.11 Evaluation of jumps . . . 50
5.3.12 Evaluation of blocks . . . 51
5.3.13 Evaluation of programs . . . 52

6 SPECIALIZATION **53**
6.1 Methodology . . . 53
6.2 Extended binding-time annotations . . . 54
6.3 Three-level FCL language (FCL-3) . . . 62
6.4 Binding-time type . . . 62
6.5 Binding-time type checking . . . 65
6.5.1 Binding-time type checking domains and values . . . 65
6.5.2 Binding-time type checking of type definitions . . . 67
6.5.3 Binding-time type checking of variable declarations . . . 68
6.5.4 Binding-time type checking of types . . . 70
6.5.5 Binding-time type checking of left-expressions . . . 71
6.5.6 Binding-time type checking of expressions . . . 73
6.5.7 Binding-time type checking of operation bt-types equivalence . . . 76
6.5.8 Binding-time type checking of assignments . . . 77
6.5.9 Binding-time type checking of assignment bt-types equivalence . . . 78
6.5.10 Binding-time type checking of jumps . . . 78
6.5.11 Binding-time type checking of blocks . . . 79
6.5.12 Binding-time type checking of programs . . . 80
6.6 Three-level FCL semantics . . . 80
6.6.1 Partial evaluation traces . . . 80
6.6.2 Semantics domains and values . . . 81
6.6.3 Specialization of type definitions . . . 82
6.6.4 Specialization of variable declarations . . . 83
6.6.5 Specialization of array members initialization . . . 85
6.6.6 Specialization rules to generate residual variable declarations . . . 85
6.6.7 Specialization of residual types . . . 86

6.6.8 Specialization of static types 86
6.6.9 Specialization of left-expressions 87
6.6.10 Specialization of expressions 90
6.6.11 Specialization of operations 94
6.6.12 Specialization of assignments 95
6.6.13 Specialization of assignment-updates 96
6.6.14 Specialization of jumps 98
6.6.15 Specialization of blocks 100
6.6.16 Specialization of programs 103
6.7 Implementation and Specialization examples 103

7 CONCLUSION **108**
7.1 Summary 108
7.2 Assessments 108
7.3 Future work 109

A INTERPRETER **113**
A.1 Module for syntax 113
A.2 Parser 114
A.3 Type checker 116
A.4 Evaluator 120
A.5 Module for tables manipulation 126

B SPECIALIZER **132**
B.1 Module for syntax 132
B.2 Parser 134
B.3 Binding-time type checker 137
B.4 Specializer 149
B.5 Unparser 163
B.6 Module for tables manipulation 166
B.7 Miscellaneous functions 171

LIST OF FIGURES

1.1 Integration of partial evaluation and FSV tools 2

2.1 A partial evaluator 5
2.2 String matching function `strstr` 6
2.3 `strstr` specialized to `"aab"` 6
2.4 Binding-time annotated function `strstr` 9

3.1 SMV input program example `main` 14
3.2 The SMV model corresponding to `main` in figure 3.1 15

4.1 The structure of Replicated Workers Framework 21
4.2 The RWF structure and task types 22
4.3 Original and specialized RWF implementation of task type `ActivePool` . . 23
4.4 Heap structure of RWF for three workers 24
4.5 Original and specialized RWF implementation of function `Create` 25

5.1 FCL program for power function 27
5.2 FCL syntax 28
5.3 Type checking domains and values 30
5.4 Type checking rules for type definitions 31
5.5 Type checking rules for variable declarations 32
5.6 Type checking rules for types 33
5.7 Type checking rules for expressions 34
5.8 Type checking rules for operation types equivalence 35
5.9 Type checking rules for assignments 36
5.10 Type checking rules for assignment types equivalence 37
5.11 Type checking rules for jumps 38
5.12 Type checking rules for blocks 39
5.13 Type checking rule for programs 40
5.14 Evaluation rules domains and values 41
5.15 Evaluation rules for variable declarations 42
5.16 Evaluation rules for array members initialization 42
5.17 Evaluation rules for initializations 43
5.18 Evaluation rules for left-expressions 44
5.19 Evaluation rules for expressions 45
5.20 Evaluation rules for operations *(part 1)* 46
5.21 Evaluation rules for operations *(part 2)* 47
5.22 Evaluation rules for assignments 48
5.23 Evaluation rules for assignment-updates 49
5.24 Evaluation rules for jumps 50
5.25 Evaluation rules for blocks 51
5.26 Evaluation rule for programs 52

6.1 The annotated FCL power function program 54
6.2 FCL-ann syntax 55
6.3 The flow of the application specialization process 56
6.4 Example of source code and specialized code of nat type 57
6.5 Example of source code and specialized code of array type 58
6.6 Example of source code and specialized code of access type 59
6.7 Example of source code and specialized code of record type 60
6.8 Three-level FCL syntax domains 62
6.9 Three-level FCL grammar 63
6.10 Binding-time type checking rules domains and values 66
6.11 Binding-time type checking rules for type definitions 67
6.12 Binding-time type checking rules for variable declarations 68
6.13 Binding-time type checking rules for types 69
6.14 Binding-time type checking rules for left-expressions (*part 1*) 70
6.15 Binding-time type checking rules for left-expressions (*part 2*) 71
6.16 Binding-time type checking rules for expressions (*part 1*) 72
6.17 Binding-time type checking rules for expressions (*part 2*) 73
6.18 Binding-time type checking rules for operation bt-types equivalence (*part 1*) 74
6.19 Binding-time type checking rules for operation bt-types equivalence (*part 2*) 75
6.20 Binding-time type checking rules for assignments 76
6.21 Binding-time type checking rules for assignment bt-types equivalence 77
6.22 Binding-time type checking rules for jumps 78
6.23 Binding-time checking rules for blocks 79
6.24 Binding-time type checking rule for programs 80
6.25 Specialization rules semantics domains and values 81
6.26 Specialization rules for type definitions 83
6.27 Specialization rules for variable declarations 84
6.28 Specialization rules for array members initialization 85
6.29 Specialization rules for residual variable declaration generations 86
6.30 Specialization rules for residual types 87
6.31 Specialization rules for static types (*part 1*) 88
6.32 Specialization rules for static types (*part 2*) 89
6.33 Specialization rules for left-expressions (*part 1*) 90
6.34 Specialization rules for left-expressions (*part 2*) 91
6.35 Specialization rules for expressions (*part 1*) 92
6.36 Specialization rules for expressions (*part 2*) 93
6.37 Specialization rules for expressions (*part 3*) 94
6.38 Specialization rules for operations (*part 1*) 95
6.39 Specialization rules for operations (*part 2*) 96
6.40 Specialization rules for assignments 97
6.41 Specialization rules for assignment-updates (*part 1*) 98
6.42 Specialization rules for assignment-updates (*part 2*) 99
6.43 Specialization rules for jumps 100
6.44 Specialization rules for blocks 101
6.45 Specialization rules for programs 102
6.46 The original FCL-ann fragment of the RWF structure and task types 104
6.47 The specialized FCL fragment of the RWF structure and task types 105
6.48 The original FCL-ann fragment of the RWF function `Create` 106
6.49 The specialized FCL fragment of the RWF function `Create` 107

CHAPTER 1

INTRODUCTION

1.1 Safety-critical high assurance concurrent software verification

The need for applications that require safety-critical concurrent software systems is steadily increasing. For example, applications in aircraft systems, hospitals, chemical and pharmaceutical plants, and industrial military must be extremely reliable since any malfunction can cause dramatic and serious consequences. To develop safety-critical high assurance concurrent software, one must be able to verify that certain important properties hold without having to run the software in real situation.

Finite state verification (FSV) techniques (originally developed for hardware verification) are emerging as a promising technology for assuring high-quality in modern software systems. In FSV, one describes the behavior of a computational system using a finite state transition system and the specification to be verified using some formalisms (*e.g.* temporal logics [29] or finite state automata [22]). The specification describes the properties that should hold true at particular states.

Verification in FSV is performed by exhaustively enumerating all reachable transition system states while checking that the specification is satisfied at each state. Even though this method of verification requires exponential time in the worse case, it has been used effectively to validate many applications including network protocols [23, 31, 39], graphical user interfaces [17], railway interlocking systems [9], and industrial control systems [8, 20].

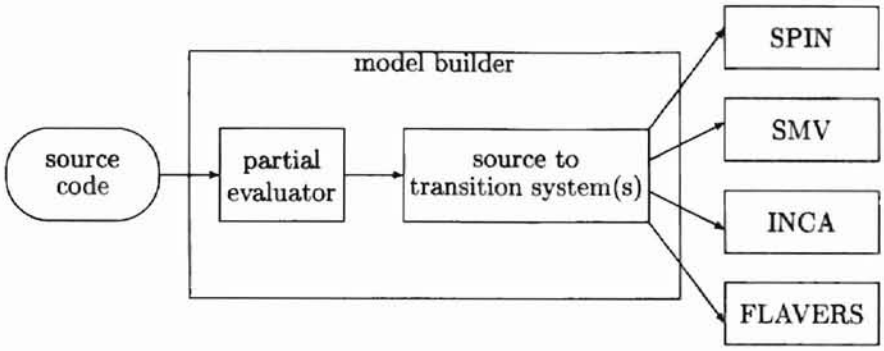


Figure 1.1: Integration of partial evaluation and FSV tools

1.2 Limitation of current FSV tools

To date, most finite state verification tools have not been targeted at mainstream programming languages.[1] In addition, these tools have not dealt with some fundamental program structures that are used widely in practice. Because of that, programs often must be transformed by hand before they can be processed by existing tools. These transformations, which typically involve unfolding selected procedure calls and migrating dynamically allocated objects to compile-time allocated data, are tedious, time-consuming and error-prone.

1.3 Pre-processing using partial evaluation as an extension of current FSV tools

Partial evaluation (PE) is a technology for *automatic* program specialization and customization that can *automatically* unfold selected procedure calls and migrate dynamically allocated objects to compile-time allocated data. It has been applied in many different areas with great success, for example, circuit simulation, computer graphics, and neural network applications.

Finite state verification tools accept finite state representations of programs created by some source to finite state transition system mapping. By pre-processing source programs with a partial evaluator before applying the source to transition systems mapping

[1] An exception is the FLAVERS system[18] that processes a subset of Ada.

(as illustrated in figure 1.1), we broaden the class of systems to which finite state verification can be applied. The idea is that the partial evaluator can automatically perform the transformations described above that have been previously performed by hand.

Because partial evaluation is a a source-to-source program transformation, we can implement the pre-processor independently from FSV tools. Thus, the pre-processor can *extend the applicability of existing tools without having to modify them.*

Our strategy for developing the required PE pre-processor is as follows:

- to design and to formalize the fundamental techniques required to extend PE technology to FSV pre-processing,
- to assess the feasibility of this approach by implementing a *Scheme prototype* for a *flow-chart language*, and
- to apply and test this system on several realistic examples.

1.4 Overview

The rest of this thesis is organized as follow.

Chapter 2 describes the basic principles of partial evaluation and highlights the partial evaluation techniques that we need to apply in our work.

Chapter 3 gives a brief overview of finite state verification techniques and tools.

Chapter 4 explains problems in current FSV tools and our approach for solving these problems.

Chapter 5 explains the interpreter that we developed. In this chapter we present the formal definition of the FCL language that is processed by the interpreter, the semantics of the interpreter type checking and the semantics of the interpreter evaluation.

Chapter 6 is the main chapter of this thesis. This chapter explains the semantics of the binding-time type checking and the specializer.

Chapter 7 holds the conclusion of our thesis.

CHAPTER 2

PARTIAL EVALUATION

2.1 What is partial evaluation?

Consider a program p with two inputs in1 and in2. If we know in advance the value for in1, a partial evaluator can *transform* program p by pre-computing the parts of p that depend only on in1 to a new *specialized* program p_{in1}. The program p_{in1} is also called the *residual* program. For the transformation to be correct, running p_{in1} with the remaining input in2 will yield the same result that would be produced by running p on both inputs in1 and in2. The computation is illustrated in figure 2.1, and can be summarize as:

$$\begin{aligned} \texttt{result} &= [\![\texttt{p}]\!][\texttt{in1}, \texttt{in2}] \\ &= [\![([\![\texttt{mix}]\!][\texttt{p}, \texttt{in1}])]\!][\texttt{in2}] \ [1] \\ &= [\![p_{in1}]\!][\texttt{in2}] \end{aligned}$$

Here, [[p]][in] denotes running program p on input in (note that p can have more than one input value).

Figure 2.2 shows example of partial evaluation of GNU library implementation of the string matcher strstr function.[2] The string-matcher takes a string (haystack) and pattern (needle) as input, and returns a pointer to the first occurrence of the pattern in the string (NULL otherwise). Figure 2.3 shows the result obtained automatically by partial evaluating strstr with respect to the input string "aab". Essentially, strstr has been customized by *evaluating* statements involving needle (whose value is known), and by *residualizing* (*i.e.* emitting code for) statements involving haystack (whose value is unknown).

[1] For historical reasons, partial evaluator is often named mix.

[2] This example is adapted from Andersen [1].

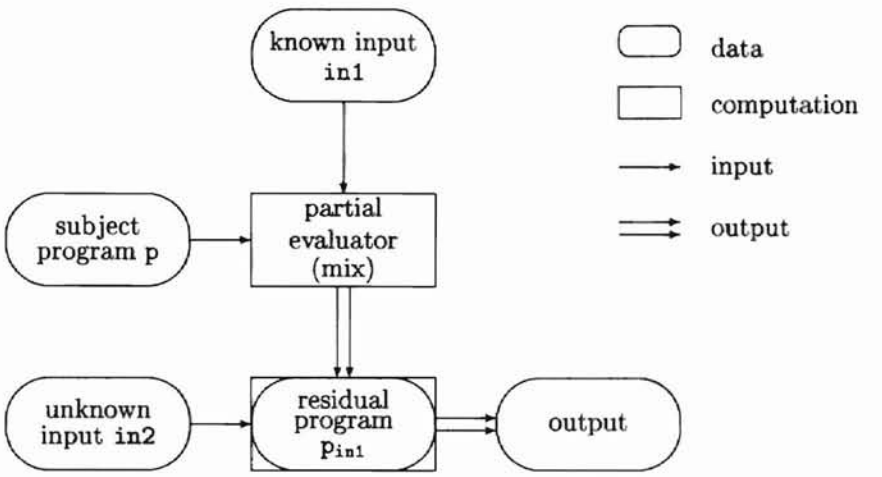


Figure 2.1: A partial evaluator

2.2 How does partial evaluation work?

Three major techniques used in partial evaluation are well known from program transformation: *symbolic computation*, *unfolding function calls and loops* and *program point specialization* [26].

Symbolic computation is a technique to pre-compute all expressions depending on known data (*i.e.* using constant propagation). Unfolding is a technique to unfold procedure calls and loops. For example, the specialized program `strstr-1` of figure 2.3 was obtained by symbolically computing constructs involving `needle` (*e.g.* `strchr(needle,'\0')`, `needle_end - needle`, *etc.*), by unfolding the `do ... while (--n >= needle)` loop, and by desugaring the `for` loop to the familiar `if/goto` equivalent.

The idea of program point specialization is that a single function or label in program `p` may appear in the residual program p_{in1} in several specialized versions, each corresponding to data determined at partial evaluation time [26]. For example, the if statement `if (*h-- != *n)` in the source program `strstr` of figure 2.2 appears in the residual program `strstr-1` of figure 2.3 in three specialized versions (one for each character in the original `needle` string "aab"): `if (*h-- != 'a')`, `if (*h-- != 'a')`, and

```
/* Copyright (C) 1991, 1992 Free Software Foundation, Inc. */
/* Return the first occurrence of NEEDLE in HAYSTACK */

char *strstr(char *haystack, char *needle)
{
  register const char *const needle_end = strchr(needle,'\0');
  register const char *const haystack_end = strchr(haystack,'\0');
  register const size_t needle_len = needle_end - needle;
  register const size_t needle_last = needle_len - 1;
  register const char *begin;

  if (needle_len == 0) return (char *) haystack;
  if ((size_t) (haystack_end - haystack) < needle_len) return NULL;
  for (begin = &haystack[needle_last]; begin < haystack_end; ++begin) {
      register const char *n = &needle[needle_last];
      register const char *h = begin;
      do
        if (*h-- != *n) goto loop;
      while (--n >= needle);
      return (char *)h+1;
    loop:;
    }
  return NULL;
}
```

Figure 2.2: String matching function `strstr`

```
char *strstr-1 (char *haystack)
{
   char *haystack_end;
   char *begin;
   char *h;
   haystack_end = strchr (haystack, '\0');
   if ((int) haystack_end - haystack < 3) return 0;
   begin = &haystack[2];
sys_label:
   if (begin < haystack_end) {
      h = begin;
      if (*h-- != 'b') {++begin; goto sys_label;
      } else
         if (*h-- != 'a') {++begin; goto sys_label;
         } else
            if (*h-- != 'a') {++begin; goto sys_label;
            } else
               return (char *) h + 1;
   } else
      return 0;
}
```

Figure 2.3: `strstr` specialized to `"aab"`

`if (*h-- != 'b').`

Although this example is extremely simple, the concepts involved are quite general. Clearly `strstr` is a general function which can be used to search for arbitrary strings with the `needle` parameter as a *specification* of configuration information (the string to search for). When given a string to search *via* the unknown `haystack` parameter, `strstr` *interprets* the known specification `needle` with three techniques described above, yielding a series of computational steps which search `haystack` for a specific string. Partial evaluation automatically customizes the general program to a more efficient *instantiation* `strstr-1`. The generality of `strstr` means that it can handle or adapt to different search string specifications. Partial evaluation automatically customizes the general program to a more efficient *instantiation* `strstr-1`. The instantiation can be *reconfigured* by specializing the general program `strstr` with respect to a different specification.

In this example, customization involved removing the *interpretive overhead* associated with the repeated looping over `needle` by *compiling* (*i.e.* coding) the specification `needle` into the control structure. Programs of a similar interpretive nature are: a ray tracer that repeatedly interprets scene information, a circuit simulator that repeatedly interprets a circuit specification, a DNA-string matcher that repeatedly interprets the DNA sequence for which to search. Many programs repeatedly interpret configuration information representing the current hardware or software environment, *etc.*

2.3 Offline and online partial evaluation

There are two different strategies for building a partial evaluator: *offline* and *online* strategies.

2.3.1 Offline partial evaluation

With the offline strategy, partial evaluation is divided into three stages or phases: a *pre-phase*, a *specialization phase*, and a *post-phase*. This strategy is called offline because specialization is guided by information computed from previous phase, not by concrete

values computed during specialization.

During the pre-phase, an offline partial evaluator performs parsing, *binding-time analysis* (BTA) and possibly other analyses depending on the nature of the transformation involved.

The binding-time analysis accepts as input an abstract syntax tree (AST) representing a source program and classification of the program parameters as either *static* (known) or *dynamic* (unknown). Then, the analysis *propagates* this information throughout the AST to compute *conservatively* the division of program constructs into two categories, either *eliminable* constructs if they depending only on static data or *residual* constructs if they depending on dynamic data.

The analysis is conservative in that sense that if there is not enough information definitely to classify program constructs as eliminable or residual, then it is always safe to classify them as residual (worst-case approximation).

The output of binding-time analysis is an annotated program in a *two-level representation* [33] where every language construct appears in two versions, the *non-underlined* constructs (representing eliminable constructs) and the *underlined* constructs (representing residual constructs). For example, figure 2.4 is the annotated program for function `strstr` of figure 2.2. In the figure, the non-underlined constructs are eliminable constructs and depending only on known data `needle`. The underlined constructs are residual constructs and may depending on unknown data `haystack`.

During the specialization phase, the transformation is guided by binding-time annotations from the annotated program, where eliminable constructs (*i.e.* known expressions) can be computed away at specialization time and residual constructs (*i.e.* unknown expression) can't be computed at specialization time (appear in residual program).

The post-phase of offline partial evaluator performs final transformations such as: unfolding, unparsing, or other transformations.

```
char *strstr (char *haystack, char *needle)
{
  register const char *const needle_end = strchr(needle,'\0');
  register const char *const haystack_end = strchr(haystack,'\0');
  register const size_t needle_len = needle_end - needle;
  register const size_t needle_last = needle_len - 1;
  register const char *begin;

  if (needle_len == 0) return (char *) haystack;
  if ((size_t) (haystack_end - haystack) < needle_len) return NULL;
  for (begin = &haystack[needle_last]; begin < haystack_end; ++begin) {
      register const char *n = &needle[needle_last];
      register const char *h = begin;
      do
         if (*h-- != *n) goto loop;
      while (--n >= needle);
      return (char *)h+1;
    loop:;
    }
  return NULL;
}
```

Figure 2.4: Binding-time annotated function `strstr`

2.3.2 Online partial evaluation

The online partial evaluator has only one phase with possibly a *post-phase* at the end. It decides on the fly which operations can be performed at specialization time. This strategy is not guided by a binding-time separation from pre-phase as the offline strategy is, but must decide from actual data whether to reduce or to residualize the expression. The post-phase for online strategy also performs final transformations like for the offline strategy.

2.3.3 Assessments

Both of these strategies have advantages and disadvantages. An online partial evaluator is more aggressive than an offline one, so it sometimes can exploit more static information than an offline partial evaluator. Thus, it can yield programs that are more specialized. An offline partial evaluator is faster than an online because binding-time checks do not need to be performed on the fly. Beside that, the offline strategy is also better for handling imperative programs, because imperative programs with pointers and side-effects require sophisticated analyses (*e.g.* alias analysis, side-effect analysis, *etc.*) before the actual specialization can take place. It is easier to perform these analyses as part of pre-phase in conjunction with

binding-time analysis.

2.4 Previous works on partial evaluation

In this section we will review some of previous works on partial evaluation. We limit our discussion to publicly available systems that have been used in one or more significant applications.

2.4.1 Similix

Developed at DIKU by Anders Bondorf and Olivier Danvy [7, 5, 6, 26], Similix is probably the most widely used partial evaluator right now. It is an offline, self-applicable partial evaluator that handles higher order function and was written in a subset of Scheme (a dialect of Lisp).

2.4.2 Schism

Schism [12, 13] was developed by C. Consel and is similar to Similix. It is an offline partial evaluator for a subset of Scheme with an additional front-end for Standard ML. Compared to Similix, it has a more sophisticated binding-time analysis, and a language of *filter* which essentially is a macro language for customizing the specialization process.

2.4.3 Fuse

In contrast to Similix and Schism, Fuse [38, 34, 37] is an online partial evaluator for a subset of Scheme. It has a back-end that can generate either Scheme or C code. Fuse was developed at Stanford University.

2.4.4 C-mix

As mentioned earlier, C-mix is a partial evaluator for ANSI C developed as part of Andersen's Ph.D. dissertation [1]. C-mix and Tempo (discussed below) are the two most sophisticated partial evaluators developed to date because they incorporates complex features of the imperative language C, such as, structures, multidimensional arrays, and pointers,

and it performs sophisticated analysis to handle those features. C-mix has been applied successfully to specialize a computer graphics ray-tracer.

2.4.5 Tempo

Tempo was developed at the University of Rennes/IRISA [15, 14]. It includes a polyvariant binding-time analysis that is more sophisticated than the one used in C-mix, and this improvement often yields better specialization results. Furthermore, Tempo also provides facilities for "run-time specialization" where specialized executable code is emitted at run-time (in contrast to specialized source code being emitted before compile-time as in standard partial evaluation). Tempo has been applied to several different kinds of system code including remote procedure call code and operating system kernels.

2.4.6 Fortran specializer F-Spec

F-Spec is an offline partial evaluator for FORTRAN programs developed by Robert Glück and his students at the University of Copenhagen and the University of Vienna [3]. F-Spec has been used to specialize a variety of numerical programs including linear equation solving by Gaussian elimination, polynomial approximation using the telescope algorithm, and numerical integration using the trapezoid rule, and a Fast Fourier Transformation [4].

2.4.7 Assessments

Most of existing partial evaluators that mentioned in previous subsection are focused on Scheme or C [3]. Among them, C-mix and Tempo are the most relevant systems for our work because they process imperative programs. For our work, we must scale up techniques used in C-mix and Tempo to handles Ada features. We have decided to concentrate on techniques in C-mix since the source code is publicly available, the techniques are rigorously justified, and it is well documented.

[3]Many partial evaluator exist for other languages, *e.g.* Logimix is partial evaluator for Prolog.

CHAPTER 3

FINITE STATE VERIFICATION

3.1 Introduction

3.1.1 Program verification approaches

Program verification tools can be separated into two categories: model-based and proof-based.

With the model-based approach, the source program is represented by a *finite model* [1] and properties to be verified are represented by formulas in a certain logic. The verification process attempts to establish that the specification formula holds true in the program model.

With the proof-based approach, the source program is modeled with a *set of formulas* in certain logic and the specification is represented with another formula. One then tries to build a proof show that the set of program formulas entails the specification formula.

The main advantage of the model-based approach is that verification is done automatically. However the restriction to finite models also impedes the applicability of this approach, because generally software has an infinite state space and one must create a mapping between infinite state space and the finite model.

On the other hand, verification in the proof-based approach is applicable to a broader class of problems compared to the model-based approach because one does not need to map source program into a finite abstraction, but this approach is not automatic. It typically requires guidance and expertise from the user in building the necessary proofs.

3.1.2 Finite state verification

Finite state verification is a model-based approach for verifying software and hardware. In FSV, the specification describes a single property of the system, not its full behavior. This

[1] The finite model is an abstraction of an actual physical system that omits irrelevant features.

type of verification reduces the complexity of the verification process. This verification method is called *model checking* and the tool is called *model checker*. This method is often used for verifying concurrent and reactive systems.

There are three steps to be performed in model checking [25]:

1. model the system as a transition relation system using the description language of model checker,

2. code the property to verify using the specification language of model checker,

3. run the model checker using some verification algorithms.

3.2 The SMV model checking system

To give the flavor of a FSV tool, we briefly summarize the Symbolic Model Verifier (SMV) tool [30]. SMV is a tool for checking finite state systems with the specifications in the temporal logic CTL (Computational Tree Logic).

3.2.1 SMV transition system

The input language of SMV describes both the model and the specification. The model is described as a labeled state-transition relation [2], whose state is defined by a collection of state variables of boolean type or scalar type (*i.e.* enumeration, integer range, and fixed array type).

The transition relation of SMV system is coded using a collection of SMV `MODULE`s. Modules define the scope of state variables and state transitions, where state variables are defined after keyword `VAR` and state transition are defined as a collection of parallel assignment statement after keyword `ASSIGN`. To initialize state variables, SMV uses `init(`v`)` command, where v is variable name. The value for state variables in the next step is defined by assigning a value to `next(`v`)`, where v is variable name.

[2] For historical reasons such structures are called *finite state Kripke structures* [24]

```
MODULE main
VAR
  request : boolean;
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   request : busy;
                   1 : {ready,busy};
                 esac;
SPEC
  AG(request -> AF state = busy)
```

Figure 3.1: SMV input program example `main`

Figure 3.1 shows an example of SMV input program. In this example, there are two state variables: `request` of boolean type and `state` of scalar type, which are `ready` or `busy`. The initial value of variable `state` is set to `ready`, while on the other hand variable `request` is not assigned, which leaves the SMV free choices for this variable. The next value of `state` is determined by the current state of the system by assigning it the value of the case expression. Thus, if `request` is true, then the result of the expression is `busy`, otherwise, it is a set of `ready,busy`, which corresponds to a non-deterministic choice between `ready` and `busy`. Being able to define non-deterministic models is useful for describing systems which are not fully implemented yet.

The model corresponding to this SMV program is shown in figure 3.2. In the figure we can see that the model has four states. Each state represent a possible value of two binary variables.

3.2.2 Computational Tree Logic (CTL)

The specification for the SMV system appears as a formula in CTL after the keyword `SPEC`. Computational Tree Logic [10, 11, 21] is a logic that based on *temporal logic*.

The main feature of temporal logic is the *dynamic* formula. A dynamic formula is a formula that is not *statically* true or false as in propositional or predicate logic, but the truth values of the formula can change from one state to the next state. Thus, with temporal

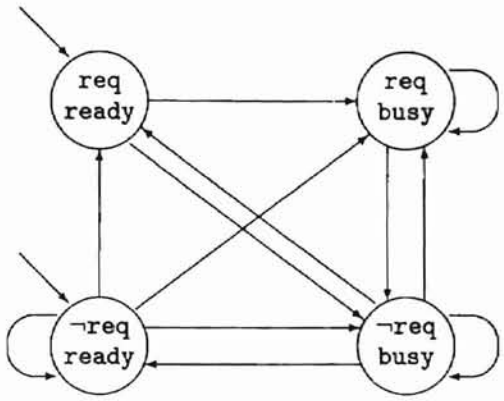


Figure 3.2: The SMV model corresponding to `main` in figure 3.1

logic, a formula can describe sequences of states in time.

Temporal logics can be classified according to their time view: *linear* time logic where time is viewed as chain of time instances, *branching* time logic where times can be branched in the future from a given point of time, and combination of both of linear time and branching time. Time also can be viewed as *discrete* and *continuous*.

Computational Tree Logic is a logic where time is branching (still, CTL uses both linear and branching time operators) and discrete. CTL has been proven powerful for verifying hardware and communication protocols[25], and now it is become a promising verification technique for software systems.

In CTL, there are two type of path quantifier: A and E, which mean *for all computational paths* and *for at least one computational path* respectively. Under the path quantifiers, we can use linear time operators: F, G, U, and X, which mean *at some future state*, *at all future states (globally)*, *at next state*, and *until* respectively.

In figure 3.1, we can see an example of CTL formula after keyword `SPEC`. In this example, the specification `AG(request -> AF state = busy)` means that at all future states in all computational paths, if `request` is true then at some future state in all computational paths, `state` is equal to `busy`.

3.2.3 Model checking

After we define the model in transition relation system using the description language and specify properties using the specification language CTL, SMV compares the given specifications to sequences of transition system states in order to determine whether the specification holds true at particular states. Conceptually, verification is performed by enumerating all reachable transition system states while checking that the specification is satisfied at each state.

The SMV verification process output is a notification that specifications are either true or false with respect to the transition system. If the output is false, SMV will also produces a *counter-example*, which is a trace of the system behavior that causes the failure. This counter-example is useful for system debugging.

CHAPTER 4

PROBLEM FORMULATION

We wish to consider using FSV techniques to verify properties of concurrent software. As the reader may have noticed from the previous chapter, a model checker input language is not in a full-fledged programming language. The source code usually is written in general purpose programming languages, *e.g.* Ada, but each model checker can only accept a description language tailored for the particular FSV tool. This language is a special purpose language for constructing finite state models (*e.g.* the SMV *guarded-transition language* for the SMV model checker).

Applying a model checker like SMV for verifying software written in Ada would require users to *compile* the source code into the SMV guarded-transition language. This compilation in essence forms an abstraction of the source code semantics.

The construction of such compiler is an open research problem. Currently, researchers are manually compiling from languages such as Ada to SMV. We are convinced that partial evaluation technology can be the key component in the construction of a compiler to translate Ada to model checker description languages automatically.

4.1 State of the art

Currently, researchers at Kansas State University and University of Massachusetts [19] have developed tools for translating a very restricted subset of Ada (called *Finite-state Ada*) to a transition system that is suitable for model checking. This Finite-state Ada has the following restrictions:

1. All variables must have finite domains and they must be one of the following types:

    - boolean, enumerated, or subrange types,

- records or arrays where their components must also have finite domains.

2. There is no dynamic creation of data objects or tasks (*dynamism*).

To use these tools, researchers currently have to hand-translate programs written in full Ada to Finite-state Ada.

Creating programs that satisfy the first restriction requires that one create appropriate abstractions of variable values. This can be done systematically using *abstract interpretation* technology [16]. We do not address this problem and it is currently being addressed by Shawn Laubach in his Master thesis [28].

In this thesis, we focus on techniques that can automatically create programs that satisfy the second restriction. The problem with dynamism is that one cannot construct a finite-state model for a system that is capable of dynamically creating arbitrarily many components. Not all programs can be transformed into a form where dynamic data and task creation have been removed. However, there is a special class of systems called *configurable systems* where such dynamism can often be removed.

A configurable system can be viewed as proceeding in two phases: a configuration phase and a computation phase [27]. During the configuration phase, user-supplied configuration parameters are received, tasks/threads are created and initialized, and the interconnection topology between tasks/threads is established. During the computation phase of a configurable system, the configuration is remains fixed.

The key to implementing a configurable system software is the use of dynamic task creation and indirect referencing to achieve flexibility in tasks interaction (*e.g.* communication, synchronization). Model checkers cannot deal directly with the dynamic object creation that occurs during the configuration phase The idea is to remove the dynamism using specialization: the original program is specialized with respect to the configuration parameters to obtain a source-level representation of a particular system configuration. This specialized version can be converted into a finite-state model on which properties of the computation

phase can be checked.

Note that this approach does not give a validation of the complete system. A complete validation of correctness properties of the system would require reasoning about the correctness of the configuration change process. However, the approach does allow many interesting properties to be verified.

In summary, our thesis is that partial evaluation technology [26] can eliminate dynamism and resolve indirect references by specializing a configurable system to a specific fixed-size system with statically known task interaction structures.

To use a partial evaluator for specializing a configurable system, parameters which control the configuration must be provided to the system. The partial evaluator *interprets* the configuration phase of the system using these parameters and produces a program with execution behavior equivalent to the execution of the original program for a specific configuration.

Because partial evaluation performs a source-to-source transformation, it can be used as a pre-processing step before applying existing FSV tools. Thus, partial evaluation can extend the applicability of FSV tools without having to modify the tools themselves.

4.1.1 Configurable systems

Concurrent and distributed systems are configurable along different non-functional dimensions [2]. For example, the number of tasks (independent threads of execution) in the system may be a parameter. This kind of flexibility allows the performance of the system (*e.g.* throughput or availability) to be tailored to a problem's needs. For example, one might select the number of tasks to match available resources (*e.g.* processors) or to match the available parallelism in the input data to be processed. Key elements in the configuration of a concurrent system include:

- the number of tasks,
- the number of communication channels per task, and

- the inter-connection of communication channels.

System configuration can occur at different times (*i.e.* compile time or run-time). Configuring a system at compile time allows for optimizing the performance of the specified system configuration, but eliminates the possibility of run-time changes in configuration. Some systems may perform significant computation on input data to determine an appropriate configuration (*e.g.* use of cellular-automata models to determine sub-problem dependence in particle simulations [32]). Configuration at run-time requires that tasks, communication channels, and channel inter-connection information can be allocated or computed as the system executes. This is usually achieved by using dynamically allocated data and tasks and associated indirect referencing of those entities.

4.1.2 Replicated Workers Framework (RWF)

In this subsection, we will explain an example of a configurable system, the replicated workers framework (RWF), and transformations needed to specialize this configurable system so it can be accepted by FSV tools.

The replicated workers framework [2] expresses a notion of a group of similar concurrent computational elements, called workers. Each element of workers repeatedly accesses data from a shared work pool (repository), processes the data, produces new data elements, and returns these new data elements back to the pool. Figure 4.1 illustrates the interaction structure between the pool and workers in the replicated workers framework.

In the RWF, users are able to configure the framework behavior through number of parameters:

- the number of workers,
- the number of work items retrieved from the pool by a worker,
- the type of work and result data,
- whether computations execute as a synchronous or asynchronous invocation, and

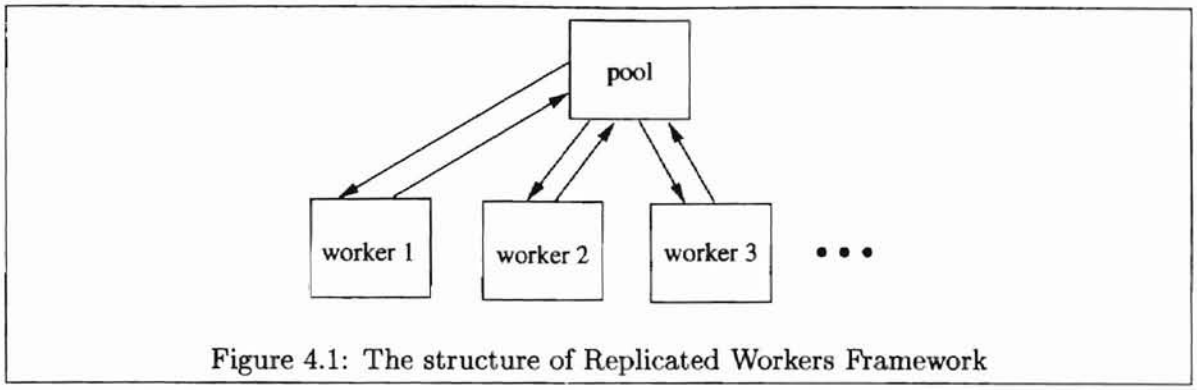


Figure 4.1: The structure of Replicated Workers Framework

- the computation to be performed by a worker on a data or result item.

Workers, the pool, and workers synchronization are implemented as dynamically created instances of several task types. The framework uses dynamic task creation so that it is possible for a single application to use multiple instances of the framework.

We can see the definition of structure types and task types of the RWF in figure 4.2 and also code fragment of the pool task body in figure 4.3. The pool task (`ActivePool`) needs to call an entry of each of the worker tasks, and each worker task (`ActiveWorker`) also need to call entries of the pool task. The cyclical entry calls between the pool and worker tasks means that task references for workers will not be known at pool allocation time and have to be allocated dynamically. The flexibility of this structure allows incremental extension of the framework during execution.

If one is given the number of workers in the system, then the number of task components and the references required for components communication can be determined statically. Thus, specializing the RWF to a particular number of workers removes the problem of dynamic allocation in the RWF and allows a finite-state model to be constructed.

An important property is that, for a given configuration, once the dynamic data and threads of control are created, the structure of inter-component references remains fixed and the heap objects themselves are preserved throughout the lifetime of a that configuration. In figure 4.4 we can see the heap structure of the RWF for three workers. The goal is to encode the heap structure that is present at that time into a static form by changing heap

```
type WorkerInfo is record
  c      : Collection;
  aw     : ActiveWorkerRef;
  ap     : ActivePoolRef;
  numIn  : Natural;
end record;
type Worker is access WorkerInfo;
type WorkerVector is array(Natural range <>) of Worker;

type CollectionInfo(max : Natural) is record
  workers    : WorkerVector(1..max);
  pool       : ActivePoolRef;
  results    : ResultList;
  done       : Boolean;
  resultLock : WorkLockRef;
end record;
type Collection is access CollectionInfo;

task type ActiveWorker(self : Worker) is
  entry StartUp;
  entry ShutDown;
  entry Execute;
end ActiveWorker;
type ActiveWorkerRef is access ActiveWorker;

task type ActivePool(c : Collection) is
  entry StartUp;
  entry ShutDown;
  entry Get(numItems : in Natural;
    newWork : in out WorkList; done : out Boolean);
  entry Put(newWork : in out WorkList);
  entry GetResult(resultItem : out ResultType);
  entry PutResults(newResults : in out ResultList);
  entry Finished;
  entry Execute;
  entry Complete;
end ActivePool;
type ActivePoolRef is access ActivePool;
```

Figure 4.2: The RWF structure and task types

objects to explicitly named and statically allocated objects which also will enable dynamic inter-object references to be converted to static references in terms of the new object names.

In figure 4.5 we can see the original code of function `Create` (left side of the figure) and the specialized version of the function with `numWorkers` = 3 (right side of the figure). Based on the knowledge of the parameter `numWorker`, the partial evaluator can symbolically execute dynamic object allocations, generate new static objects to replace them, and propagate this information throughout the program. For example, `ActivePool` dynamic allocation at line 8 of the original code will be replaced with new unique static declaration of `GEN1ActivePool` and indirect reference to this object at line 17 of the original code can

```
-- original code fragment
task body ActivePool is
  executeDone : Boolean := TRUE;
  workCount, waitPhaseWorkers, idleWorkers :
    Natural;
  work : WorkPool.List;
  ...
  begin
    accept StartUp;
    WorkPool.Create(work);
    workCount := 0;
    Outer: loop
      loop
        select ...
        or accept Execute;
          c.done := FALSE;
          for i in 1 .. c.max loop
            c.workers(i).aw.Execute;
          end loop;
          exit;
        ...
        end select;
      end loop;
      executeDone := FALSE;
      idleWorkers := c.max;
      waitPhaseWorkers := 0;
      loop
        select ...
        or accept GetResult(resultItem :
          out ResultType) do
            Remove(c.results, resultItem);

          end GetResult;
        ...
        end select;
        if idleWorkers = c.max and workCount=0 then
          executeDone := TRUE;
        end if;
        exit when waitPhaseWorkers = c.max;
      end loop;
      ...
      c.done := TRUE;
    end loop Outer;
  end ActivePool;
```

```
-- specialized code fragment
task body GEN1ActivePool is
  executeDone : Boolean := TRUE;
  workCount, waitPhaseWorkers, idleWorkers :
    Natural;
  work : WorkPool.List;
  ...
  begin
    accept StartUp;
    WorkPool.Create(work);
    workCount := 0;
    Outer: loop
      loop
        select ...
        or accept Execute;
          GEN1CollectionInfo.done := FALSE;
          GEN1ActiveWorker.Execute;
          GEN2ActiveWorker.Execute;
          GEN3ActiveWorker.Execute;
          exit;
        ...
        end select;
      end loop;
      executeDone := FALSE;
      idleWorkers := 3;
      waitPhaseWorkers := 0;
      loop
        select ...
        or accept GetResult(resultItem :
          out ResultType) do
            Remove(GEN1CollectionInfo.results,
                   resultItem);
        end GetResult;
        ...
        end select;
        if idleWorkers = 3 and workCount=0 then
          executeDone := TRUE;
        end if;
        exit when waitPhaseWorkers = 3;
      end loop;
      ...
      GEN1CollectionInfo.done := TRUE;
    end loop Outer;
  end GEN1ActivePool;
```

Figure 4.3: Original and specialized RWF implementation of task type `ActivePool`

be resolved at specialization time yielding the corresponding entity static name (line 10 of the specialized code).

Partial evaluation also can unfold (unroll) both for-loops because they are now bound to known value of `numWorkers`. In the specialized code we can see that the loop is completely computed away and dynamic allocations of `ActiveWorker` at line 10 from the original code yields three new static objects of `GEN1ActiveWorker`, `GEN2ActiveWorker`, and `GEN3ActiveWorker`.

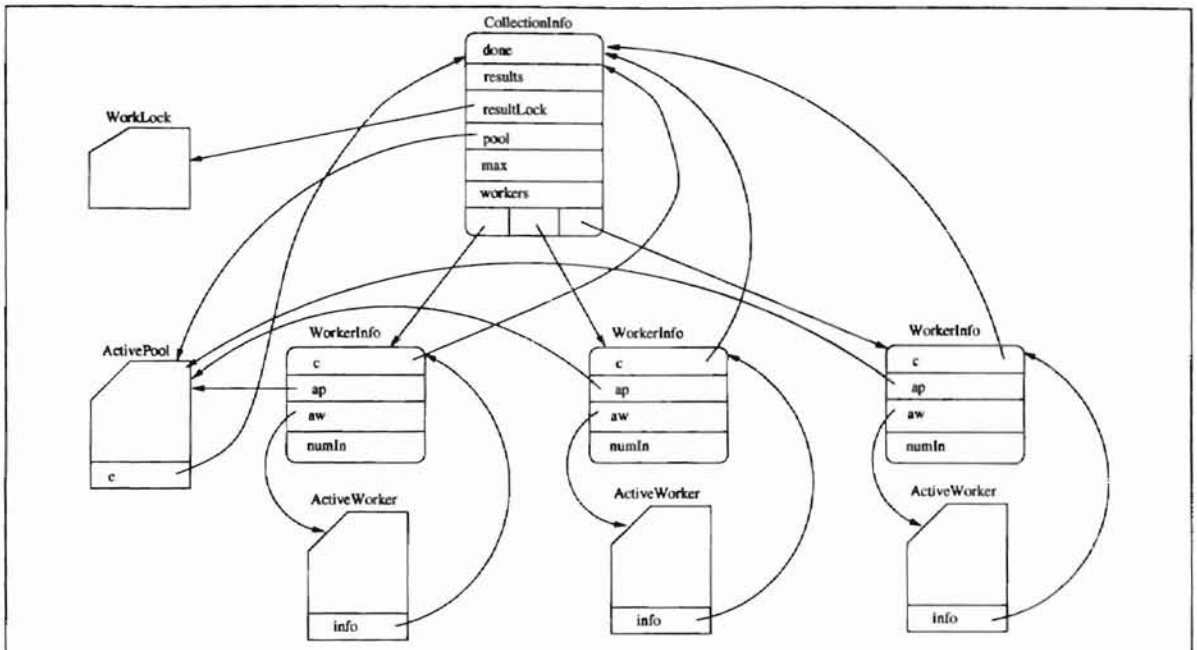


Figure 4.4: Heap structure of RWF for three workers

The code at the right side of figure 4.3 shows the specialized code of `ActivePool` task body related to `numWorkers` = 3 where all dynamic allocated object references have been resolved with their corresponding static names.

4.2 Goal

The goal of this thesis is to illustrate that transformation that we mentioned previously can be made automatically. To this end, we build a prototype for a simple imperative language that is able to perform essential aspects of the configurable system specialization. The essential aspects of specializing the configurable system above are:

- moving dynamically allocated tasks into compile-time objects,
- chaining through known pointers,
- indexing into known records, and
- indexing into known arrays.

```
    -- original code fragment
 1: function Create(numWorkers, numIn : Natural)
 2:   return Collection;
 3:   newCol : Collection;
 4: begin
 5:   newCol := new CollectionInfo(numWorkers);
 6:   Create(newCol.results);
 7:   newCol.done := TRUE;
 8:   newCol.pool := new ActivePool(newCol);
 9:   for i in 1..numWorkers loop
10:     newCol.Workers(i) := new WorkerInfo;
11:     newCol.Workers(i).c := newCol;
12:     newCol.Workers(i).aw :=
13:       new ActiveWorker(newCol.Workers(i));
14:     newCol.Workers(i).ap := newCol.pool;
15:     newCol.Workers(i).numIn := numIn;
16:   end loop;
17:   newCol.pool.StartUp;
18:   for i in 1..numWorkers loop
19:     newCol.Workers(i).aw.Startup;
20:   end loop;
21:   ...
22: end Create;
```

```
-- specialized code fragment
procedure GEN1Create(numIn : Natural) is
begin
  Create(GEN1CollectionInfo.results);
  GEN1CollectionInfo.done := TRUE;

  GEN1WorkerInfo.numIn := numIn;
  GEN2WorkerInfo.numIn := numIn;
  GEN3WorkerInfo.numIn := numIn;

  GEN1ActivePool.StartUp;

  GEN1ActiveWorker.Startup;
  GEN2ActiveWorker.Startup;
  GEN3ActiveWorker.Startup;
  ...
end GEN1Create;
```

Figure 4.5: Original and specialized RWF implementation of function `Create`

Not all programs can be specialized this way, but many configurable systems can. The idea is that one takes some appropriate configuration information (for example, number of workers for the RWF) then creates a specialized instance of the system where all dynamism that gets in the way of verifying the desired properties has been removed.

For some configurable system, dynamism can't be removed using this approach. If a system is allocating, deallocating, and reorganizing its dynamic data structure as it is being executed, then the specialization is not going to be effective. With this approach, to be able to specialize a configurable system, the system has to retain its dynamically created data structure.

CHAPTER 5

INTERPRETER

To verify that our ideas for applying PE to FSV are feasible, we have implemented several Scheme prototypes of the system. Specifically, we have constructed an interpreter for a flow-chart language (FCL) [26] with natural number, access, array and record data types. We also have constructed an offline specializer for a variant of the flow-chart language which is going to be discussed in chapter 6.

The FCL is a small and a simple language. The simplicity of the language makes it appropriate for studying and assessing the feasibility of our approach to the problem. We believe that these language features sufficiently illustrate most of the interesting properties of data manipulation that we will have to address in our problems.

In this chapter, we explain the syntax of the FCL, the type checking and the evaluation of the interpreter.

5.1 Syntax of the FCL

Figure 5.1 presents an example of an FCL program that computes the power function. The program input parameters are `m` and `n` and the result of computing the nth power of `m` (m^n) is held in `result`.

An FCL program is essentially a list of one or more basic-blocks with a list of declarations and a list of input parameters. In turn, the declarations list contains a list of type declarations [1] and a list of variable declarations.

Each basic-block is started with a label followed by a possibly empty list of assignment statements and it is concluded with a jump that transfers control from that block to another block (*i.e.* `goto` or `if`) or that terminates the program via `return`.

[1] Refer to figure 6.46 for an example containing a list of type declarations

```
(((()
  ((result : nat)
   (m : nat)
   (n : nat)))
 (m n)
 ((start ((result := 1))
         (goto test))
  (test ()
        (if (< n 1) done loop))
  (loop ((result := (* result m))
         (n := (- n 1)))
        (goto test))
  (done ()
        (return result))))
```

Figure 5.1: FCL program for power function

The basic aspect of computation in FCL programs is *transformation of computer memory* (computed by assignment statements in each basic-block) and *control transfers* (computed by a jump at the end of each basic-block).

5.1.1 Formal definition of FCL syntax

Figure 5.2 presents the formal definition of the FCL syntax.

At the top of the figure, we can see the syntactic categories of the FCL (*e.g.* program, basic-block, asssignment, *etc.*). The syntactic categories are written using the sans serif font (*e.g.* Program, Block, Assignment, *etc.*).

We assume that an FCL program p is well-formed in the sense that every label used in a jump in p appears as a label of a block.

Definition 5.1 *An FCL program p is well-formed if:*

- (goto l) $\in$ Jump *implies that* $l \in p's$ Block-label, *and*
- (if $e\ l_1\ l_2$) $\in$ Jump *implies that* $l_1, l_2 \in p's$ Block-label.

5.2 Type checking

A program that can be executed by the interpreter has to follow both syntactic and semantic conventions of the FCL. To ensure that certain kinds of programming errors can be

Syntax domains

p	$\in$	Program	a	$\in$	Assignment
f	$\in$	Type-definition	le	$\in$	Left-expression
d	$\in$	Variable-declaration	e	$\in$	Expression
x	$\in$	Variable	op	$\in$	Operation
t	$\in$	Type	n	$\in$	$\mathbb{N}$
b	$\in$	Block	j	$\in$	Jump
l	$\in$	Block-label			

Grammar

p	::=	$(((f^*)(d^*))\ (x^*)\ (b^+))$
f	::=	(record x (d^+))
d	::=	$(x : t)$
t	::=	nat
	|	(access t)
	|	(array n n t)
	|	(record x)
b	::=	$(l\ (a^*)\ j)$
a	::=	$(le := e)$
le	::=	x
	|	(deref e)
	|	([] e e)
	|	(recmem e x)
e	::=	n
	|	x
	|	(op e e)
	|	(deref e)
	|	(new t)
	|	(nil t)
	|	([] e e)
	|	(recmem e x)
j	::=	(goto l)
	|	(if e l l)
	|	(return e)
op	::=	+ | - | * | / | = | > | gte | < | lte | neq
n	::=	0 | 1 | 2 | ...

Figure 5.2: FCL syntax

detected, the interpreter performs *type checking* to check for type errors before the program is executed.

Performing type checking before run-time is called *static* type checking to distinguish it from *dynamic* type checking that performs type checking during program execution or at run-time [36].

5.2.1 Type

As figure 5.2 shows, types that are allowed in the FCL are:

- basic type nat is a natural number ($\mathbb{N}$), whose values are zero and positive whole numbers,

- types constructed using following constructors:
    - `access` to construct a pointer type which is a type that contains a memory address (location) or a constant `nil`. A variable that belong to this type is initialized automatically with `nil` value,
    - `array` to construct an array type which is a collection of variables of the same type that can be indexed and passed around as an entity,
    - `record` to construct a record type which is a type that consists of one or more logically related variables, and like array type, it is also can be passed around as an entity. To define a new record variable, we have to declare its type in the type definitions list. Only after that we can declare the record variable.

5.2.2 Type compatibility

What does it mean for two types to be equivalent? There are two methods to define types compatibility, *name equivalence* and *structural equivalence* [36].

In the name equivalence, two types are compatible if they have the same user-defined type name, while in the structural equivalence, two types are compatible if they have the same basic structure.

We use both of these methods in our prototypes. Specifically, name equivalence is used to define type compatibility for record type and structural equivalence is used to define type compatibility for nat, access and array types. This is similar to what is done in Ada and C.

5.2.3 Type checking domains and values

The type checking rules domains and values are presented in figure 5.3

- A semantic value $g \in \mathsf{Type\text{-}tag} = \mathsf{Nat} + \mathsf{Access} + \mathsf{Array} + \mathsf{Record}$ is one of the following:
    - A $\mathsf{Nat} = \{\cdot\}$ is a type-tag for nat type.

g	$\in$	Type-tag	$=$	Nat + Access + Array + Record
		Nat	$=$	$\{\cdot\}$
		Access	$=$	Type-tag
		Array	$=$	Type-tag $\times$ $\mathbb{N}$ $\times$ $\mathbb{N}$
		Record	$=$	Variable
η	$\in$	Type-header	$\subseteq$	$\wp$(Variable)
$\bar{\tau}$	$\in$	Type-environment	$=$	Variable $\rightarrow$ Record-type-environment
ρ	$\in$	Record-type-environment	$=$	Variable $\rightarrow$ Type-tag
$\bar{\varepsilon}$	$\in$	Variable-environment	$=$	Variable $\rightarrow$ Type-tag

Figure 5.3: Type checking domains and values

- An Access = Type-tag is a type-tag for access type. If the access type-tag is access(g), then g is the type-tag of the access component.
- An Array = Type-tag $\times$ $\mathbb{N}$ $\times$ $\mathbb{N}$ is a type-tag for array type. If the array type-tag is array(g, n_1, n_2), then g is the type-tag of the array members, n_1 is the array lower bound and n_2 is the array upper bound.
- A Record = Variable is a type-tag for record type. If the record type-tag is record(x), then x is the record type name.
- operator + performs *disjoint union* (sum) operation which is a form of union ($\cup$) operation that preserves each members domain of origin [35], using tags or labels.

- A *type name list* $\eta \in$ Type-header $\subseteq$ $\wp$(Variable) [2] is a list of all record type names that are declared in the program type definitions list and these names are gathered at the type checking pre-phase. This list is used to check for the existence of a record type, so we are able to declare circular (recursive) type definitions.
- A *type environment* $\bar{\tau} \in$ Type-environment is a partial function from Variable to $\rho \in$ Record-type-environment, while Record-type-environment itself will map Variable to their Type-tag. Intuitively, the type environment $\bar{\tau}$ for a program p will include all record types that are declared in p's type definitions list.

[2] $\wp(T)$ is the symbol for powerset of T

$$\frac{\bar{\tau}, \rho, \eta \vdash^{tc}_{decls} (d^+) : \rho' \quad \text{where } \rho = \text{empty Record-type-environment} \qquad x \notin Domain(\bar{\tau})}{\bar{\tau}, \eta \vdash^{tc}_{def} (\texttt{record } x\ (d^+)) : \bar{\tau}[x \mapsto \rho']}$$

$$\frac{}{\bar{\tau}, \eta \vdash^{tc}_{defs} () : \bar{\tau}} \qquad \frac{\bar{\tau}, \eta \vdash^{tc}_{def} f : \bar{\tau}' \qquad \bar{\tau}', \eta \vdash^{tc}_{defs} (f^*) : \bar{\tau}''}{\bar{\tau}, \eta \vdash^{tc}_{defs} (f\ f^*) : \bar{\tau}''}$$

Figure 5.4: Type checking rules for type definitions

- An *environment* $\bar{\varepsilon} \in$ Variable-environment is a partial function mapping all variables $\in$ Variable that are declared in the list of variable declarations to their Type-tag.

5.2.4 Type checking of type definitions

The type checking of type definitions is defined as a transition relation from Type-environment, Type-header, and Type-definition to Type-environment, as follows:

$\bar{\tau}, \eta \vdash^{tc}_{def} f : \bar{\tau}'$

and this relation is defined in figure 5.4.

The first rule evaluates a single type definition. The intuition behind the rule is as follows. First, we type check all variable declarations in the record using the type checking rules for variable declarations in figure 5.5. These evaluations begin with an empty table $\in$ Record-type-environment. Then, we check for the non-existence of the record name in the table η domain. Finally, we update the type environment $\bar{\tau}$ for this record with the final modification of the record type environment ρ'.

The second rule explains the meaning of type checking an empty type definition.

The third rule says that type checking a sequence of type definitions is actually type checking the first type definition and type checking the rest.

5.2.5 Type checking of variable declarations

The type checking of variable declarations is defined as a transition relation from Type-environment, Variable-environment, Type-header, and Variable-declaration to

$$\frac{\bar{\tau}, \eta \vdash^{tc}_{type} t : g \qquad x \notin Domain(\bar{\varepsilon})}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{decl} (x : t) : \bar{\varepsilon}[x \mapsto g]}$$

$$\frac{}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{decls} () : \bar{\varepsilon}} \qquad \frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{decl} d : \bar{\varepsilon}' \qquad \bar{\tau}, \bar{\varepsilon}', \eta \vdash^{tc}_{decls} (d^*) : \bar{\varepsilon}''}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{decls} (d\ d^*) : \bar{\varepsilon}''}$$

Figure 5.5: Type checking rules for variable declarations

Variable-environment, as follows:

$$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{decl} d : \bar{\varepsilon}'$$

and figure 5.5 defines this relation.

The first rule evaluates a single variable declaration by first type checking the type t to get its type-tag g. Then, the environment $\bar{\varepsilon}$ for the variable is updated this type-tag g.

The second and the third rules define the type checking rules for an empty variable declaration and a sequence of variable declarations. These rules are similar to the corresponding type checking rules for type definitions in the previous subsection.

5.2.6 Type checking of types

Definition 5.2 *Let t be a type, $\bar{\tau}$ be a type environment, and η be a type list defined for all type names in the type definitions list. Then, the type t is type correct if there exists a type-tag $g \in$* Type-tag *such that:*

$$\bar{\tau}, \eta \vdash^{tc}_{type} t : g$$

where the relation $\vdash^{tc}_{type}$ is defined in figure 5.6.

The type checking rule for nat type is straightforward.

The type checking rules for access and array types are quite similar. First, we have to type check the type t yielding the type-tag g. Then, we return the appropriate type-tag for each rule with the type-tag g as its component.

For record types, we have to check for the existence of the record name x in the type

Nat type

$$\frac{}{\bar{\tau}, \eta \vdash^{tc}_{type} \mathbf{nat} : \mathsf{nat}}$$

Access type

$$\frac{\bar{\tau}, \eta \vdash^{tc}_{type} t : g}{\bar{\tau}, \eta \vdash^{tc}_{type} (\mathbf{access}\ t) : \mathsf{access}(g)}$$

Array type

$$\frac{\bar{\tau}, \eta \vdash^{tc}_{type} t : g}{\bar{\tau}, \eta \vdash^{tc}_{type} (\mathbf{array}\ t\ n_1\ n_2) : \mathsf{array}(g, n_1, n_2)}$$

Record type

$$\frac{x \in \eta}{\bar{\tau}, \eta \vdash^{tc}_{type} (\mathbf{record}\ x) : \mathsf{record}(x)}$$

Figure 5.6: Type checking rules for types

name list η. The rule returns a record type-tag with the name x as its component.

5.2.7 Type checking of expressions

Definition 5.3 *Let e be a left-expression or a (right) expression, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, $\bar{\varepsilon}$ be an environment defined for all variable names in the variable declarations list, and η be a type list defined for all type names in the type definition list. Then, the expression e is type correct if there exists a type-tag $g \in$ Type-tag such that:*

$$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e : g$$

where the relation $\vdash^{tc}_{exp}$ is defined in figure 5.7.

The first rule for expressions is the type checking rule for a natural number n. The rule returns a nat type-tag $\in$ Nat.

A variable reference type checking returns a type-tag associated with the variable in the environment $\bar{\varepsilon}$.

The type checking rule for binary operation expressions is evaluated as follows. First, we type check expressions e_1 and e_2 yielding their type-tags g_1 and g_2 respectively. Then, we map both expression type-tags using operation types equivalence rules in figure 5.8 to get the result type-tag g.

Constant

$$\frac{}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} n : \mathsf{nat}}$$

Variable reference

$$\frac{}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} x : \bar{\varepsilon}(x)}$$

Binary operation

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e_1 : g_1 \quad \bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e_2 : g_2 \quad op \vdash^{tc}_{op-equiv} \langle g_1, g_2 \rangle \rightarrow g}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} (op\ e_1\ e_2) : g}$$

Access dereference

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e : \mathsf{access}(g)}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} (\mathtt{deref}\ e) : g}$$

New

$$\frac{\bar{\tau}, \eta \vdash^{tc}_{type} t : g}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} (\mathtt{new}\ t) : \mathsf{access}(g)}$$

Nil

$$\frac{\bar{\tau}, \eta \vdash^{tc}_{type} t : g}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} (\mathtt{nil}\ t) : \mathsf{access}(g)}$$

Array indexing

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e_1 : \mathsf{array}(g, n_1, n_2) \quad \bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e_2 : \mathsf{nat}}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} ([]\ e_1\ e_2) : g}$$

Record indexing

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e : \mathsf{record}(x')}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} (\mathtt{recmem}\ e\ x) : \bar{\tau}(x')(x)}$$

Figure 5.7: Type checking rules for expressions

The type checking rule for access dereference expressions says that type checking the expression e should return an access type-tag with the type-tag g as its component. The result of the overall expression is the type-tag g.

Type checking rules for new and nil expressions return an access type-tag with g, the result of the type t type checking, as the component.

To type check an array indexing expression, we type check the first expression e_1 yielding an array type-tag and the second expression e_2 yielding a nat type-tag. The result of type checking the whole expression is g, a type-tag extracted from the access type-tag.

The final rule for expressions type checking is the rule for record indexing expressions. The rule evaluated as follow. First, we type check the expression e yielding a record type-tag. Then, we lookup the type environment $\bar{\tau}$ for the record x' which return a local record type environment. Finally, we use the local table to extract the type-tag of the record

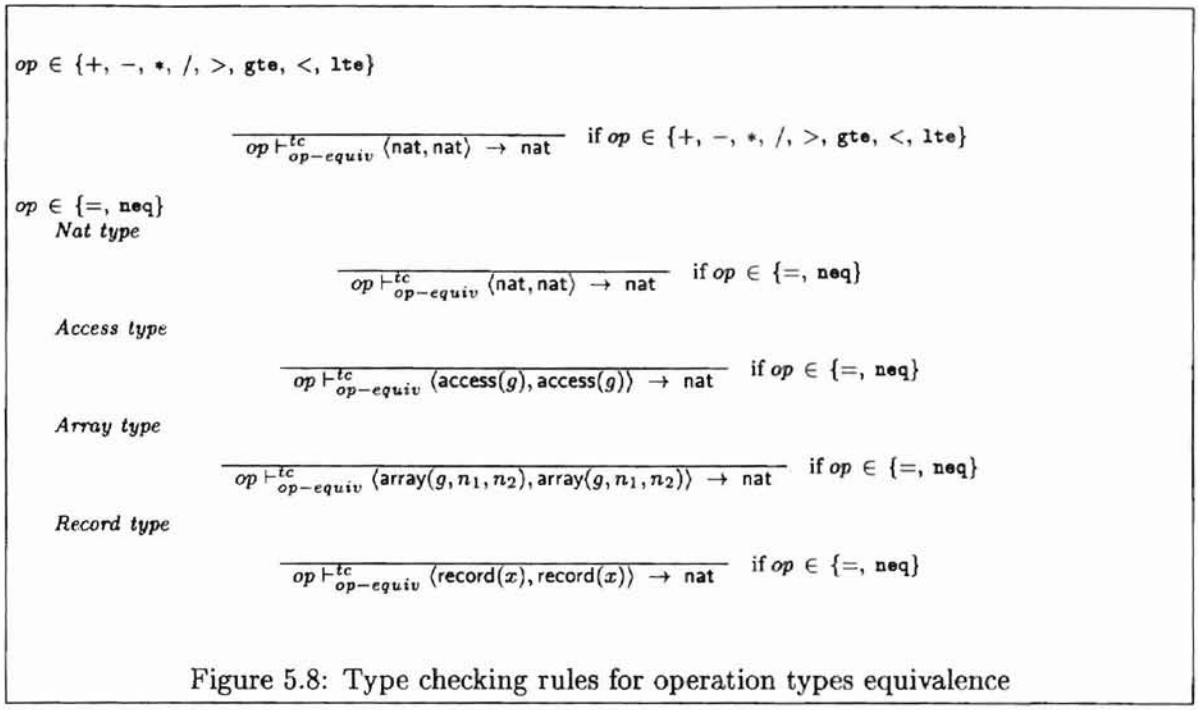


Figure 5.8: Type checking rules for operation types equivalence

member x.

5.2.8 Type checking of operation types equivalence

Definition 5.4 *Let op be an operator, g_1 and g_2 be type-tags. Then, the operation op between g_1 and g_2 is type correct if there exists a $g \in$ Type-tag such that:*

$$op \vdash^{tc}_{op-equiv} \langle g_1, g_1 \rangle \rightarrow g$$

where the relation $\vdash^{tc}_{op-equiv}$ is defined in figure 5.8.

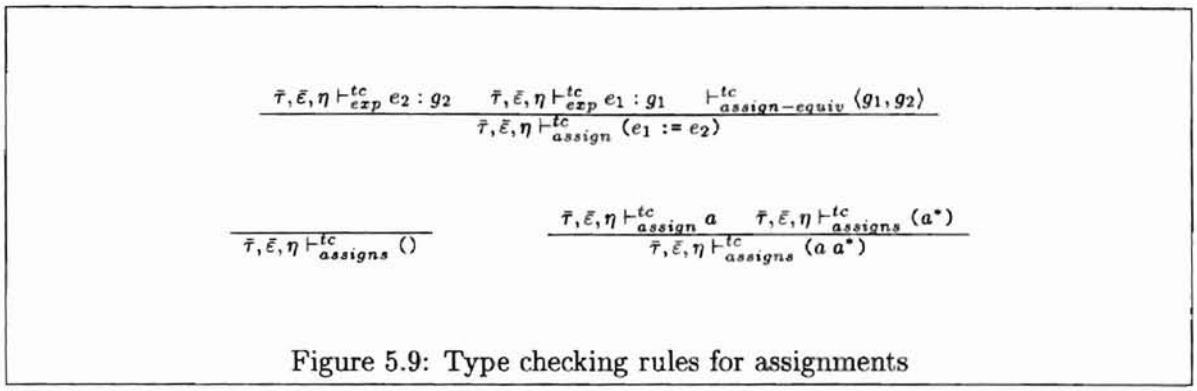
The first rule is an operation types equivalence rule for an operation $op \in \{+, -, *, /, >, \texttt{gte}, <, \texttt{lte}\}$. The operation is type correct if g_1 and g_2 are nat type-tags and it returns a nat type-tag.

The rest of the rules are the type checking rules for operation $op \in \{=, \texttt{neq}\}$.

The second rule is textually equal with the previous rule.

The third rule is defined if type-tags g_1 and g_2 are access type-tags. Implicitly, the rule says that the operation is type correct if both type-tags components are equivalent.

The fourth rule has similar style with the third rule. The operation is type correct if



$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e_2 : g_2 \quad \bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e_1 : g_1 \quad \vdash^{tc}_{assign-equiv} \langle g_1, g_2 \rangle}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{assign} (e_1 := e_2)}$$

$$\frac{}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{assigns} ()} \qquad \frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{assign} a \quad \bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{assigns} (a^*)}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{assigns} (a\ a^*)}$$

Figure 5.9: Type checking rules for assignments

both g_1 and g_2 are array type-tags and components g, n_1 and n_2 from both type-tags are equivalent.

The last rule says that the operation is type correct if both type-tags are record type-tags and they have a same name.

5.2.9 Type checking of assignments

Definition 5.5 *Let $a \equiv (e_1 := e_2)$ be an assignment in a block, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, $\bar{\varepsilon}$ be an environment defined for all variable names in the variable declarations list, and η be a type list defined for all type names in the type definitions list. Then, the assignment a is type correct iff:*

$$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{assign} a$$

where the relation $\vdash^{tc}_{assign}$ is defined in figure 5.9.

The first rule is defined for a single assignment. The rule is evaluated as follow. First, we type check both expressions e_2 and e_1 yielding type-tags g_2 and g_1 respectively. Then, we apply these type-tags into type checking rules for assignment types equivalence from figure 5.10.

The second and the third rules show the type checking rules for an empty assignment and a sequence of assignments.

Nat type

$$\frac{}{\vdash^{tc}_{assign-equiv} \langle \mathsf{nat}, \mathsf{nat} \rangle}$$

Access type

$$\frac{}{\vdash^{tc}_{assign-equiv} \langle \mathsf{access}(g), \mathsf{access}(g) \rangle}$$

Array type

$$\frac{}{\vdash^{tc}_{assign-equiv} \langle \mathsf{array}(g, n_1, n_2), \mathsf{array}(g, n_1, n_2) \rangle}$$

Record type

$$\frac{}{\vdash^{tc}_{assign-equiv} \langle \mathsf{record}(x), \mathsf{record}(x) \rangle}$$

Figure 5.10: Type checking rules for assignment types equivalence

5.2.10 Type checking of assignment types equivalence

Definition 5.6 *Let g_1 and g_2 be type-tags. Then, the assignment between g_1 and g_2 is type correct iff:*

$$\vdash^{tc}_{assign-equiv} \langle g_1, g_1 \rangle$$

where the relation $\vdash^{tc}_{assign-equiv}$ is defined in figure 5.10.

The rules in figure 5.10 have a similar justification as the operation types equivalence rules in figure 5.8. The difference between them is these rules do not return anything as the result of the evaluation like in the operation types equivalence rules.

5.2.11 Type checking of jumps

Definition 5.7 *Let j be a jump in a block, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, $\bar{\varepsilon}$ be an environment defined for all variable names in the variable declarations list, and η be a type list defined for all type names in the type definitions list. Then, the jump j is type correct iff:*

$$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{jump} j$$

where the relation $\vdash^{tc}_{jump}$ is defined in figure 5.11.

A goto jump is always type correct.

Goto

$$\frac{}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{jump} (\texttt{goto}\ l)}$$

If

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e : \mathsf{nat}}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{jump} (\texttt{if}\ e\ l_1\ l_2)}$$

Return

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{exp} e : \mathsf{nat}}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{jump} (\texttt{return}\ e)}$$

Figure 5.11: Type checking rules for jumps

The second rule is the type checking rule for if jumps. The rule says that an if jump is type correct if the expression e is type correct where the expression e is type correct and returns a nat type-tag.

The type checking rule for return jumps is similar to the previous type checking rule for if jumps.

5.2.12 Type checking of blocks

Definition 5.8 *Let $b \equiv (l\ (a^*)\ j)$ be a block in a program, $\bar{\tau}$ is a type environment defined for all type names in the type definitions list, $\bar{\varepsilon}$ be an environment defined for all variable names in the variable declarations list, and η be a type list defined for all type names in the type definitions list. Then, the block b is type correct iff:*

$$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{tc}_{block} b$$

where the relation $\vdash^{tc}_{block}$ is defined in figure 5.12.

The first rule is defined for a single block. The rule is evaluated as follow. First, we type check the assignments list (a^*) using the type checking rules for assignments. Then, we type check the jump j using type checking rules for jumps. The block is type correct if both evaluations are type correct.

The second and the third rules define type checking rules for an empty block and a sequence of blocks.

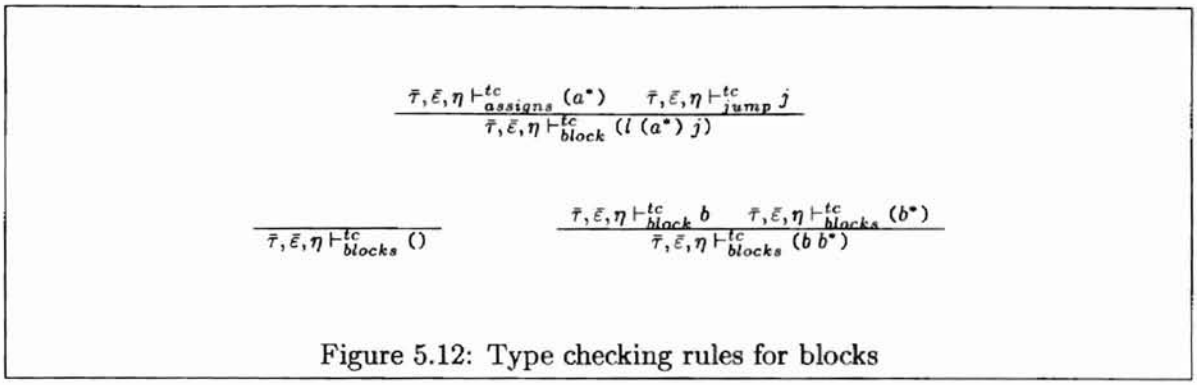


Figure 5.12: Type checking rules for blocks

5.2.13 Type checking of programs

Definition 5.9 *Let* $p \equiv (((f^*)(d^*))\ (x^*)\ (b^+))$ *be a program and* η *be a type list defined for all type names in the type definition list. Then, the program* p *is type correct iff:*

$$\eta \vdash^{tc}_{prog} p$$

where the relation $\vdash^{tc}_{prog}$ *is defined in figure 5.13.*

The rule says that to type check a program, we have to type check the program's type definitions, variable declarations, and blocks. The program is type correct if all three evaluations are type correct.

5.3 Evaluation of the FCL interpreter

In this section, we will formalize the evaluation rules for the interpreter. The rules are defined in term of execution traces using operational semantics.

5.3.1 Execution traces

An execution trace shows the steps a program makes between computational states. In the FCL, a computational state consists of a label indicating the current basic-block and the current value of the store. For example, the following is an execution trace of the power program in figure 5.1 computing 4^2 :

$$\begin{aligned} & (\texttt{start}, [\texttt{m} \mapsto 4, \texttt{n} \mapsto 2, \texttt{result} \mapsto 0]) \\ \rightarrow\ & (\texttt{test}, [\texttt{m} \mapsto 4, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) \end{aligned}$$

$$\frac{\bar{\tau},\eta \vdash^{tc}_{defs} (f^*) : \bar{\tau}' \quad \bar{\tau}',\bar{\varepsilon},\eta \vdash^{tc}_{decls} (d^*) : \bar{\varepsilon}' \quad \bar{\tau}',\bar{\varepsilon}',\eta \vdash^{tc}_{blocks} (b^+)}{\eta \vdash^{tc}_{prog} (((f^*)(d^*))\ (x^*)\ (b^+))}$$

where $\bar{\tau}$ = empty Type-environment
$\bar{\varepsilon}$ = empty Variable-environment

Figure 5.13: Type checking rule for programs

$$\begin{aligned}
&\rightarrow (\texttt{loop}, [\texttt{m} \mapsto 4, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) \\
&\rightarrow (\texttt{test}, [\texttt{m} \mapsto 4, \texttt{n} \mapsto 1, \texttt{result} \mapsto 4]) \\
&\rightarrow (\texttt{loop}, [\texttt{m} \mapsto 4, \texttt{n} \mapsto 1, \texttt{result} \mapsto 4]) \\
&\rightarrow (\texttt{test}, [\texttt{m} \mapsto 4, \texttt{n} \mapsto 0, \texttt{result} \mapsto 16]) \\
&\rightarrow (\texttt{done}, [\texttt{m} \mapsto 4, \texttt{n} \mapsto 0, \texttt{result} \mapsto 16]) \\
&\rightarrow (\mathsf{halt}(16), [\texttt{m} \mapsto 4, \texttt{n} \mapsto 0, \texttt{result} \mapsto 16])
\end{aligned}$$

Here we introduce a special label $\mathsf{halt}(n)$ not found in the original program. This special label labels the final program state where n is the program return value.

5.3.2 Evaluation domains and values

Domains and values of the evaluation rules are presented in figure 5.14.

- A semantic value $v \in \mathsf{Value} = \mathsf{Nat} + \mathsf{Loc} + \mathsf{Record} + \mathsf{Array}$ is one of the following:
  - An $n \in \mathsf{Nat} = \mathbb{N}$ is a natural number.
  - An $h \in \mathsf{Loc} = \mathbb{N} \cup \{nil\}$ is a memory location where nil is a special location means undefined locations.
  - An $\alpha \in \mathsf{Array\text{-}map} = \mathbb{N} \rightarrow \mathsf{Loc}$ is a partial function mapping an array index $\in \mathbb{N}$ to its memory location $\in$ Loc. The array value in $\mathsf{Array} = \mathsf{Array\text{-}map} \times \mathbb{N} \times \mathbb{N}$ consist of an array map, and the upper and lower bound of the array index set.
  - A $\rho \in \mathsf{Record} = \mathsf{Variable} \rightarrow \mathsf{Loc}$ is a record value which is a mapping from an identifier $\in$ Variable to its memory location $\in$ Loc.

v	$\in$	Value	$=$	Nat + Loc + Array + Record
n	$\in$	Nat	$=$	$\mathbb{N}$
h	$\in$	Loc	$=$	$\mathbb{N} \cup \{nil\}$
		Array	$=$	Array-map $\times$ $\mathbb{N}$ $\times$ $\mathbb{N}$
α	$\in$	Array-map	$=$	$\mathbb{N} \rightarrow$ Loc
ρ	$\in$	Record	$=$	Variable $\rightarrow$ Loc
l	$\in$	Label	$=$	Block-label $\cup$ halt($\mathbb{N}$)
$\bar{\tau}$	$\in$	Type-environment	$=$	Variable $\rightarrow$ Record-type-environment
		Record-type-environment	$=$	Variable $\rightarrow$ Type
ε	$\in$	Variable-environment	$=$	Variable $\rightarrow$ Loc
σ	$\in$	Store	$=$	Loc $\rightarrow$ Value
Γ	$\in$	Block-map	$=$	Block-label $\rightarrow$ Block

Figure 5.14: Evaluation rules domains and values

- A semantic value $l \in$ Label $=$ Block-label $\cup$ halt($\mathbb{N}$).

- A *type environment* $\bar{\tau} \in$ Type-environment is a partial function from Variable to Record-type-environment, while Record-type-environment itself will map a variable to its type. The type environment we use for evaluation is actually the same type environment we use for type checking.

- An *environment* $\varepsilon \in$ Variable-environment is a partial function mapping all variables $\in$ Variable that are declared in the list of variable declarations to their locations $\in$ Loc in the store σ.

- A *store* $\sigma \in$ Store is a partial function from locations $\in$ Loc in the environment ε to their values $\in$ Value. A store σ will hold both statically and dynamically allocated objects. For a program p, a store σ will be defined for all static variables declared in p's variable declarations list and dynamic objects created during program p execution.

5.3.3 Evaluation of variable declarations

The evaluation of variable declarations is defined as a transition relation from Type-environment, Variable-environment, Store, Loc, and Variable-declaration to Variable-environment, Store, and Loc, as follows:

$$\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{decls} d \Rightarrow \langle \varepsilon', \sigma', h' \rangle$$

$$\frac{\bar{\tau}, \sigma, h' \vdash^{eval}_{init} t \Rightarrow \langle v, \sigma', h'' \rangle}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{decl} (x : t) \Rightarrow \langle \varepsilon[x \mapsto h], \sigma'[h \mapsto v], h'' \rangle} \quad \text{where } h' = succ(h)$$

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{decls} () \Rightarrow \langle \varepsilon, \sigma, h \rangle} \qquad \frac{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{decl} d \Rightarrow \langle \varepsilon', \sigma', h' \rangle \quad \bar{\tau}, \varepsilon', \sigma', h' \vdash^{eval}_{decls} (d^*) \Rightarrow \langle \varepsilon'', \sigma'', h'' \rangle}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{decls} (d\, d^*) \Rightarrow \langle \varepsilon'', \sigma'', h'' \rangle}$$

Figure 5.15: Evaluation rules for variable declarations

$$\frac{\bar{\tau}, \sigma, h' \vdash^{eval}_{init} t \Rightarrow \langle v, \sigma', h'' \rangle}{\bar{\tau}, \alpha, \sigma, h \vdash^{eval}_{array-init} \langle n, t \rangle \Rightarrow \langle \alpha[n \mapsto h], \sigma'[h \mapsto v], h'' \rangle} \quad \text{where } h' = succ(h)$$

Figure 5.16: Evaluation rules for array members initialization

and this relation is defined in figure 5.15.

The first rule is the evaluation rule for a single variable declaration. The rule is justified as follow. First, the environment ε corresponding to the current location h is modified to map the variable name introduced by the variable declaration. Then, the current next available location h is updated with a new next available location h' (by invoking function *succ*). Finally, the store σ is modified to map the current next available location h with the type t initial value v.

The second and the third rules define how to evaluate an empty variable declaration and a sequence of variable declarations.

5.3.4 Evaluation of array members initialization

The evaluation of array members initialization is defined as a transition relation from Type-environment, Array-map, Store, Loc, $\mathbb{N}$, and t to Array-map, Store, and Loc, as follows:

$$\bar{\tau}, \alpha, \sigma, h \vdash^{eval}_{array-init} \langle n, t \rangle \Rightarrow \langle \alpha', \sigma', h' \rangle$$

and this relation is defined in figure 5.16.

The evaluation rule for array members initialization is a rule used to initialized each

Nat type

$$\frac{}{\bar{\tau}, \sigma, h \vdash^{eval}_{init} \textbf{nat} \Rightarrow \langle \mathsf{nat}(0), \sigma, h \rangle}$$

Access type

$$\frac{}{\bar{\tau}, \sigma, h \vdash^{eval}_{init} (\textbf{access } t) \Rightarrow \langle \mathsf{loc}(nil), \sigma, h \rangle}$$

Array type

$$\frac{\begin{array}{l}\forall i \in \{n_1, \ldots, n_2\} \;.\; \bar{\tau}, \alpha_{i-1}, \sigma_{i-1}, h_{i-1} \vdash^{eval}_{array-init} \langle i, t \rangle \Rightarrow \langle \alpha_i, \sigma_i, h_i \rangle \\ \quad \text{where} \quad \alpha_{n_1-1} = \text{empty Array-map} \\ \qquad\qquad \sigma_{n_1-1} = \sigma \\ \qquad\qquad h_{n_1-1} = h\end{array}}{\bar{\tau}, \sigma, h \vdash^{eval}_{init} (\textbf{array } n_1\ n_2\ t) \Rightarrow \langle \mathsf{array}(\alpha_{n_2}, n_1, n_2), \sigma_{n_2}, h_{n_2} \rangle}$$

Record type

$$\frac{\bar{\tau}(x) = \begin{bmatrix} x_1 \mapsto t_1 \\ \vdots \\ x_n \mapsto t_n \end{bmatrix} \qquad \begin{array}{l}\forall i \in \{1, \ldots, n\} \;.\; \bar{\tau}, \rho_{i-1}, \sigma_{i-1}, h_{i-1} \vdash^{eval}_{decl} (x_i : t_i) \Rightarrow \langle \rho_i, \sigma_i, h_i \rangle \\ \quad \text{where} \quad \rho_0 = \text{empty Record} \\ \qquad\qquad \sigma_0 = \sigma \\ \qquad\qquad h_0 = h\end{array}}{\bar{\tau}, \sigma, h \vdash^{eval}_{init} (\textbf{record } x) \Rightarrow \langle \mathsf{record}(\rho_n), \sigma_n, h_n \rangle}$$

Figure 5.17: Evaluation rules for initializations

array member base on the index. The rule is justified similarly to the evaluation rule for a single variable declaration.

5.3.5 Evaluation of initializations

Definition 5.10 *Let t be a type, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, σ be the current store, and h be the next available location. Then, the initialization of t yields a value v, a new store σ', and a new next available location h' iff:*

$$\bar{\tau}, \sigma, h \vdash^{eval}_{init} t \Rightarrow \langle v, \sigma', h' \rangle$$

where the relation $\vdash^{eval}_{init}$ is defined in figure 5.17.

A nat type is initialized with $\mathsf{nat}(0) \in \mathsf{Nat}$.

An access type is initialized with $\mathsf{loc}(nil) \in \mathsf{Loc}$.

The initialization value for array types is $\mathsf{array}(\alpha, n_1, n_2) \in \mathsf{Array}$, where $\alpha \in \mathsf{Array\text{-}map}$ is defined for all index $n \in \{n_1, \ldots, n_2\}$ and is modified by repeatedly invoking the evaluation rule for array members initialization in figure 5.16, n_1 is the array lower bound and n_2 is the array upper bound.

Variable reference

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{lexp} x \Rightarrow \langle \varepsilon(x), \sigma, h \rangle}$$

Access dereference

$$\frac{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} e \Rightarrow \langle \mathsf{loc}(h_e), \sigma', h' \rangle \qquad h_e \neq nil}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} (\mathbf{deref}\ e) \Rightarrow \langle h_e, \sigma', h' \rangle}$$

Array indexing

$$\frac{\begin{array}{l}\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} e_1 \Rightarrow \langle \mathsf{array}(\alpha, n_1, n_2), \sigma', h' \rangle \\ \bar{\tau}, \varepsilon, \sigma', h' \vdash^{eval}_{exp} e_2 \Rightarrow \langle \mathsf{nat}(n), \sigma'', h'' \rangle \qquad n_1 \leq n \leq n_2\end{array}}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{lexp} ([]\ e_1\ e_2) \Rightarrow \langle \alpha(n), \sigma'', h'' \rangle}$$

Record indexing

$$\frac{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} e \Rightarrow \langle \mathsf{record}(\rho), \sigma', h' \rangle}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{lexp} (\mathbf{recmem}\ e\ x) \Rightarrow \langle \rho(x), \sigma', h' \rangle}$$

Figure 5.18: Evaluation rules for left-expressions

The initialization rule for record types has similar style with the initialization rule for an array type, except that this rule uses the rule in figure 5.15 instead of the rule in figure 5.16. It returns $\mathsf{record}(\rho) \in \mathsf{Record}$, where ρ is defined for all variable $x_i \in \{x_1, \ldots, x_n\}$.

5.3.6 Evaluation of left-expressions

Definition 5.11 *Let le be a left-expression (left hand side expression) in an assignment, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, ε be an environment defined for all variable names in the variable declarations list, σ be the current store, and h be the next available location. Then, the meaning of the left-expression le is a memory location h_l, a new store σ', and a new next available location h' iff:*

$$\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{lexp} le \Rightarrow \langle h_l, \sigma', h' \rangle$$

where the relation $\vdash^{eval}_{lexp}$ is defined in figure 5.18.

The evaluation of a variable returns a location associated with the variable in the environment ε.

Evaluating an access dereference left-expression yields a location $h_e \neq nil$ extracted from a value $\in \mathsf{Loc}$ where the value itself is the result of evaluating the expression e.

To evaluate an array indexing left expression, we have to evaluate both expressions e_1

Constant

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} n \Rightarrow \langle \mathsf{nat}(n), \sigma, h \rangle}$$

Variable reference

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} x \Rightarrow \langle \sigma(\varepsilon(x)), \sigma, h \rangle}$$

Binary operation

$$\frac{\begin{array}{l}\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} e_1 \Rightarrow \langle v_1, \sigma', h' \rangle \\ \bar{\tau}, \varepsilon, \sigma', h' \vdash^{eval}_{exp} e_2 \Rightarrow \langle v_2, \sigma'', h'' \rangle \qquad \sigma'', op \vdash^{eval}_{op} (v_1\ ,\ v_2) \rightarrow v\end{array}}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} (op\ e_1\ e_2) \Rightarrow \langle v, \sigma'', h'' \rangle}$$

Access dereference

$$\frac{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} e \Rightarrow \langle \mathsf{loc}(h_e), \sigma', h' \rangle \qquad h_e \neq nil}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} (\mathtt{deref}\ e) \Rightarrow \langle \sigma'(h_e), \sigma', h' \rangle}$$

New

$$\frac{\bar{\tau}, \sigma, h' \vdash^{eval}_{init} t \Rightarrow \langle v, \sigma', h'' \rangle}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} (\mathtt{new}\ t) \Rightarrow \langle \mathsf{loc}(h), \sigma'[h \mapsto v], h'' \rangle} \quad \text{where } h' = succ(h)$$

Nil

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} (\mathtt{nil}\ t) \Rightarrow \langle \mathsf{loc}(nil), \sigma, h \rangle}$$

Array indexing

$$\frac{\begin{array}{l}\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} e_1 \Rightarrow \langle \mathsf{array}(\alpha, n_1, n_2), \sigma', h' \rangle \\ \bar{\tau}, \varepsilon, \sigma', h' \vdash^{eval}_{exp} e_2 \Rightarrow \langle \mathsf{nat}(n), \sigma'', h'' \rangle \qquad n_1 \leq n \leq n_2\end{array}}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} ([]\ e_1\ e_2) \Rightarrow \langle \sigma''(\alpha(n)), \sigma'', h'' \rangle}$$

Record indexing

$$\frac{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} e \Rightarrow \langle \mathsf{record}(\rho), \sigma', h' \rangle}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} (\mathtt{recmem}\ e\ x) \Rightarrow \langle \sigma'(\rho(x)), \sigma', h' \rangle}$$

Figure 5.19: Evaluation rules for expressions

and e_2. The first expression e_1 yields an array value and the second expression e_2 evaluates to a nat value. The result of evaluating the overall left-expression is a location extracted from α for index n where the index n has to be within the array boundary.

The last rule explains the meaning of evaluating a record indexing left-expression. First, the expression e in the record indexing left-expression should evaluate to a record value. Then, we apply table lookup from the relation ρ for the variable x.

5.3.7 Evaluation of expressions

Definition 5.12 *Let e be an expression, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, ε be an environment defined for all variable names in the variable declarations list, σ be the current store, and h be the next available location. Then, the meaning of the expression e is a value v, a new store σ', and a new next available location h'*

Operation evaluation for $op \in \{+, -, *, /\}$

$$\frac{}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{nat}(n_1)\ ,\ \mathsf{nat}(n_2)\rangle \rightarrow \mathsf{nat}([\![op]\!]\ n_1\ n_2)} \quad \text{if } op \in \{+, -, *, /\}$$

Operation evaluation for $op \in \{>, \texttt{gte}, <, \texttt{lte}\}$

$$\frac{\textit{true-value}([\![op]\!]\ n_1\ n_2)}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{nat}(n_1)\ ,\ \mathsf{nat}(n_2)\rangle \rightarrow \mathsf{nat}(1)} \quad \text{if } op \in \{>, \texttt{gte}, <, \texttt{lte}\}$$

$$\frac{\textit{false-value}([\![op]\!]\ n_1\ n_2)}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{nat}(n_1)\ ,\ \mathsf{nat}(n_2)\rangle \rightarrow \mathsf{nat}(0)} \quad \text{if } op \in \{>, \texttt{gte}, <, \texttt{lte}\}$$

Figure 5.20: Evaluation rules for operations *(part 1)*

iff:

$$\bar{\tau}, \varepsilon, \sigma, h \vdash_{exp}^{eval} e \Rightarrow \langle v, \sigma', h' \rangle$$

where the relation $\vdash_{exp}^{eval}$ *is defined in figure 5.19.*

The result of evaluating a constant n is a nat value.

A variable reference expression is evaluated similarly to the left-expression one. Except that the location returned in the left-expression rules is used to extract a value from the store σ. The rule returns the value.

The rule for binary operations is evaluated as follow. First, both expression e_1 and e_2 are evaluated to their values v_1 and v_2 respectively. Then, we use the evaluation rules for operations to map the current store, the operator op, the value v_1 and the value v_2 to the operation result value v.

The evaluation of an access dereference expression yields a value extracted from the store σ for a location $h_e \neq nil$, where the location h_e is the component of the expression e evaluation result.

A new expression returns a value of current next available location h. The rule modifies the store for location h with the type t initial value and updates the current next available location h with a new next available location h'.

The result of a nil expression is the location `nil`.

Operation evaluation for $op \in \{=, \texttt{neq}\}$

Nat

$$\frac{}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{nat}(n)\ ,\ \mathsf{nat}(n)\rangle \rightarrow \mathsf{nat}(1)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{nat}(n_1)\ ,\ \mathsf{nat}(n_2)\rangle \rightarrow \mathsf{nat}(0)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

Loc

$$\frac{}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{loc}(h)\ ,\ \mathsf{loc}(h)\rangle \rightarrow \mathsf{nat}(1)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{loc}(h_1)\ ,\ \mathsf{loc}(h_2)\rangle \rightarrow \mathsf{nat}(0)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

Array

$$\frac{\forall i \in \{n_1, \ldots, n_2\}\ .\ \sigma, op \vdash_{op}^{eval} \langle \sigma(\alpha_1(i))\ ,\ \sigma(\alpha_2(i))\rangle \rightarrow \mathsf{nat}(1)}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{array}(\alpha_1, n_1, n_2)\ ,\ \mathsf{array}(\alpha_2, n_1, n_2)\rangle \rightarrow \mathsf{nat}(1)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{\exists i \in \{n_1, \ldots, n_2\}\ .\ \sigma, op \vdash_{op}^{eval} \langle \sigma(\alpha_1(i))\ ,\ \sigma(\alpha_2(i))\rangle \rightarrow \mathsf{nat}(0)}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{array}(\alpha_1, n_1, n_2)\ ,\ \mathsf{array}(\alpha_2, n_1, n_2)\rangle \rightarrow \mathsf{nat}(0)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

Record

$$\frac{\forall i \in \{1, \ldots, n\}\ .\ \sigma, op \vdash_{op}^{eval} \langle \sigma(\rho_1(x_i))\ ,\ \sigma(\rho_2(x_i))\rangle \rightarrow \mathsf{nat}(1)}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{record}(\rho_1)\ ,\ \mathsf{record}(\rho_2)\rangle \rightarrow \mathsf{nat}(1)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{\exists i \in \{1, \ldots, n\}\ .\ \sigma, op \vdash_{op}^{eval} \langle \sigma(\rho_1(x_i))\ ,\ \sigma(\rho_2(x_i))\rangle \rightarrow \mathsf{nat}(0)}{\sigma, op \vdash_{op}^{eval} \langle \mathsf{record}(\rho_1)\ ,\ \mathsf{record}(\rho_2)\rangle \rightarrow \mathsf{nat}(0)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

Figure 5.21: Evaluation rules for operations (*part 2)*

The evaluation rules for array indexing expressions and record indexing expressions are quite similar to their corresponding left-expressions rules. The difference is these rules return a value extracted from the store σ for a location that being returned in their expressions evaluation rule counterparts.

5.3.8 Evaluation of operations

Definition 5.13 *Let op be an operator,* v_1 *and* v_2 *be values, and* σ *be the current store. Then, the meaning of operation op between* v_1 *and* v_2 *is a value v iff:*

$$\sigma, op \vdash_{op}^{eval} \langle v_1\ ,\ v_2\rangle \rightarrow v$$

where the relation $\vdash_{op}^{eval}$ *is defined in figure 5.20 and figure 5.21.*

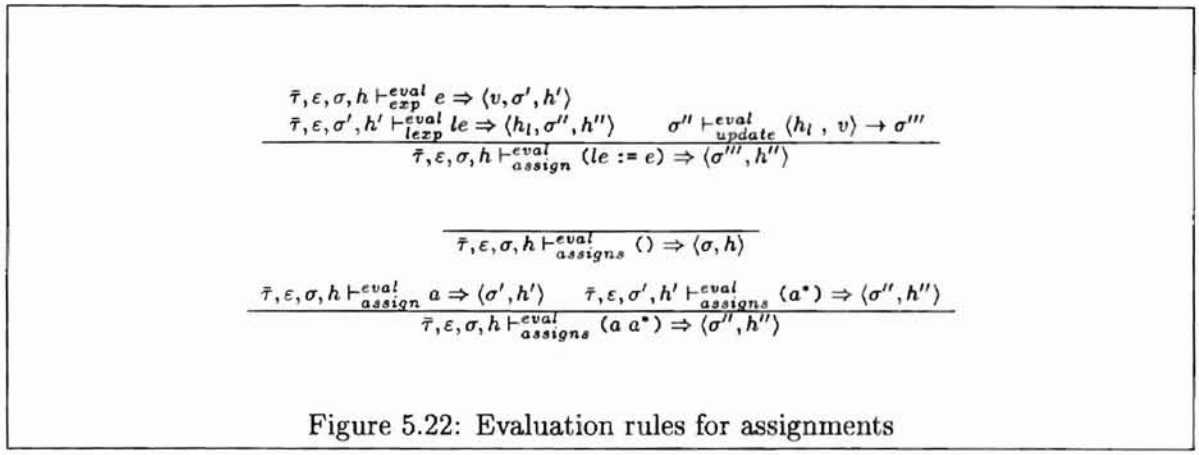


Figure 5.22: Evaluation rules for assignments

The first rule defines the meaning of an operation where operator $op \in \{+, -, *, /\}$. The rule says that the operation is only allowed if v_1 and v_2 are nat values and it returns the nat value of executing the *op* operation between n_1 and n_2.

The second and the third rules are the meaning of an operation where operator $op \in \{>, \texttt{gte}, <, \texttt{lte}\}$. Both rules are also defined only if v_1 and v_2 are nat values. The second rule says if the result of invoking *true-value* function with the result of executing *op* operation between n_1 and n_2 is true, then return nat(1) or else return nat(0).

The rules in figure 5.21 are the meaning of operation if operator $op \in \{=, \texttt{neq}\}$.

The first two rules are the case where $v_1, v_2 \in$ Nat or $v_1, v_2 \in$ Loc. In this case, we check for values component equality. If they are equal, then return nat(1) or else return nat(0).

For array and record, the operation evaluation is allowed only if both values v_1 and v_2 are compatible. To check values compatibility, we evaluate array or record values members. If all members evaluation return nat(1), then the result of the overall evaluation is nat(1) or else return nat(0).

5.3.9 Evaluation of assignments

Definition 5.14 *Let $a \equiv (le := e)$ be an assignment in a block, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, ε be an environment defined for all variable names in the variable declarations list, σ be the current store, and h be the next*

Nat type

$$\frac{}{\sigma \vdash^{eval}_{update} \langle h\ ,\ \mathsf{nat}(n)\rangle \rightarrow \sigma[h \mapsto \mathsf{nat}(n)]}$$

Access type

$$\frac{}{\sigma \vdash^{eval}_{update} \langle h\ ,\ \mathsf{loc}(h')\rangle \rightarrow \sigma[h \mapsto \mathsf{loc}(h')]}$$

Array type

$$\frac{\sigma(h) = \mathsf{array}(\alpha_h, n_1, n_2) \qquad \begin{array}{c}\forall i \in \{n_1, \ldots, n_2\}\ .\ \sigma_{i-1} \vdash^{eval}_{update} \langle \alpha_h(i)\ ,\ \sigma_{i-1}(\alpha(i))\rangle \rightarrow \sigma_i \\ \text{where } \sigma_{n_1-1} = \sigma\end{array}}{\sigma \vdash^{eval}_{update} \langle h\ ,\ \mathsf{array}(\alpha, n_1, n_2)\rangle \rightarrow \sigma_{n_2}}$$

Record type

$$\frac{\sigma(h) = \mathsf{record}(\rho_h) \qquad \begin{array}{c}\forall i \in \{1, \ldots, n\}\ .\ \sigma_{i-1} \vdash^{eval}_{update} \langle \rho_h(x_i)\ ,\ \sigma_{i-1}(\rho(x_i))\rangle \rightarrow \sigma_i \\ \text{where } \sigma_0 = \sigma\end{array}}{\sigma \vdash^{eval}_{update} \langle h\ ,\ \mathsf{record}(\rho)\rangle \rightarrow \sigma_n}$$

Figure 5.23: Evaluation rules for assignment-updates

available location. Then, the meaning of the assignment a is a new store σ' and a new next available location h' iff:

$$\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{assign} a \Rightarrow \langle \sigma', h' \rangle$$

where the relation $\vdash^{eval}_{assign}$ is defined in figure 5.22.

In the first rule, we define the meaning of a single assignment. The rule is evaluated as follow. First, we evaluate expressions e and le yielding a value v and a location h_l respectively. Then, we apply assignment-updates rules from figure 5.23 to modify the current store.

The second and the third rules define how to evaluate an empty assignment and a sequence of assignments.

5.3.10 Evaluation of assignment-updates

Definition 5.15 *Let h be a location, v be a value, and σ be the current store. Then, the meaning of the assignment-update for h and v is a new store σ' iff:*

$$\sigma \vdash^{eval}_{update} \langle h\ ,\ v\rangle \rightarrow \sigma'$$

where the relation $\vdash^{eval}_{update}$ is defined in figure 5.23.

Goto

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{jump} (\textbf{goto } l) \Rightarrow \langle l, \sigma, h \rangle}$$

If

$$\frac{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} e \Rightarrow \langle \textsf{nat}(n), \sigma', h' \rangle \qquad \textit{true-value}(n)}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{jump} (\textbf{if } e\ l_1\ l_2) \Rightarrow \langle l_1, \sigma', h' \rangle}$$

$$\frac{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} e \Rightarrow \langle \textsf{nat}(n), \sigma', h' \rangle \qquad \textit{false-value}(n)}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{jump} (\textbf{if } e\ l_1\ l_2) \Rightarrow \langle l_2, \sigma', h' \rangle}$$

Return

$$\frac{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{exp} e \Rightarrow \langle \textsf{nat}(n), \sigma', h' \rangle}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{jump} (\textbf{return } e) \Rightarrow \langle \textsf{halt}(n), \sigma', h' \rangle}$$

Figure 5.24: Evaluation rules for jumps

The first and the second rules cover the case where $v \in \textsf{Nat}$ and $v \in \textsf{Loc}$ respectively. These rules are straightforward.

In the case $v \in \textsf{Array}$ and $v \in \textsf{Record}$, the rules involved recursive calls to update all members in the array or in the record. The rules modify the current store σ to a new store.

5.3.11 Evaluation of jumps

Definition 5.16 *Let j be a jump in a block, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, ε be an environment defined for all variable names in the variable declarations list, σ be the current store, and h be the next available location. Then, the meaning of the jump j is a label l, a new store σ', and a new next available location h' iff:*

$$\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{jump} j \Rightarrow \langle l, \sigma', h' \rangle$$

where the relation $\vdash^{eval}_{jump}$ is defined in figure 5.24.

A goto jump evaluation returns the jump target label l.

An if jump is evaluated by evaluating the expression e. The result of the expression evaluation is a nat value which define which label to extract (by invoking *true-value* or *false-value* functions). If the function evaluation is true, then extract the label l_1 (label for the true branch), but if it is false, then extract the label l_2 (label for the false branch).

Block

$$\frac{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{assigns} (a^*) \Rightarrow \langle \sigma', h' \rangle \quad \bar{\tau}, \varepsilon, \sigma', h' \vdash^{eval}_{jump} j \Rightarrow \langle l', \sigma'', h'' \rangle}{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{block} (l\ (a^*)\ j) \Rightarrow \langle l', \sigma'', h'' \rangle}$$

Blocks transition

$$\frac{\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{block} \Gamma(l) \Rightarrow \langle l', \sigma', h' \rangle}{\vdash_\Gamma \langle l, \sigma, h \rangle \rightarrow \langle l', \sigma', h' \rangle}$$

Figure 5.25: Evaluation rules for blocks

A return jump evaluation returns the special label halt with the result of evaluating the expression e as its component.

5.3.12 Evaluation of blocks

Definition 5.17 *Let $b \equiv (l\ (a^*)\ j)$ be a block in a program, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, ε be an environment defined for all variable names in the variable declarations list, σ be the current store, and h be the next available location. Then, the meaning of the block b is a label l, a new store σ', and a new next available location h' iff:*

$$\bar{\tau}, \varepsilon, \sigma, h \vdash^{eval}_{block} b \Rightarrow \langle l, \sigma', h' \rangle$$

where the relation $\vdash^{eval}_{block}$ is defined in figure 5.25.

To evaluate a single block, we evaluate the assignments list (a^*) and the jump j. The result of the block evaluation is the next label to execute.

A *block map* $\Gamma \in$ Block-map in the blocks transition rule is a partial function from Block-label to Block. Intuitively, Γ is a lookup table for blocks using a label as its key. A block map Γ will be defined for all labels occurring in the program being described and undefined otherwise.

The Γ-indexed transition relation of block map is:

$$\rightarrow_\Gamma \subseteq (\mathsf{Label} \times \mathsf{Store} \times \mathsf{Loc}) \times (\mathsf{Label} \times \mathsf{Store} \times \mathsf{Loc})$$

The blocks transition rule is essentially a mapping from one block to the next block using the function Γ.

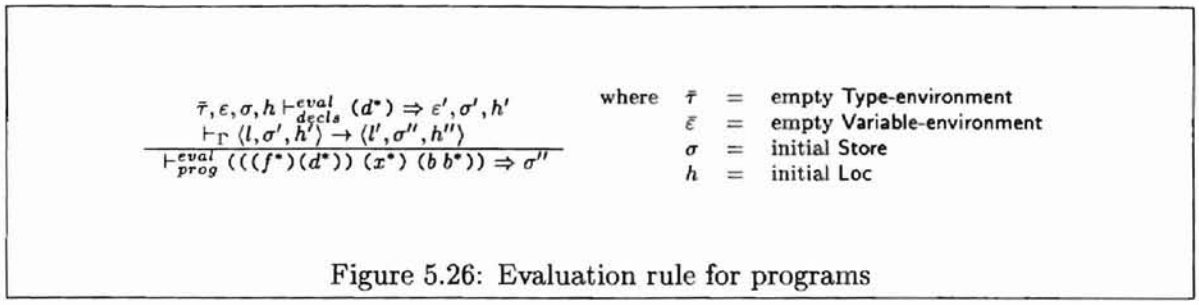


Figure 5.26: Evaluation rule for programs

5.3.13 Evaluation of programs

Definition 5.18 *Let* $p \equiv (((f^*)(d^*))\ (x^*)\ (b^+))$ *be a program. Then, the meaning of the program* p *is a new store* σ' *iff:*

$$\vdash^{eval}_{prog} p \Rightarrow \sigma'$$

where the relation $\vdash^{eval}_{prog}$ *is defined in figure 5.26.*

A program is evaluated as follows. First, we evaluate variable declarations in the variable declarations list using the rule from figure 5.15. Then, using the blocks transition rule, we evaluate blocks in the program, begin with the first block in the program until the the label returned by the rule is halt. Finally, the rule returns the current store σ''.

CHAPTER 6

SPECIALIZATION

This chapter is dedicated to the offline specializer prototype that we built. First, we explain the methodology that should be followed when using the specializer and present the formal definition of the specializer input.

Then, we review the notion of two-level language [33] presented in chapter 2, explain why this framework is insufficient for our application domain, and propose a more general framework capable of handling our application.

We redefine the notion of binding-time type [33] and develop a binding-time type checking system that is used to check for the consistency of user-supplied annotations and input program constructs.

Finally, we present the specification of the specialization process that actually carries out the transformations illustrated in chapter 4, and conclude with examples of the specializer transformation on the fragment of the replicated workers framework (RWF) program from chapter 4.

6.1 Methodology

As we mentioned in chapter 4, the goal of this work is to specialize the configurable system using partial evaluation technology. To be able to perform the specialization, users have to supply some configuration information to the system (*e.g.* number of workers in the RWF example). Ideally, using *binding-time inference* rules, the system with the initial information should be able to classify automatically each construct in a program as eliminable or residual.

Currently, in our system, users still have to manually mark all types occurring in the

```
((()
  ((result : (nat D R))
   (m : (nat D R))
   (n : (nat S E))))
 (m n)
 ((start ((result := 1))
         (goto test))
  (test ()
        (if (< n 1) done loop))
  (loop ((result := (* result m))
         (n := (- n 1)))
        (goto test))
  (done ()
        (return result))))
```

Figure 6.1: The annotated FCL power function program

program to guide the specialization process. Then, the subsequent binding-time type checking phase checks if these annotations are consistent, and propagates this initial classification information throughout the program.

As a consequence, an annotated source program is created for the original source program, where each type in the program is attached with annotations (marked). For example, in figure 6.1, we can see the corresponding annotated program for the power function illustrated in figure 5.1. In conjunction with this idea, we develop an FCL language with annotations (FCL-ann) where the formal definition of this language can be seen in figure 6.2.

Based on the information from the binding-time type checking, users may want to take in additional configuration information (*e.g.* to improve specialization). Finally, if users are satisfied with the degree of the specialization, then actual configuration parameter values are supplied and the actual specialization is carried out.

We can see steps to be followed when using this system application in figure 6.3

6.2 Extended binding-time annotations

We would like to follow the standard practice for offline partial evaluation and use the two-level language framework for specifying program construct binding-times [33]. However, this turns out to be insufficient for our purposes, so we have to extend the standard binding-time annotations and the two-level language.

Syntax domains

p	$\in$	Program	l	$\in$	Block-label
f	$\in$	Type-definition	a	$\in$	Assignment
d	$\in$	Variable-declaration	le	$\in$	Left-expression
x	$\in$	Variable	e	$\in$	Expression
T	$\in$	Type-ann	op	$\in$	Operation
c	$\in$	Bt-value	n	$\in$	$\mathbb{N}$
m	$\in$	Bt-mode	j	$\in$	Jump
b	$\in$	Block			

Grammar

p	::=	$(((f^*)(d^*))\ (x^*)\ ((b^+)))$
f	::=	(record x (d^+) c m)
d	::=	$(x : T)$
T	::=	(nat c m)
	|	(access T c m)
	|	(array n n T c m)
	|	(record x)
c	::=	S | D
m	::=	E | R
b	::=	$(l\ (a^*)\ j)$
a	::=	$(le := e)$
le	::=	x
	|	(deref e)
	|	([] e e)
	|	(recmem e x)
e	::=	n
	|	x
	|	$(op\ e\ e)$
	|	(deref e)
	|	(new T c m)
	|	(nil T c m)
	|	([] e e)
	|	(recmem e x)
j	::=	(goto l)
	|	(if e l l)
	|	(return e)
op	::=	+ | − | * | / | = | > | gte | < | lte | neq
n	::=	0 | 1 | 2 | ...

Figure 6.2: FCL-ann syntax

In the standard binding-time annotations, if a construct is static, then it will always be eliminated at the specialization time. If it is dynamic, then it will always be residualized at the specialization time. These constraints cause the specialization to become too conservative, where intuitively we have to annotate a construct as dynamic if it can't be eliminated. For example, consider a record returned by a function. This record has to be annotated as dynamic [1] because it can't be eliminated, even though in someccases the record value is known. With the extended binding-time annotations, we can annotate a construct as static and use its value to improve the specialization but still leave the construct in the residual

[1]Following [1], we will describe later why all records returned from function calls should be dynamic

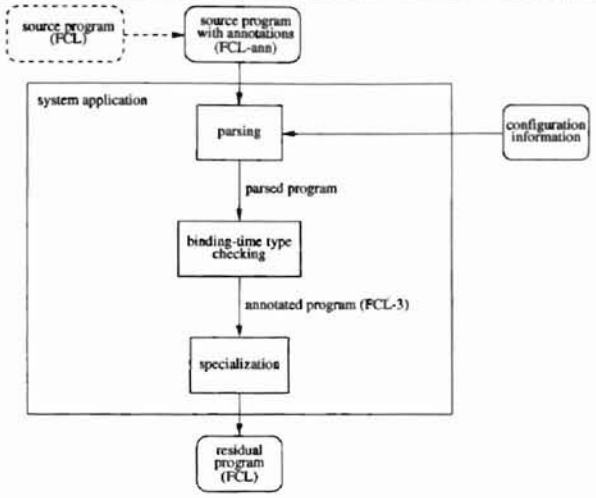


Figure 6.3: The flow of the application specialization process

program.

Also with the standard binding-time annotations, the specialization is limited to only one predefined way. For example, in C-mix, the technique for specializing a static record (eliminable) is to split the record into separate variables. With the extended annotations, a static record can be specialized in two different ways, the record can be split into separate variables or it can be narrowed into a new record structure.

In the extended binding-time annotations, we combine two type of annotations. The first type of annotations is to define *when* a construct can be evaluated to its value (*i.e.* specialization time or run-time). If the construct value can be evaluated at specialization time, then it is annotated as *static* or otherwise *dynamic*. The second type of annotations is to define the construct behavior at specialization time. If the construct is to be eliminated at specialization, then it is annotated as *eliminable* or otherwise *residual*.

By combining these two type of annotations as a tuple (*binding-time tuple*) to describe the binding-time information, we separate explicitly *when* the construct value is known (*binding-time value*) and *how* the construct should be treated at specialization time (*binding-time mode*).

Definition 6.1 *A binding-time tuple (bt-tuple) B is a tuple of $\langle c, m \rangle$, where:*

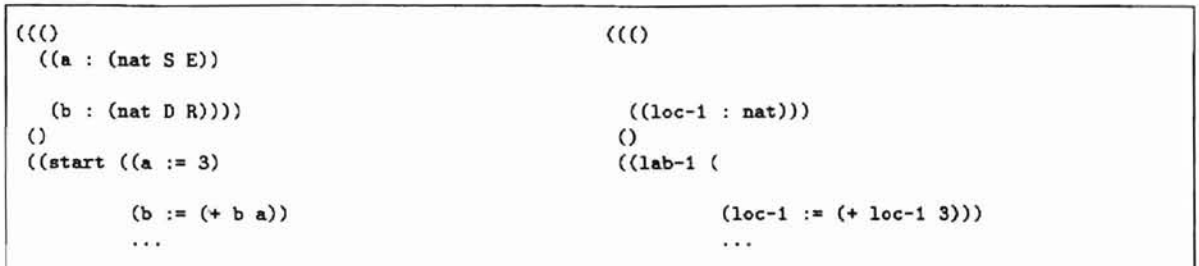


Figure 6.4: Example of source code and specialized code of nat type

- *a binding-time value (bt-value)* $c \in \{\mathtt{S}, \mathtt{D}\}$ *describes when a value of a construct can be determined,*
- *a binding-time mode (bt-mode)* $m \in \{\mathtt{E}, \mathtt{R}\}$ *is describes the construct behavior at specialization time.*

In the example from figure 6.1, we can see that the nat type variable `n` is annotated as static (`S`) and eliminable (`E`), which means that the variable `n` value is known at specialization time and it will be eliminated from the residual code. The nat type variables `m` and `result` are annotated as dynamic (`D`) and residual (`R`). This annotation means that values of variable `m` and `result` are unknown at specialization time and they will be residualized.

The basic idea of the extended binding-time annotations for each type in the FCL language is as follow:

- static and eliminable
  - nat type

    A variable of nat type is classified as static if values of the variables are known at specialization time (all values that the variable depends on are known at specialization time). All occurences of a static nat type variable, along with its declaration, will be eliminated at specialization time. For example, variable `a` at the left side of figure 6.4 will be eliminated in the specialized version of the program at the right side of the figure.
  - array type

```
(((()                                          (((()
  ((a : (array 1 2 (nat D R) S E))               ((loc-2 : nat)
                                                  (loc-3 : nat)

   (b : (array 1 2 (nat D R) D R))                (loc-4 : (array 1 2 nat))

   (m : (nat D R))))                              (loc-5 : nat)))
 (m)                                            (loc-5)
 ((start ((([] a 1) := 1)                       ((lab-1 ((loc-2 := 1)
          (([] a 2) := (+ ([] a 1) m))                   (loc-3 := (+ loc-2 loc-5))

          (([] b 1) := 1)                                (([] loc-4 1) := 1)
          (([] b 2) := (+ ([] b 1) m))                   (([] loc-4 2) := (+ ([] loc-4 1) loc-5))
          ...                                            ...
```

Figure 6.5: Example of source code and specialized code of array type

An array type variable is classified as static if all indexing expressions into the array are known at specialization time. A static and eliminable array with eliminable components will be eliminated at specialization time. If the binding-time mode of the array components is residual, then the array will be split into separate variables. For example, array a of two components at the left side of figure 6.5 will be replaced by two variables (`loc-2` and `loc-3`) at the right side of the figure, one for each component.

- access type

  An access type variable is classified as static if values of the access variable (addresses that it point to) are always known at compile time. A static and eliminable access type variable is eliminated at specialization time. Similar to array type, the decision to eliminate or to residualize the access component that is generated at run-time depends on the component binding-time information. For example, variable a at the left side of figure 6.6 that is allocated dynamically will be replaced with new static variable `loc-2` at the right side of the figure.

- record type

  A record type can be defined as static if members of the record can be accessed at specialization time. A record type that is annotated as static and eliminable is specialized by eliminating the record type definition, while variables of this

```
(((()                                          (((()
  ((a : (access (nat D R) S E))                  ((loc-2 : nat)

   (b : (nat D R))))                              (loc-3 : nat)))
 ()                                             ()
 ((start ((a := (new (nat D R) S E))            ((lab-1 (
          ((deref a) := 4)                              (loc-3 := 4)
          (m := (+ (deref a) 1))                        (loc-2 := (+ loc-3 1)))
          ...                                           ...
```

Figure 6.6: Example of source code and specialized code of access type

record type is split into separate variables where each variable corresponds to a residual component of the record. For example, record type `t1` in the source code at the left side of figure 6.7 will be eliminated in the specialized code at the right side of the figure, while variable `a` in the source code will be split and replaced by array variable `loc-3` in the specialized code, which corresponds to the record `t`'s residual member `y`.

- dynamic and residual

  - nat type

    A dynamic nat type variable will be residualized at specialization time and will be annotated as dynamic and residual. For example, variable `b` at the left side of figure 6.4 will be residualized as `loc-1` in the specialized version of the program at the right side of the figure.

  - array type

    An array type variable that is annotated as dynamic and residual must have dynamic and residual components. At specialization time, the array variable will be residualized. For example, array `b` at the left side of figure 6.5 will be residualized as array `loc-4` at the right side of the figure.

  - access type

    Similar to dynamic and residual array type variables, a dynamic and residual access type variable must point to a dynamic and residual object. The special-

```
((((record t1                                    (((
          ((x : (nat S E))
           (y : (nat D R)))
          S E)
   (record t2                                      (record t2
          ((x : (nat D R))                                ((x : nat)
           (y : (nat D R)))                                (y : nat))
          D R)
   (record t3                                      (record t3
          ((x : (nat S E))                                (
           (y : (nat D R)))                                (y : nat))))
          S R))
  ((a : (record t1))                              ((loc-3 : nat)
   (b : (record t2))                              ((loc-4 : (record t2))
   (c : (record t3))                               (loc-5 : (record t3))
   (m : (nat D R))))                               (loc-6 : nat)))
  (m)                                            (loc-6)
  ((start (((recmem a x) := 2)                   ((lab-1 (
          ((recmem a y) := m)                             (loc-3 := loc-6)

          ((recmem b x) := 2)                             ((recmem loc-4 x) := 2)
          ((recmem b y) := m)                             ((recmem loc-4 y) := loc-6)

          ((recmem c x) := 2)
          ((recmem c y) := m)                             ((recmem loc-5 y) := loc-6)
          ...                                             ...
```

Figure 6.7: Example of source code and specialized code of record type

ization of the access variable is similar to the previous dynamic and residual array type variable.

- record type

  If a record type is annotated as dynamic and residual, then all of its members must also be dynamic and residual. At specialization time, the record type definition and variables of this record type will be residualized. For example, record `t2` and variable `b` at the left side of figure 6.7 will be residualized as record `t2` and variable `loc-4` in the specialized version of the program at the right side of the figure.

- static and residual

  - nat type

    A nat type is never classified as static and residual. To residualize a static nat type variable reference, we use *lifting method* [33] by inserting operator lift. With lifting, we evaluate the static nat expression yielding its value, then we residualize

the literal of the value.

- array type

  A static and residual array type variable will be residualized at specialization time just like its dynamic and residual counterpart. Currently, we are not exploiting the staticness of the array, but we believe that in the future we can use the known information from the array to enhance the specialization. For example, we can compute and eliminate the array indexing expression if it is at the right hand side of an assignment statement.

- access type

  For static and residual access variables, the decision to eliminate or to residualize them depends on their construct type and context of use. If the access construct is a variable declaration, then it will be residualized. If the construct is a dereference expression, then we have to check the construct context of use, if it is used in a static and eliminable context, then it will be computed away by returning its value (using operator get-val), but if it is used in dynamic and residual context, then it will be residualized by returning the code representing the construct (using operator get-exp). If the construct context of use is also static and residual, then we will residualize the construct and in the same time, compute the construct value.

- record type

  If a record type is annotated as static and residual, then the record type should has at least one residual members. At specialization time, this record type definition will be narrowed into a new record which consist only residual members from the original type definition. All of its instance variable declarations will be residualized. For example, record `t3` at the left side of figure 6.7 will be narrowed into a new record with the same name,which consist only the residual member `y`. The record indexing expression to the eliminable member `x` will be eliminated,

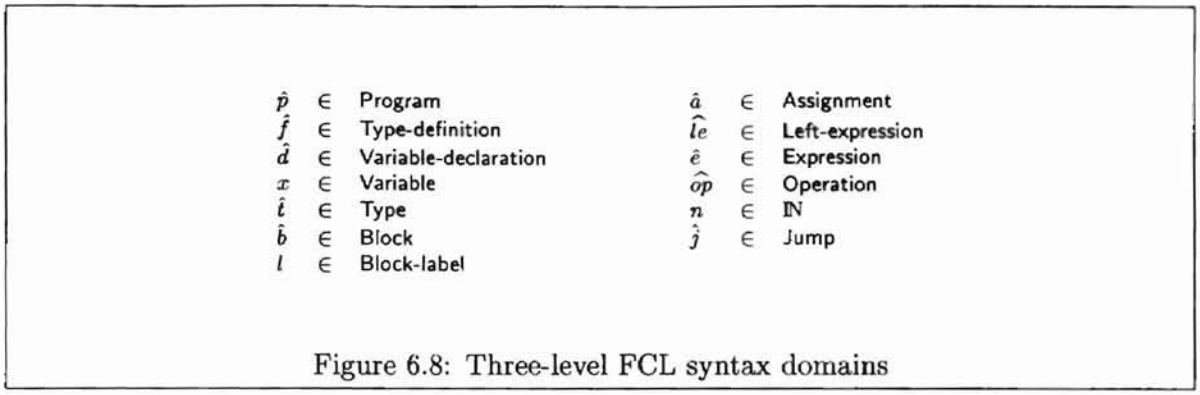


Figure 6.8: Three-level FCL syntax domains

while indexing expression to the residual member y will be residualized.

6.3 Three-level FCL language (FCL-3)

We adopt the idea of two-level language and extend it into a more general framework called *the three-level language*. This language is internal to the system application and is used to specify the result of the binding-time type checking phase (refer to figure 6.3).

Traditionally, in the two-level language, every construct appears in two versions. The *non-underline* to specify eliminable constructs (known) and the *underline* to specify residual constructs (unknown).

In the three-level language framework, every construct will appear in three different versions. The non-underline to specify static and eliminable constructs, the underline to specify dynamic and residual constructs and the *dash-underline* to specify static and residual constructs.

The formal definition of the three-level FCL can be seen in figure 6.8 and figure 6.9. Figure 6.8 defines the language syntax domains and figure 6.9 defines the language grammar.

6.4 Binding-time type

We also adopt the *binding-time type* [1] framework for specifying the binding-time information of a type. In the binding-time type framework, we extend each type to include binding-time information. For example, binding-time type of the variable n in figure 6.1 is

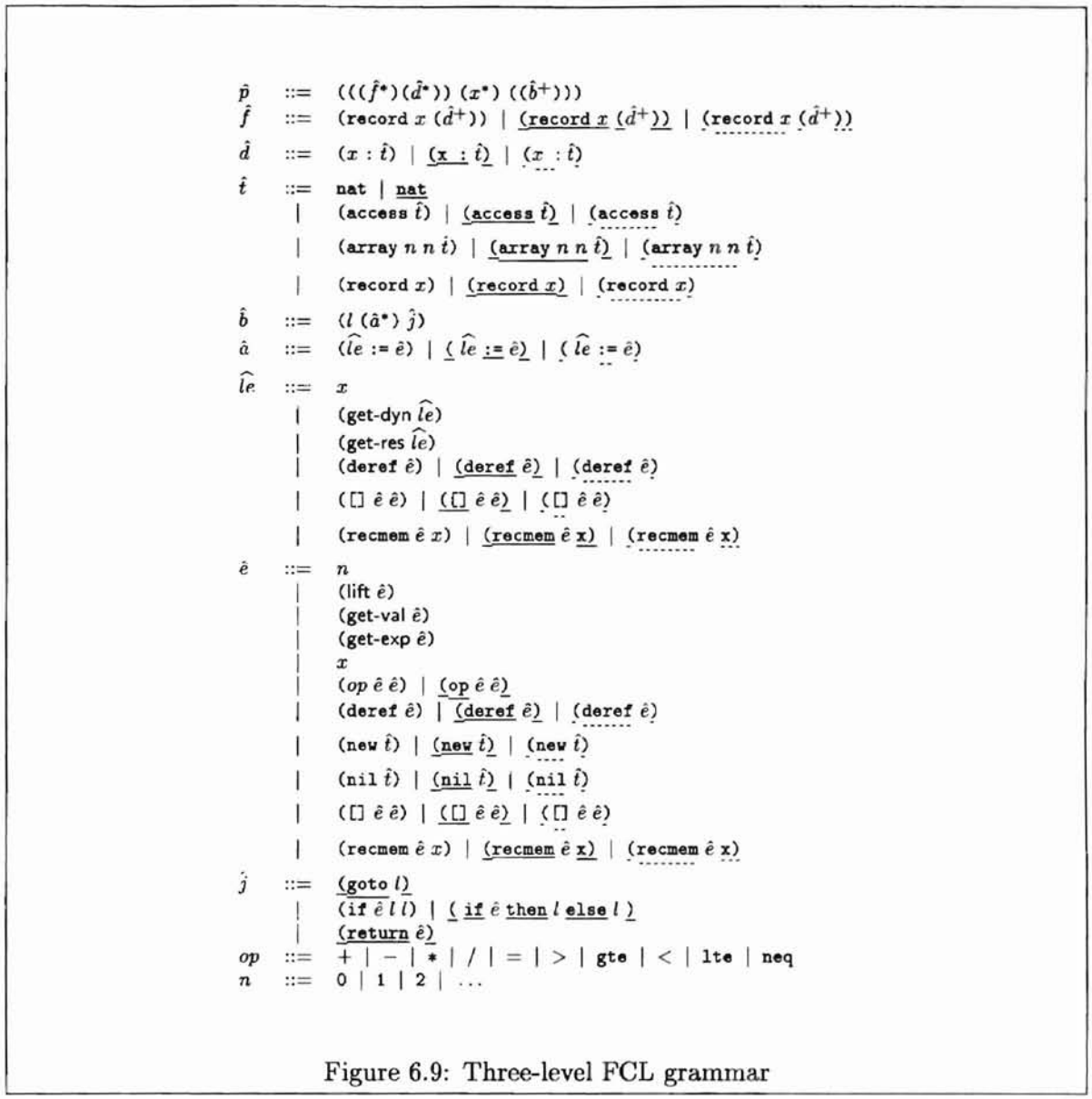


Figure 6.9: Three-level FCL grammar

nat(S,E).

Definition 6.2 *The syntax of binding-time type BT is defined inductively as follow:*

$$
\begin{aligned}
BT \quad ::= \quad & \mathsf{nat}(c, m) \\
\mid \quad & \mathsf{access}(BT, c, m) \\
\mid \quad & \mathsf{array}(BT, n_1, n_2, c, m) \\
\mid \quad & \mathsf{record}(x, c, m)
\end{aligned}
$$

Let BT is a binding-time type. Then, $BT\#b$ denotes the binding-time value of BT and $BT\#m$ denotes the binding-time mode of BT. For example, if $BT = \mathsf{nat}(\mathtt{S},\mathtt{E})$, then $BT\#b = \mathtt{S}$ and $BT\#m = \mathtt{E}$.

As we illustrated in section 6.1, some binding-time types are unreasonable from a computation point of view. For example, it makes no sense to classify a program construct as dynamic and eliminable. The value of dynamic constructs is unknown at compile time, so it is impossible to compute and eliminate them specialization time. This constraint is captured as the first condition in the next definition.

Following constraints identify those binding-time types that we consider to be well-formed.

Definition 6.3 *A bt-type BT is well-formed if it is satisfies the following requirements:*

1. *if $BT\#b = \mathtt{D}$, then $BT\#m \neq \mathtt{E}$.*
2. *if $BT = \mathsf{nat}(c,m)$, then $\langle c,m\rangle \in \{\langle \mathtt{S},\mathtt{E}\rangle, \langle \mathtt{D},\mathtt{R}\rangle\}$,*
3. *if $BT = \mathsf{access}(T,c,m)$ or $BT = \mathsf{array}(T,n_1,n_2,c,m)$ where $BT\#b = \mathtt{D}$ ($c = \mathtt{D}$) and $BT\#m = \mathtt{R}$ ($m = \mathtt{R}$), then $T\#b = \mathtt{D}$, $T\#m = \mathtt{R}$, and T is well-formed,*
4. *if $BT = \mathsf{access}(T,c,m)$ or $BT = \mathsf{array}(T,n_1,n_2,c,m)$ where $BT\#b = \mathtt{S}$ ($c = \mathtt{S}$) and $BT\#m = \mathtt{R}$ ($m = \mathtt{R}$), then $T\#b = \mathtt{S}$ and $T\#m = \mathtt{R}$, or $T\#b = \mathtt{D}$ and $T\#m = \mathtt{R}$, and T is well-formed.*

The second condition says that a nat type construct can be classified only as static and eliminable or dynamic and residual.

The third and fourth conditions are requirements for access and array binding-time types. These conditions say that their binding-time type and their component binding-time type have to be in the order of $\langle \mathtt{S},\mathtt{E}\rangle \sqsubseteq \langle \mathtt{S},\mathtt{R}\rangle \sqsubseteq \langle \mathtt{D},\mathtt{R}\rangle$. For example, if an access type is annotated as dynamic and residual, then so must its dereferenced binding-time type. These

conditions are deduced because it makes no sense to have an unknown (dynamic) pointer to a known (static) object or an unknown array with known members and a residual pointer to eliminable object or a residual array with eliminable members.

6.5 Binding-time type checking

The binding-time type checking takes an FCL-ann program, performs checking to ensure that the input program binding-time types are well-formed and consistent. In addition, an analysis similar to binding-time analysis computes a division classifying all constructs as non-underline, underline or dash-underline where these divisions should be consistent with the initial annotations given by user.

In this section, we give a set of rules to define *well-annotatedness* of the FCL-ann program. This rules form a type checking system. A program that fulfills these rules is said to be well-annotated.

6.5.1 Binding-time type checking domains and values

Figure 6.10 presents domains and values of the binding-time type checking rules.

- A semantic value $BT \in \mathsf{Bt\text{-}type} = \mathsf{Nat} + \mathsf{Access} + \mathsf{Array} + \mathsf{Record}$ is one of the following:
  - A $\mathsf{Nat} = \mathsf{Bt\text{-}value} \times \mathsf{Bt\text{-}mode}$ is a binding-time type tag for nat types. If a nat bt-type is $\mathsf{nat}(c, m)$, then c is the nat binding-time value and m is the nat binding-time mode.
  - An $\mathsf{Access} = \mathsf{Bt\text{-}type} \times \mathsf{Bt\text{-}value} \times \mathsf{Bt\text{-}mode}$ is a binding-time type tag for access types. If an access bt-type is $\mathsf{access}(BT, c, m)$, then c is the access binding-time value, m is the access binding-time mode and BT is the access component binding-time type.
  - An $\mathsf{Array} = \mathsf{Bt\text{-}type} \times \mathbb{N} \times \mathbb{N} \times \mathsf{Bt\text{-}value} \times \mathsf{Bt\text{-}mode}$ is a binding-time type tag for array types. If an array bt-type is $\mathsf{array}(BT, n_1, n_2, c, m)$, then n_1 is the array

BT	$\in$	Bt-type	=	Nat + Access + Array + Record
		Nat	=	Bt-value × Bt-mode
		Access	=	Bt-type × Bt-value × Bt-mode
		Array	=	Bt-type × $\mathbb{N}$ × $\mathbb{N}$ × Bt-value × Bt-mode
		Record	=	Variable × Bt-value × Bt-mode
η	$\in$	Type-header	=	Variable → Bt-value × Bt-mode
ρ	$\in$	Record-type-environment	=	Variable → Bt-type
$\bar{\tau}$	$\in$	Type-environment	=	Variable → Record-type-environment
$\bar{\varepsilon}$	$\in$	Variable-environment	=	Variable → Bt-type

Figure 6.10: Binding-time type checking rules domains and values

lower bound, n_2 is the array upper bound, c is the array binding-time value, m is the array binding-time mode and BT is the array member's binding-time type.

- A Record = Variable × Bt-value × Bt-mode is a binding-time type tag for record types. If a record bt-type is record(x, c, m), then x is the record name, c is the record binding-time value and m is the record binding-time mode.

- A *type name table* $\eta \in$ Type-header is a partial function from a variable $\in$ Variable to its binding-time value $\in$ Bt-value and binding-time mode $\in$ Bt-mode. For the type checking in chapter 5, η was simply a set of record identifiers accumulated before type checking began. Now, we also assume that a pre-phase associates each record identifier with a Bt-value and a Bt-mode. If variable x is not a record identifier, then η is undefined at x.

- A *type environment* $\bar{\tau} \in$ Type-environment is a partial function from a variable $\in$ Variable to its record type environment $\rho \in$ Record-type-environment. The Record-type-environment is a mapping from a variable $\in$ Variable to its binding-time type $\in$ Bt-type.

- An *environment* $\bar{\varepsilon} \in$ Variable-environment is a partial function mapping all variables $\in$ Variable that are declared in the list of variable declarations to their binding-time type $\in$ Bt-type.

$$\frac{\forall i \in \{1,\ldots,n\} \;.\; \bar{\tau}, \rho_{i-1}, \eta \vdash^{bt}_{decl} (x_i : T_{x_i}) : \rho_i \quad \text{where } \rho_0 = \text{empty Record-type-environment} \qquad x \notin \mathit{Domain}(\bar{\tau})}{\bar{\tau}, \eta \vdash^{bt}_{def} (\texttt{record}\ x\ ((x_1 : T_{x_1}) \ldots (x_n : T_{x_n}))\ \texttt{S}\ \texttt{E}) : \bar{\tau}[x \mapsto \rho_n]}$$

$$\frac{\forall i \in \{1,\ldots,n\} \;.\; \bar{\tau}, \rho_{i-1}, \eta \vdash^{bt}_{decl} (x_i : T_{x_i}) : \rho_i \quad \text{where } \rho_0 = \text{empty Record-type-environment} \qquad \forall i \in \{1,\ldots,n\} \;.\; \rho_n(x_i)\#b = \texttt{D} \qquad x \notin \mathit{Domain}(\bar{\tau})}{\bar{\tau}, \eta \vdash^{bt}_{def} (\texttt{record}\ x\ ((x_1 : T_{x_1}) \ldots (x_n : T_{x_n}))\ \texttt{D}\ \texttt{R}) : \bar{\tau}[x \mapsto \rho_n]}$$

$$\frac{\forall i \in \{1,\ldots,n\} \;.\; \bar{\tau}, \rho_{i-1}, \eta \vdash^{bt}_{decl} (x_i : T_{x_i}) : \rho_i \quad \text{where } \rho_0 = \text{empty Record-type-environment} \qquad \exists i \in \{1,\ldots,n\} \;.\; \rho_n(x_i)\#m = \texttt{R} \qquad x \notin \mathit{Domain}(\bar{\tau})}{\bar{\tau}, \eta \vdash^{bt}_{def} (\texttt{record}\ x\ ((x_1 : T_{x_1}) \ldots (x_n : T_{x_n}))\ \texttt{S}\ \texttt{R}) : \bar{\tau}[x \mapsto \rho_n]}$$

$$\frac{}{\bar{\tau}, \eta \vdash^{bt}_{defs} () : \bar{\tau}} \qquad \frac{\bar{\tau}, \eta \vdash^{bt}_{def} f : \bar{\tau}' \qquad \bar{\tau}', \eta \vdash^{bt}_{defs} (f^*) : \bar{\tau}''}{\bar{\tau}, \eta \vdash^{bt}_{defs} (f\ f^*) : \bar{\tau}''}$$

Figure 6.11: Binding-time type checking rules for type definitions

6.5.2 Binding-time type checking of type definitions

The binding-time type checking of type definitions is given as a transition relation from Type-environment, Type-header, and Type-definition to Type-environment, as follows:

$$\bar{\tau}, \eta \vdash^{bt}_{def} f : \bar{\tau}'$$

and this relation is defined in figure 6.11.

As we explained in section 6.1, types in the FCL-ann language will have annotations attached to them by users. The purpose of the binding-time type checking rules for type definitions is to check for the correctness of the given annotations and to define the division of the type definition based on the given annotations.

The first three rules for type definitions define the binding-time type checking rules for a single type definition. These rules are evaluated in a similar manner. First, we have to evaluate all variables in the record using rules in figure 6.12. These evaluations begin with an empty table $\in$ Record-type-environment and each evaluation repeatedly modifies this table. Then, we check for the non-existence of the record name in the table $\bar{\tau}$ domain. Finally, we update the type environment $\bar{\tau}$ for this record.

A record bt-type and bt-mode are based on the attached annotations. If the attached

$$\frac{\bar{\tau},\eta \vdash^{bt}_{type} T : BT, \quad BT\#b = \mathtt{S} \quad \& \quad BT\#m = \mathtt{E} \quad x \notin Domain(\bar{\varepsilon})}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{decl} (x : T) : \bar{\varepsilon}[x \mapsto BT]}$$

$$\frac{\bar{\tau},\eta \vdash^{bt}_{type} T : BT, \quad BT\#b = \mathtt{D} \quad \& \quad BT\#m = \mathtt{R} \quad x \notin Domain(\bar{\varepsilon})}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{decl} \underline{(x : T)} : \bar{\varepsilon}[x \mapsto BT]}$$

$$\frac{\bar{\tau},\eta \vdash^{bt}_{type} T : BT, \quad BT\#b = \mathtt{S} \quad \& \quad BT\#m = \mathtt{R} \quad x \notin Domain(\bar{\varepsilon})}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{decl} \underline{\underline{(x : T)}} : \bar{\varepsilon}[x \mapsto BT]}$$

$$\frac{}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{spec}_{decls} () \Rightarrow \bar{\varepsilon}'} \qquad \frac{\bar{\tau},\bar{\varepsilon},\eta \vdash^{spec}_{decl} d \Rightarrow \bar{\varepsilon}' \quad \bar{\tau},\bar{\varepsilon}',\eta \vdash^{spec}_{decls} (d^*) \Rightarrow \bar{\varepsilon}''}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{spec}_{decls} (d\ d^*) \Rightarrow \bar{\varepsilon}''}$$

Figure 6.12: Binding-time type checking rules for variable declarations

annotation is dynamic and residual and the binding-time value of all variable declarations in the record is dynamic (so residual), then the type definition is underlined. If the attached annotation is static and residual and there must exist at least one residual variable declaration, then the type definition is dash-underlined. Otherwise, the type definition is a non-underlined.

The fourth and the fifth rules define the binding-time type checking rules for an empty type definition and a sequence of type definitions respectively.

6.5.3 Binding-time type checking of variable declarations

The binding-type time checking of variable declarations is expressed as a transition relation from Variable-environment, Type-environment, Type-header, and Variable-declaration to Variable-environment, as follows:

$$\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{decl} f : \bar{\varepsilon}'$$

and this relation is defined in figure 6.12.

The binding-time type checking rules for a single variable declaration are evaluated as follows. First, we evaluate the type T yielding the binding-time type BT. Then, we check for the non-existence of the variable name in the domain of the environment $\bar{\varepsilon}$. Finally, we update the environment $\bar{\varepsilon}$ for that variable with binding-time type BT and annotate the

Nat type

$$\frac{}{\bar{\tau}, \eta \vdash^{bt}_{type} (\texttt{nat S E}) : \mathsf{nat}(\mathtt{S}, \mathtt{E})} \qquad \frac{}{\bar{\tau}, \eta \vdash^{bt}_{type} (\underline{\texttt{nat D R}}) : \mathsf{nat}(\mathtt{D}, \mathtt{R})}$$

Access type

$$\frac{\bar{\tau}, \eta \vdash^{bt}_{type} T : BT}{\bar{\tau}, \eta \vdash^{bt}_{type} (\texttt{access } T \texttt{ S E}) : \mathsf{access}(BT, \mathtt{S}, \mathtt{E})} \qquad \frac{\bar{\tau}, \eta \vdash^{bt}_{type} T : BT, \quad BT\#b = \mathtt{D} \quad \& \quad BT\#m = \mathtt{R}}{\bar{\tau}, \eta \vdash^{bt}_{type} \underline{(\texttt{access } T \texttt{ D R})} : \mathsf{access}(BT, \mathtt{D}, \mathtt{R})}$$

$$\frac{\bar{\tau}, \eta \vdash^{bt}_{type} T : BT, \quad BT\#m = \mathtt{R}}{\bar{\tau}, \eta \vdash^{bt}_{type} \underline{\underline{(\texttt{access } T \texttt{ S R})}} : \mathsf{access}(BT, \mathtt{S}, \mathtt{R})}$$

Array type

$$\frac{\bar{\tau}, \eta \vdash^{bt}_{type} T : BT}{\bar{\tau}, \eta \vdash^{bt}_{type} (\texttt{array } n_1\ n_2\ T \texttt{ S E}) : \mathsf{array}(BT, n_1, n_2, \mathtt{S}, \mathtt{E})}$$

$$\frac{\bar{\tau}, \eta \vdash^{bt}_{type} T : BT, \quad BT\#b = \mathtt{D} \quad \& \quad BT\#m = \mathtt{R}}{\bar{\tau}, \eta \vdash^{bt}_{type} \underline{(\texttt{array } n_1\ n_2\ T \texttt{ D R})} : \mathsf{array}(BT, n_1, n_2, \mathtt{D}, \mathtt{R})}$$

$$\frac{\bar{\tau}, \eta \vdash^{bt}_{type} T : BT, \quad BT\#m = \mathtt{R}}{\bar{\tau}, \eta \vdash^{bt}_{type} \underline{\underline{(\texttt{array } n_1\ n_2\ T \texttt{ S R})}} : \mathsf{array}(BT, n_1, n_2, \mathtt{S}, \mathtt{R})}$$

Record type

$$\frac{\eta(x) = \langle \mathtt{S}, \mathtt{E} \rangle}{\bar{\tau}, \eta \vdash^{bt}_{type} (\texttt{record } x) : \mathsf{record}(x, \mathtt{S}, \mathtt{E})} \qquad \frac{\eta(x) = \langle \mathtt{D}, \mathtt{R} \rangle}{\bar{\tau}, \eta \vdash^{bt}_{type} \underline{(\texttt{record } x)} : \mathsf{record}(x, \mathtt{D}, \mathtt{R})}$$

$$\frac{\eta(x) = \langle \mathtt{S}, \mathtt{R} \rangle}{\bar{\tau}, \eta \vdash^{bt}_{type} \underline{\underline{(\texttt{record } x)}} : \mathsf{record}(x, \mathtt{S}, \mathtt{R})}$$

Figure 6.13: Binding-time type checking rules for types

variable declaration based on the binding-time type BT.

If the binding-time type BT is dynamic and residual, then the variable declaration is underlined. If the binding-time type BT is static and residual, then the variable declaration is dash-underlined. Otherwise, it is a non-underlined variable declaration.

The fourth and the the fifth rules define the binding-time type checking rules for an empty variable declaration and a sequence of variable declarations.

Get variable from dynamic left-expression

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{lexp} le : BT}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{lexp} \text{(get-dyn } le\text{)} : BT}$$

Get variable and location from static-residual left-expression

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{lexp} le : BT}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{lexp} \text{(get-res } le\text{)} : BT}$$

Variable reference

$$\frac{}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{lexp} x : \bar{\varepsilon}(x)}$$

Figure 6.14: Binding-time type checking rules for left-expressions (*part 1*)

6.5.4 Binding-time type checking of types

Definition 6.4 *Let T be an annotated type, $\bar{\tau}$ be a type environment, and η be a type table defined for all type names in the type definitions list. Then, the type T is binding-time type correct if there exists a $BT \in$ Bt-type such that:*

$$\bar{\tau}, \eta \vdash^{bt}_{type} T : BT$$

where the relation $\vdash^{bt}_{type}$ is defined in figure 6.13.

As a consequence of the binding-time type well-formedness in definition 6.3, the binding-time type checking rules for nat types are defined only for the non-underline and the underline types. These rules are straightforward.

The binding-time type checking rules for an access types and array types are also have to follow the requirements in definition 6.3. If the type binding-time type is static and residual, then its components or members binding-time mode have to be residual. If the type binding-time type is dynamic and residual, then its components or members binding-time type have to be dynamic and residual too. The rules return an access binding-time type tag for access types or an array binding-time type tag for array types.

For record types, the rules return a record binding-time type tag and annotate the record type based on the binding-time tuple fetched from the table η for the record name x.

Array indexing

$$\frac{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e_1 : \mathsf{array}(BT, n_1, n_2, \mathtt{S}, \mathtt{E}) \quad \bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e_2 : \mathsf{nat}(\mathtt{S},\mathtt{E})}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{lexp} (\mathtt{[]}\ e_1\ e_2) : BT}$$

$$\frac{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e_1 : \mathsf{array}(BT, n_1, n_2, \mathtt{D}, \mathtt{R}) \quad \bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e_2 : \mathsf{nat}(\mathtt{D},\mathtt{R})}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{lexp} \underline{(\mathtt{[]}\ e_1\ e_2)} : BT}$$

$$\frac{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e_1 : \mathsf{array}(BT, n_1, n_2, \mathtt{S}, \mathtt{R}) \quad \bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e_2 : \mathsf{nat}(\mathtt{S},\mathtt{E})}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{lexp} \underline{\underline{(\mathtt{[]}\ e_1\ e_2)}} : BT}$$

Record indexing

$$\frac{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e : \mathsf{record}(x', \mathtt{S}, \mathtt{E}) \quad \bar{\tau}(x')(x) = BT}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{lexp} (\mathtt{recmem}\ e\ x) : BT} \qquad \frac{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e : \mathsf{record}(x', \mathtt{D}, \mathtt{R}) \quad \bar{\tau}(x')(x) = BT}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{lexp} \underline{(\mathtt{recmem}\ e\ \mathtt{x})} : BT}$$

$$\frac{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e : \mathsf{record}(x', \mathtt{S}, \mathtt{R}) \quad \bar{\tau}(x')(x) = BT}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{lexp} \underline{\underline{(\mathtt{recmem}\ e\ \mathtt{x})}} : BT}$$

Access dereference

$$\frac{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e : \mathsf{access}(BT, \mathtt{S}, \mathtt{E})}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{lexp} (\mathtt{deref}\ e) : BT} \qquad \frac{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e : \mathsf{access}(BT, \mathtt{D}, \mathtt{R})}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{lexp} \underline{(\mathtt{deref}\ e)} : BT}$$

$$\frac{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{exp} e : \mathsf{access}(BT, \mathtt{S}, \mathtt{R})}{\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{lexp} \underline{\underline{(\mathtt{deref}\ e)}} : BT}$$

Figure 6.15: Binding-time type checking rules for left-expressions (*part 2*)

6.5.5 Binding-time type checking of left-expressions

Definition 6.5 *Let le be a left-expression in an assignment, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, $\bar{\varepsilon}$ be an environment defined for all variable names in the variable declarations list, and η be a type table defined for all type names in the type definition list. Then, the left-expression le is binding-time type correct if there exists a $BT \in$ Bt-type such that:*

$$\bar{\tau},\bar{\varepsilon},\eta \vdash^{bt}_{lexp} le : BT$$

where the relation $\vdash^{bt}_{lexp}$ is defined in figure 6.14 and figure 6.15.

To simplify the specialization process in our application, a variable reference left-expression always returns a location. If the left-expression appears in a dynamic and residual assignment, then to fetch the residual variable name from the store σ at specialization time, we insert a new operator, called get-dyn. If the left-expression appears in a static

Constant

$$\frac{}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} n : \mathsf{nat}(\mathtt{S}, \mathtt{E})}$$

Lift

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e : \mathsf{nat}(\mathtt{S}, \mathtt{E})}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} (\mathsf{lift}\ e) : \mathsf{nat}(\mathtt{D}, \mathtt{R})}$$

Get value

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e : \mathsf{record}(x', \mathtt{S}, \mathtt{R})}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} (\mathsf{get\text{-}val}\ e) : \mathsf{record}(x', \mathtt{S}, \mathtt{E})}$$

Get expression

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e : \mathsf{access}(BT, \mathtt{S}, \mathtt{R})}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} (\mathsf{get\text{-}exp}\ e) : \mathsf{access}(BT, \mathtt{D}, \mathtt{R})} \qquad \frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e : \mathsf{array}(BT, n_1, n_2, \mathtt{S}, \mathtt{R})}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} (\mathsf{get\text{-}exp}\ e) : \mathsf{array}(BT, n_1, n_2, \mathtt{D}, \mathtt{R})}$$

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e : \mathsf{record}(x', \mathtt{S}, \mathtt{R})}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} (\mathsf{get\text{-}exp}\ e) : \mathsf{record}(x', \mathtt{D}, \mathtt{R})}$$

Variable reference

$$\frac{}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{lexp} x : \bar{\varepsilon}(x)}$$

Binary operation

$$\frac{\begin{array}{l}\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e_1 : BT_1 \\ \bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e_2 : BT_2 \end{array} \qquad \bar{\tau}, op \vdash^{bt}_{op-equiv} (BT_1, BT_2) \rightarrow BT, \quad BT\#b = \mathtt{S} \quad \& \quad BT\#m = \mathtt{E}}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} (op\ e_1\ e_2) : BT}$$

$$\frac{\begin{array}{l}\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e_1 : BT_1 \\ \bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e_2 : BT_2 \end{array} \qquad \bar{\tau}, op \vdash^{bt}_{op-equiv} (BT_1, BT_2) \rightarrow BT, \quad BT\#b = \mathtt{D} \quad \& \quad BT\#m = \mathtt{R}}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} \underline{(op\ e_1\ e_2)} : BT}$$

Figure 6.16: Binding-time type checking rules for expressions (*part 1*)

and residual context, then for a similar reason as above, we insert another new operator, called get-res. Both operators return the same binding-time type as the binding-time type returned by the left-expression they are applied to.

A variable reference is not annotated. It is bound to a binding-time type associated with the variable in the environment $\bar{\varepsilon}$.

The rules for access dereference left-expressions and array indexing left-expressions are similar in style to their corresponding type checking rules from the previous chapter. An access dereference left-expression is annotated based on the binding-time type of the component expression e and the rules return the component binding-time type BT, while an array indexing left-expression is annotated based on the binding-time type of its expression e_1 and the rules return the binding-time types BT.

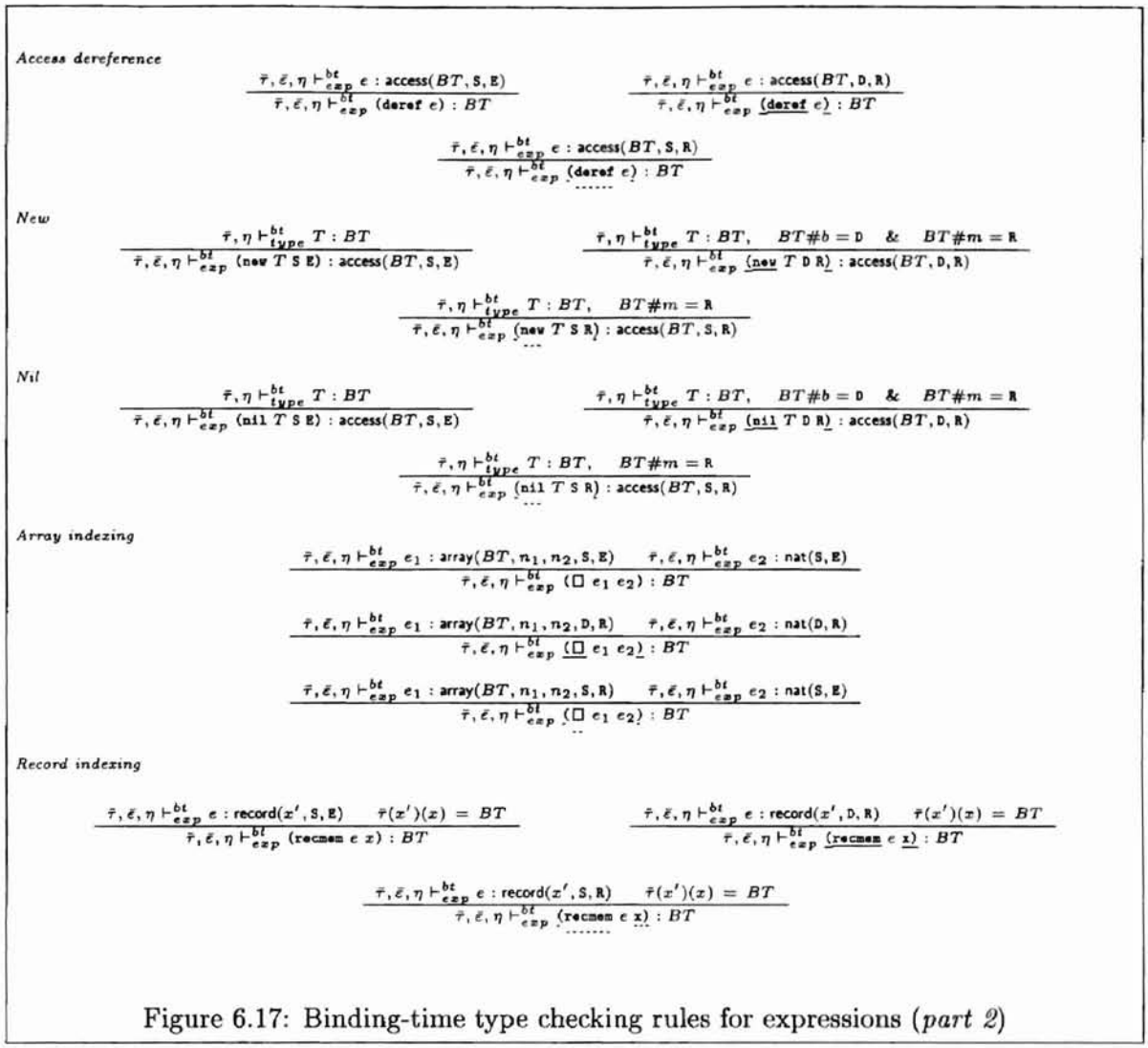


Figure 6.17: Binding-time type checking rules for expressions (*part 2*)

The binding-time type checking rules for record indexing left-expressions are also similar to the corresponding type checking rule. The left-expression is annotated based on the binding-time type of its expression e and the rules return the binding-time type extracted from the type environment $\bar{\tau}$.

6.5.6 Binding-time type checking of expressions

Definition 6.6 *Let e be an expression in an assignment, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, $\bar{\varepsilon}$ be an environment defined for all variable names in the variable declarations list, and η be a type table defined for all type names in the type definitions list. Then, the expression e is binding-time type correct if there exists a*

Operation bt-type equivalence for op $\in \{+, -, *, /, >, \texttt{gte}, <, \texttt{lte}\}$

Nat type

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{nat(S,E)}, \mathsf{nat(S,E)} \rangle \rightarrow \mathsf{nat(S,E)}} \quad \text{if } op \in \{+, -, *, /, >, \texttt{gte}, <, \texttt{lte}\}$$

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{nat(S,E)}, \mathsf{nat(D,R)} \rangle \rightarrow \mathsf{nat(D,R)}} \quad \text{if } op \in \{+, -, *, /, >, \texttt{gte}, <, \texttt{lte}\}$$

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{nat(D,R)}, \mathsf{nat(S,E)} \rangle \rightarrow \mathsf{nat(D,R)}} \quad \text{if } op \in \{+, -, *, /, >, \texttt{gte}, <, \texttt{lte}\}$$

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{nat(D,R)}, \mathsf{nat(D,R)} \rangle \rightarrow \mathsf{nat(D,R)}} \quad \text{if } op \in \{+, -, *, /, >, \texttt{gte}, <, \texttt{lte}\}$$

Operation bt-type equivalence for op $\in \{=, \texttt{neq}\}$

Nat type

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{nat(S,E)}, \mathsf{nat(S,E)} \rangle \rightarrow \mathsf{nat(S,E)}} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{nat(S,E)}, \mathsf{nat(D,R)} \rangle \rightarrow \mathsf{nat(D,R)}} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{nat(D,R)}, \mathsf{nat(S,E)} \rangle \rightarrow \mathsf{nat(D,R)}} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{nat(D,R)}, \mathsf{nat(D,R)} \rangle \rightarrow \mathsf{nat(D,R)}} \quad \text{if } op \in \{=, \texttt{neq}\}$$

Figure 6.18: Binding-time type checking rules for operation bt-types equivalence (*part 1*)

$BT \in$ Bt-type *such that:*

$$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e : BT$$

where the relation $\vdash^{bt}_{exp}$ *is defined in figure 6.16 and figure 6.17.*

A constant is always annotated as non-underline and the rule returns a static and eliminable nat binding-time type tag.

The lift operator is applied to an expression that returns a static and eliminable nat binding-time type tag and the rule returns a dynamic and residual nat binding-time type tag.

Consider a situation where an expression that returns a static and residual binding-time type tag exists in a non-underlined or an underlined context. At specialization time, to be able to residualize or to eliminate this expression, we have to fetch the appropriate part from the tuple returned by the expression. If the expression is a record type expression,

Access type

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{access}(BT, \mathtt{S}, \mathtt{E}), \mathsf{access}(BT, \mathtt{S}, \mathtt{E}) \rangle \rightarrow \mathsf{nat}(\mathtt{S}, \mathtt{E})} \quad \text{if } op \in \{=, \mathtt{neq}\}$$

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{access}(BT, c, \mathtt{R}), \mathsf{access}(BT, c, \mathtt{R}) \rangle \rightarrow \mathsf{nat}(\mathtt{D}, \mathtt{R})} \quad \text{if } op \in \{=, \mathtt{neq}\}$$

Array type

$$\frac{BT\#m = \mathtt{E}}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{array}(BT, n_1, n_2, \mathtt{S}, \mathtt{E}), \mathsf{array}(BT, n_1, n_2, \mathtt{S}, \mathtt{E}) \rangle \rightarrow \mathsf{nat}(\mathtt{S}, \mathtt{E})} \quad \text{if } op \in \{=, \mathtt{neq}\}$$

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{array}(BT, n_1, n_2, c, \mathtt{R}), \mathsf{array}(BT, n_1, n_2, c, \mathtt{R}) \rangle \rightarrow \mathsf{nat}(\mathtt{D}, \mathtt{R})} \quad \text{if } op \in \{=, \mathtt{neq}\}$$

Record type

$$\frac{\bar{\tau}(x) = \begin{bmatrix} x_1 \mapsto BT_1 \\ \vdots \\ x_n \mapsto BT_n \end{bmatrix} \qquad \forall i \in \{1, \ldots, n\} \;.\; BT_i\#m = \mathtt{E}}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{record}(x, \mathtt{S}, \mathtt{E}), \mathsf{record}(x, \mathtt{S}, \mathtt{E}) \rangle \rightarrow \mathsf{nat}(\mathtt{S}, \mathtt{E})} \quad \text{if } op \in \{=, \mathtt{neq}\}$$

$$\frac{}{\bar{\tau}, op \vdash^{bt}_{op-equiv} \langle \mathsf{record}(x, c, \mathtt{R}), \mathsf{record}(x, c, \mathtt{R}) \rangle \rightarrow \mathsf{nat}(\mathtt{D}, \mathtt{R})} \quad \text{if } op \in \{=, \mathtt{neq}\}$$

Figure 6.19: Binding-time type checking rules for operation bt-types equivalence (*part 2*)

then to fetch the value part from the returning tuple, we introduce a new operator, called get-val. This operator returns a static and eliminable record binding-time type tag.

If the expression is either an access, an array or a record type expression, then to fetch the residual code part from the returning tuple, we introduce a new operator, called get-exp. This operator returns a dynamic and residual binding-time type tag for the corresponding type.

For variable reference, access dereference, array indexing, and record indexing expressions, the binding-time type checking rules are similar to their corresponding binding-time type checking rules for left-expressions.

Since binary operations are defined only for nat type operands, their binding-time type checking rules are also defined only for the non-underlined and the underlined versions. These rules use the binding-time type checking rules in figure 6.18 and figure 6.19 to map both expression binding-time types to the result binding-time type BT. The binary

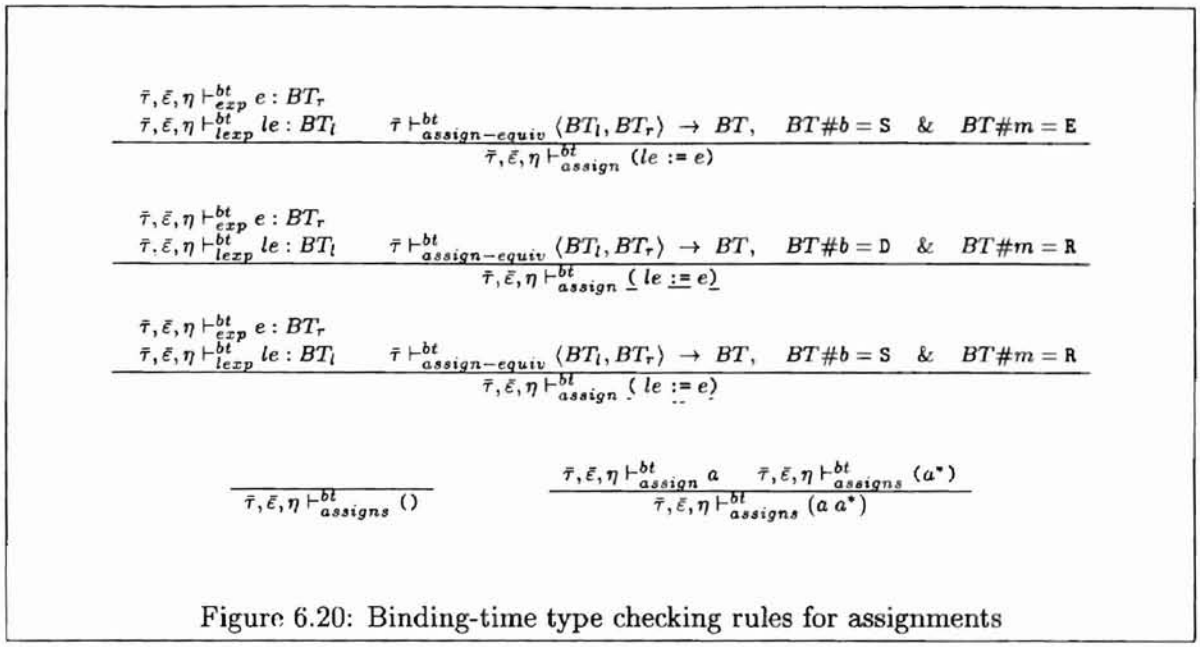


Figure 6.20: Binding-time type checking rules for assignments

operation expression is annotated based on the binding-time type BT.

A new or a nil expression is annotated based on the annotations given by users. These rules follow the same requirement as for the binding-time type checking rules for access types.

6.5.7 Binding-time type checking of operation bt-types equivalence

Definition 6.7 *Let op be an operator in an expression, BT_1 and BT_2 be binding-time types. Then, the operation op between BT_1 and BT_2 is binding-time type correct if there exists a $BT \in$ Bt-type such that:*

$$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{op-equiv} \langle BT_1, BT_2 \rangle \rightarrow BT$$

where the relation $\vdash^{bt}_{op-equiv}$ is defined in figure 6.18 and figure 6.19.

The binding-time type checking rules for operation bt-types equivalence are similar to the type checking rules for operation types equivalence.

The rules define valid operand binding-time types in a particular operation *op* and the operation result binding-time type. For example, if the operator *op* $\in$ {+, −, *, /, >, gte, <, lte} and the binding-time type of both operands is static and

Nat type

$$\frac{}{\bar{\tau} \vdash^{bt}_{assign-equiv} \langle \mathsf{nat(S,E)}, \mathsf{nat(S,E)} \rangle \rightarrow \mathsf{nat(S,E)}} \qquad \frac{}{\bar{\tau} \vdash^{bt}_{assign-equiv} \langle \mathsf{nat(D,R)}, \mathsf{nat(S,E)} \rangle \rightarrow \mathsf{nat(D,R)}}$$

$$\frac{}{\bar{\tau} \vdash^{bt}_{assign-equiv} \langle \mathsf{nat(D,R)}, \mathsf{nat(D,R)} \rangle \rightarrow \mathsf{nat(D,R)}}$$

Access type

$$\frac{}{\bar{\tau} \vdash^{bt}_{assign-equiv} \langle \mathsf{access}(BT, c, m), \mathsf{access}(BT, c, m) \rangle \rightarrow \mathsf{access}(BT, c, m)}$$

Array type

$$\frac{}{\bar{\tau} \vdash^{bt}_{assign-equiv} \langle \mathsf{array}(BT, n_1, n_2, c, m), \mathsf{array}(BT, n_1, n_2, c, m) \rangle \rightarrow \mathsf{array}(BT, n_1, n_2, c, m)}$$

Record type

$$\frac{}{\bar{\tau} \vdash^{bt}_{assign-equiv} \langle \mathsf{record}(x, c, m), \mathsf{record}(x, c, m) \rangle \rightarrow \mathsf{record}(x, c, m)}$$

Figure 6.21: Binding-time type checking rules for assignment bt-types equivalence

eliminable nat type, then the result binding-time type is also static and eliminable nat type.

The rest of rules are evaluated in similar fashion.

6.5.8 Binding-time type checking of assignments

Definition 6.8 *Let* $a \equiv (le := e)$ *be an assignment in a block,* $\bar{\tau}$ *be a type environment defined for all type names in the type definitions list,* $\bar{\varepsilon}$ *be an environment defined for all variable names in the variable declarations list, and* η *be a type table defined for all type names in the type definitions list. Then, the assignment a is binding-time type correct iff:*

$$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{assign} a$$

where the relation $\vdash^{bt}_{assign}$ *is defined in figure 6.20.*

The binding-time type checking rules for assignments are similar to their corresponding type checking rules in the previous chapter.

For a single assignment, we use binding-time type checking rules for assignment binding-time type equivalence in figure 6.21 to map both expression binding-time types to a binding-

Goto

$$\frac{}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{jump} \texttt{(goto } l\texttt{)}}$$

If

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e : \mathsf{nat(S,E)}}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{jump} \texttt{(if } e\ l_1\ l_2\texttt{)}} \qquad \frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e : \mathsf{nat(D,R)}}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{jump} \underline{\texttt{(if } e \texttt{ then } l_1 \texttt{ else } l_2 \texttt{)}}}$$

Return

$$\frac{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{exp} e : \mathsf{nat(D,R)}}{\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{jump} \underline{\texttt{(return } e\texttt{)}}}$$

Figure 6.22: Binding-time type checking rules for jumps

time type BT. The assignment is annotated based on the binding-time type BT.

The fourth and the the fifth rules define the binding-time type checking rules for an empty assignment and a sequence of assignments.

6.5.9 Binding-time type checking of assignment bt-types equivalence

Definition 6.9 *Let BT_1 and BT_2 be binding-time types. Then, BT_1 and BT_2 are assignment binding-time types equivalent if there exists a $BT \in$ Bt-type such that:*

$$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{assign-equiv} \langle BT_1, BT_2 \rangle \rightarrow BT$$

where the relation $\vdash^{bt}_{assign-equiv}$ is defined in figure 6.21.

The binding-time type checking rules for assignment binding-time types equivalence are similar in style to the binding-time type checking rules for operation binding-time types equivalence.

The rules define valid left-expression and (right) expression binding-time types in an assignment and return the binding-time type of the assignment.

6.5.10 Binding-time type checking of jumps

Definition 6.10 *Let j be a jump in a block, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, $\bar{\varepsilon}$ be an environment defined for all variable names in the variable declarations list, and η be a type table defined for all type names in the type definitions list. Then, the jump j is binding-time type correct iff:*

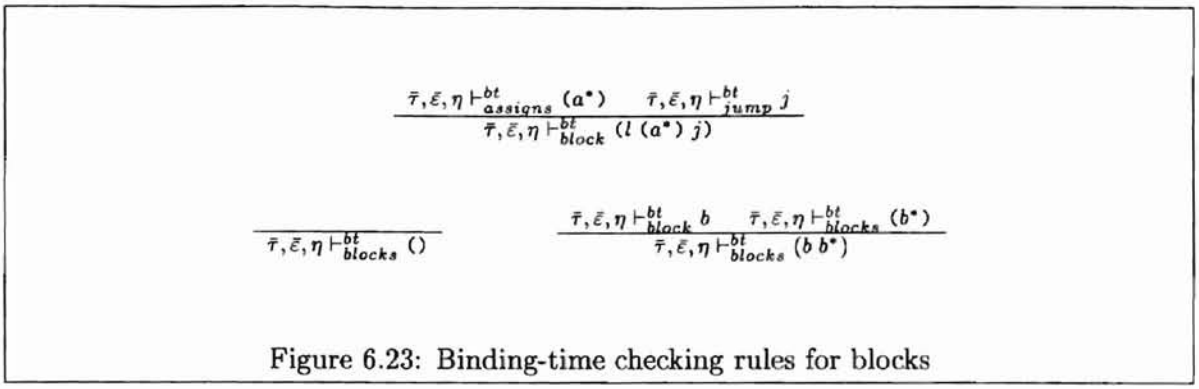


Figure 6.23: Binding-time checking rules for blocks

$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{jump} j$

where the relation $\vdash^{bt}_{jump}$ *is defined in figure 6.22.*

A goto jump will always be removed at specialization time, so it should be annotated as non-underline.

The rules for if jumps are defined only for the non-underline and the underline versions. The jump is annotated based on the binding-time type returned by the expression e.

A return jump is always annotated as underline because opposite to goto jump, all occurences of return will be residualized at specialization time.

6.5.11 Binding-time type checking of blocks

Definition 6.11 *Let b be a block in a program, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, $\bar{\varepsilon}$ be an environment defined for all variable names in the variable declarations list, and η be a type table defined for all type names in the type definitions list. Then, the block b is binding-time type correct iff:*

$\bar{\tau}, \bar{\varepsilon}, \eta \vdash^{bt}_{block} b$

where the relation $\vdash^{bt}_{block}$ *is defined in figure 6.23.*

The binding-time type checking rules for blocks are textually equal with their corresponding type checking rules.

$$\frac{\bar{\tau},\eta \vdash^{bt}_{defs} (f^*) : \bar{\tau}' \quad \bar{\tau}',\bar{\varepsilon},\eta \vdash^{bt}_{decls} (d^*) : \bar{\varepsilon}' \quad \bar{\tau}',\bar{\varepsilon}',\eta \vdash^{bt}_{blocks} (\hat{b}^+)}{\eta \vdash^{bt}_{prog} (((f^*)(d^*))\ (x^*)\ ((\hat{b}^+)))}$$

where $\bar{\tau}$ = empty Type-environment
$\bar{\varepsilon}$ = empty Variable-environment

Figure 6.24: Binding-time type checking rule for programs

6.5.12 Binding-time type checking of programs

Definition 6.12 *Let* $p \equiv (((f^*)(d^*))\ (x^*)\ (b^+))$ *be an program and* η *be a type list defined for all type names in the type definition list. Then, the program* p *is binding-time type correct iff:*

$$\eta \vdash^{bt}_{prog} p$$

where the relation $\vdash^{bt}_{prog}$ *is defined in figure 6.24.*

The binding-time type checking rules for programs is also textually equal with its corresponding type checking rules.

6.6 Three-level FCL semantics

In this section, we formally define the semantics of the three-level FCL language. This also defines the semantics of the specialization. Specialization is guided by annotations that are attached to constructs by the previous binding-time type checking. The non-underlined program constructs will be computed away, the underlined program constructs will be residualized, and the dash-underline program constructs will be computed and residualized.

6.6.1 Partial evaluation traces

Similar to the FCL evaluation semantics in chapter 5, we will formalize the specialization semantics in terms of *partial evaluation traces*, that contain values only for known informations. For example, the following is a partial evaluation traces for the power program with the known parameter n = 2. In this example, we represent an unknown value with a special tag D.

u	$\in$	PE-lvalue	=	Loc $\cup$ Left-expression $\cup$ (Left-expression $\times$ Loc)
w	$\in$	PE-value	=	Value $\cup$ Expression $\cup$ (Expression $\times$ Value)
v	$\in$	Value	=	Nat + Loc + Array + Record
n	$\in$	Nat	=	$\mathbb{N}$
h	$\in$	Loc	=	$\mathbb{N} \cup \{nil\}$
α	$\in$	Array-map	=	$\mathbb{N} \rightarrow$ Loc
		Array	=	Array-map $\times$ $\mathbb{N}$ $\times$ $\mathbb{N}$
ρ	$\in$	Record	=	Variable $\rightarrow$ Loc
l	$\in$	Label	=	Block-label $\cup$ halt($\mathbb{N}$)
$\bar{\tau}$	$\in$	Type-environment	=	Variable $\rightarrow$ Record-type-environment
		Record-type-environment	=	Variable $\rightarrow$ Type
ε	$\in$	Variable-environment	=	Variable $\rightarrow$ Loc
σ	$\in$	Store	=	Loc $\rightarrow$ PE-value
Γ	$\in$	Block-map	=	Block-label $\rightarrow$ Block

Figure 6.25: Specialization rules semantics domains and values

$$
\begin{aligned}
& (\texttt{start}, [\texttt{m} \mapsto \texttt{D}, \texttt{n} \mapsto 2, \texttt{result} \mapsto 0]) \\
\rightarrow\ & (\texttt{test}, [\texttt{m} \mapsto \texttt{D}, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) \\
\rightarrow\ & (\texttt{loop}, [\texttt{m} \mapsto \texttt{D}, \texttt{n} \mapsto 2, \texttt{result} \mapsto 1]) \\
\rightarrow\ & (\texttt{test}, [\texttt{m} \mapsto \texttt{D}, \texttt{n} \mapsto 1, \texttt{result} \mapsto \texttt{D}]) \\
\rightarrow\ & (\texttt{loop}, [\texttt{m} \mapsto \texttt{D}, \texttt{n} \mapsto 1, \texttt{result} \mapsto \texttt{D}]) \\
\rightarrow\ & (\texttt{test}, [\texttt{m} \mapsto \texttt{D}, \texttt{n} \mapsto 0, \texttt{result} \mapsto \texttt{D}]) \\
\rightarrow\ & (\texttt{done}, [\texttt{m} \mapsto \texttt{D}, \texttt{n} \mapsto 0, \texttt{result} \mapsto \texttt{D}]) \\
\rightarrow\ & (\texttt{halt}, [\texttt{m} \mapsto \texttt{D}, \texttt{n} \mapsto 0, \texttt{result} \mapsto \texttt{D}])
\end{aligned}
$$

6.6.2 Semantics domains and values

Domains and values of the specialization rules are presented in figure 6.25.

- A semantic PE-lvalue $u \in$ PE-lvalue = Loc $\cup$ Left-expression $\cup$ (Left-expression $\times$ Loc) where Loc is an $h \in$ Loc = $\mathbb{N} \cup \{nil\}$.
- A semantic PE-value $w \in$ PE-value = Value $\cup$ Expression $\cup$ (Expression $\times$ Value).
- A semantic value $v \in$ Value = Nat + Loc + Record + Array is one of the following:

- An $n \in$ Nat $=$ $\mathbb{N}$.
- An $h \in$ Loc.
- An $\alpha \in$ Array-map $=$ $\mathbb{N} \rightarrow$ Loc is a partial function mapping an array index $\in$ $\mathbb{N}$ to its memory location $\in$ Loc. The array value in Array $=$ Array-map $\times$ $\mathbb{N}$ $\times$ $\mathbb{N}$ consist of an array map, and the upper and lower bound of the array index set.
- A $\rho \in$ Record $=$ Variable$\rightarrow$Loc.

- A semantic value $l \in$ Label $=$ Block-label $\cup$ halt.
- A *type environment* $\bar{\tau} \in$ Type-environment is a partial function from a variable $\in$ Variable to $\rho \in$ Record-type-environment, where a ρ is a mapping from a variable $\in$ Variable to its type $\in$ Type.
- An *environment* $\varepsilon \in$ Variable-environment is a partial function that maps a variable $\in$ Variable to its location $\in$ Loc.
- A *store* $\sigma \in$ Store is a partial function from a location $\in$ Loc to its PE-value $\in$ PE-value.

6.6.3 Specialization of type definitions

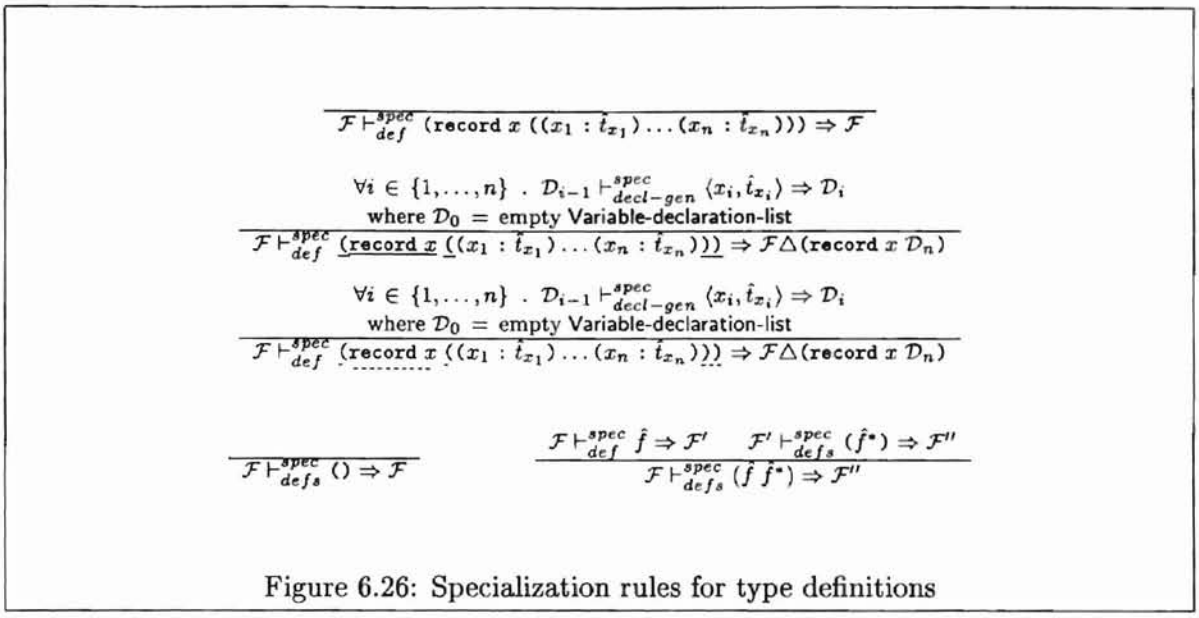
Let $\mathcal{F} \in$ Type-definition-list be a residual FCL program type definitions list. The specialization of type definitions is defined as a transition relation from Type-definition-list and Type-definition to Type-definition-list, as follows:

$$\mathcal{F} \vdash^{spec}_{def} \hat{f} \Rightarrow \mathcal{F}'$$

and this relation is defined in figure 6.26.

The first rule is for a single non-underlined type definition. The rule returns the original residual FCL program type definitions list $\mathcal{F}$ without any modification.

The second rule and the third rule are defined for a single underlined type definition and a single dash-underlined type definition respectively. These rules use a new data structure



$$\frac{}{\mathcal{F} \vdash^{spec}_{def} (\texttt{record}\ x\ ((x_1 : \hat{t}_{x_1}) \ldots (x_n : \hat{t}_{x_n}))) \Rightarrow \mathcal{F}}$$

$$\frac{\begin{array}{c}\forall i \in \{1,\ldots,n\}\ .\ \mathcal{D}_{i-1} \vdash^{spec}_{decl-gen} \langle x_i, \hat{t}_{x_i} \rangle \Rightarrow \mathcal{D}_i \\ \text{where } \mathcal{D}_0 = \text{empty Variable-declaration-list}\end{array}}{\mathcal{F} \vdash^{spec}_{def} (\texttt{record}\ x\ ((x_1 : \hat{t}_{x_1}) \ldots (x_n : \hat{t}_{x_n}))) \Rightarrow \mathcal{F} \triangle (\texttt{record}\ x\ \mathcal{D}_n)}$$

$$\frac{\begin{array}{c}\forall i \in \{1,\ldots,n\}\ .\ \mathcal{D}_{i-1} \vdash^{spec}_{decl-gen} \langle x_i, \hat{t}_{x_i} \rangle \Rightarrow \mathcal{D}_i \\ \text{where } \mathcal{D}_0 = \text{empty Variable-declaration-list}\end{array}}{\mathcal{F} \vdash^{spec}_{def} (\texttt{record}\ x\ ((x_1 : \hat{t}_{x_1}) \ldots (x_n : \hat{t}_{x_n}))) \Rightarrow \mathcal{F} \triangle (\texttt{record}\ x\ \mathcal{D}_n)}$$

$$\frac{}{\mathcal{F} \vdash^{spec}_{defs} () \Rightarrow \mathcal{F}} \qquad \frac{\mathcal{F} \vdash^{spec}_{def} \hat{f} \Rightarrow \mathcal{F}' \quad \mathcal{F}' \vdash^{spec}_{defs} (\hat{f}^*) \Rightarrow \mathcal{F}''}{\mathcal{F} \vdash^{spec}_{defs} (\hat{f}\ \hat{f}^*) \Rightarrow \mathcal{F}''}$$

Figure 6.26: Specialization rules for type definitions

$\mathcal{D} \in$ Variable-declaration-list which essentially is a residual FCL program variable declarations list. Both rules are evaluated as follow. First, we repeatedly insert [2] a new residual FCL variable declaration at the end of the variable declarations list $\mathcal{D}$ using rules in figure 6.27. This process begins with an empty Variable-declaration-list ($\mathcal{D}_0$). Then, we insert a new FCL record type definition using the final modification of the variable declarations list ($\mathcal{D}_n$) as its component at the end of the FCL type definitions list $\mathcal{F}$ and return this FCL type definitions list.

The fourth and the fifth rules define the specialization rules for an empty type definition and a sequence of type definitions.

6.6.4 Specialization of variable declarations

Let $\mathcal{D} \in$ Variable-declaration-list be a residual FCL program variable declarations list. The specialization of variable declarations is defined as a transition relation from Type-environment, Variable-environment, Store, Variable-declaration-list, Loc, and Variable-declaration to Type-environment, Variable-environment, Store, Variable-declaration-list, and Loc, as follows:

[2] $\triangle$ is the symbol of snoc function which will add a singleton at the end of a list

$$\frac{\bar{\tau}, \sigma, \mathcal{D}, h' \vdash^{spec}_{init} \hat{t} \Rightarrow \langle v, \sigma', \mathcal{D}', h'' \rangle}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{decl} (x : \hat{t}) \Rightarrow \langle \varepsilon[x \mapsto h], \sigma'[h \mapsto v], \mathcal{D}', h'' \rangle} \quad \text{where } h' = succ(h)$$

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{decl} (\underline{x : \hat{t}}) \Rightarrow \langle \varepsilon[x \mapsto h], \sigma[h \mapsto \textsf{make-var}(h)], \mathcal{D}, h' \rangle} \quad \text{where } h' = succ(h)$$

$$\frac{\bar{\tau}, \sigma, \mathcal{D}, h' \vdash^{spec}_{init} \hat{t} \Rightarrow \langle v, \sigma', \mathcal{D}', h'' \rangle}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{decl} (\underline{\underline{x : \hat{t}}}) \Rightarrow \langle \varepsilon[x \mapsto h], \sigma'[h \mapsto \langle \textsf{make-var}(h), v \rangle], \mathcal{D}', h'' \rangle} \quad \text{where } h' = succ(h)$$

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{decls} () \Rightarrow \langle \varepsilon, \sigma, \mathcal{D}, h \rangle}$$

$$\frac{\begin{array}{l}\mathcal{D} \vdash^{spec}_{decl-gen} \langle h, \hat{t} \rangle \Rightarrow \mathcal{D}' \\ \quad \text{where } \hat{d} = (x : \hat{t}) \\ \bar{\tau}, \varepsilon, \sigma, \mathcal{D}', h \vdash^{spec}_{decl} \hat{d} \Rightarrow \langle \varepsilon', \sigma', \mathcal{D}'', h' \rangle \qquad \bar{\tau}, \varepsilon', \sigma', \mathcal{D}'', h' \vdash^{spec}_{decls} (\hat{d}^*) \Rightarrow \langle \varepsilon'', \sigma'', \mathcal{D}''', h'' \rangle\end{array}}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{decls} (\hat{d}\ \hat{d}^*) \Rightarrow \langle \varepsilon'', \sigma'', \mathcal{D}''', h'' \rangle}$$

Figure 6.27: Specialization rules for variable declarations

$$\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{decl} \hat{d} \Rightarrow \langle \varepsilon', \sigma', \mathcal{D}', h' \rangle$$

and this relation is defined in figure 6.27.

The first three rules are the specialization rules for a single variable declaration. They cover the case where the variable declaration is either a non-underlined, an underlined, or a dash-underlined variable declaration. These rules are evaluated like the evaluation rules for variable declarations, except for a slight different with the underlined version and the dash-underlined version. Instead of setting the store so that the next available location maps to the variable initial value (by invoking rules in figure 6.31 and figure 6.32), we map it to a unique new variable name corresponds to h (by invoking function make-var that converts a location to a variable) that represents the variable x in the residual code for the underlined version and with a tuple containing the unique new variable name and the variable initial value for the dash-underlined version.

The fourth rule defines the meaning of the specialization rule for an empty variable declaration.

Since not all residual variable declarations are going to be residualized (*e.g.* variable declarations in an underlined or a dash-underlined type definition), we generate a new

$$\frac{\bar{\tau}, \sigma, \mathcal{D}, h' \vdash^{spec}_{init} \hat{t} \Rightarrow \langle v, \sigma', \mathcal{D}', h'' \rangle}{\bar{\tau}, \alpha, \sigma, \mathcal{D}, h \vdash^{spec}_{array-init} \langle n, \hat{t} \rangle \Rightarrow \langle \alpha[n \mapsto h], \sigma'[h \mapsto v], \mathcal{D}', h'' \rangle} \quad \text{if } \hat{t} \text{ is a non-underlined type where } h' = succ(h)$$

$$\frac{}{\bar{\tau}, \alpha, \sigma, \mathcal{D}, h \vdash^{spec}_{array-init} \langle n, \hat{t} \rangle \Rightarrow \langle \alpha[n \mapsto h], \sigma[h \mapsto \mathsf{make\text{-}var}(h)], \mathcal{D}, h' \rangle} \quad \text{if } \hat{t} \text{ is an underlined type where } h' = succ(h)$$

$$\frac{\bar{\tau}, \sigma, \mathcal{D}, h' \vdash^{spec}_{init} \hat{t} \Rightarrow \langle v, \sigma', \mathcal{D}', h'' \rangle}{\bar{\tau}, \alpha, \sigma, \mathcal{D}, h \vdash^{spec}_{array-init} \langle n, \hat{t} \rangle \Rightarrow \langle \alpha[n \mapsto h], \sigma'[h \mapsto \langle \mathsf{make\text{-}var}(h), v \rangle], \mathcal{D}', h'' \rangle} \quad \text{if } \hat{t} \text{ is a dash-underlined type where } h' = succ(h)$$

Figure 6.28: Specialization rules for array members initialization

residual FCL variable declaration and append it into the list not when we specialize each variable declaration, but when we specialize a sequence of variable declarations, as we can see in the last rule in the figure. With this method, we can use the specialization rules for a single variable declaration for general situations.

6.6.5 Specialization of array members initialization

Let $\mathcal{D} \in$ Variable-declaration-list be a residual FCL program variable declarations list. The specialization of array members initialization is defined as a transition relation from Type-environment, Array-map, Store, Variable-declaration-list, Loc, IN, and Type to Type-environment, Array-map, Store, Variable-declaration-list, and Loc, as follows:

$$\bar{\tau}, \alpha, \sigma, \mathcal{D}, h \vdash^{spec}_{array-init} \langle n, \hat{t} \rangle \Rightarrow \langle \alpha', \sigma', \mathcal{D}', h' \rangle$$

and this relation is defined in figure 6.27.

The specialization rules for array members initialization are evaluated similar to the specialization rules for a singe variable declaration in the previous subsection, except these rules use a natural number (array index) instead of a variable.

6.6.6 Specialization rules to generate residual variable declarations

Let $\mathcal{D}$ be a residual FCL program variable declarations list. The specialization rules to generate a residual variable declaration are defined as a transition relation from Variable-declaration-list, Variable, and Type to Variable-declaration-list, as follows

$$\mathcal{D} \vdash^{spec}_{decl-gen} \langle x, \hat{t} \rangle \Rightarrow \mathcal{D}'$$

$$\frac{}{\mathcal{D} \vdash^{spec}_{decl-gen} \langle x, \hat{t} \rangle \Rightarrow \mathcal{D}} \quad \text{if } \hat{t} \text{ is a non-underlined type}$$

$$\frac{\vdash^{spec}_{type} \hat{t} \Rightarrow t}{\mathcal{D} \vdash^{spec}_{decl-gen} \langle x, \hat{t} \rangle \Rightarrow \mathcal{D} \triangle (x : t)} \quad \text{if } \hat{t} \text{ is not a non-underlined type}$$

Figure 6.29: Specialization rules for residual variable declaration generations

where this relation is defined in figure 6.29.

The purpose of the rules is to generate an FCL residual variable declaration whenever the annotated type $\hat{t}$ binding-time mode is residual and to insert the residual variable declaration at the end of the residual FCL program variable declarations list $\mathcal{D}$.

The rules are straightforward.

6.6.7 Specialization of residual types

Definition 6.13 *Let $\hat{t}$ be an annotated type. Then, the specialization of the annotated type $\hat{t}$ yields a type t iff:*

$\vdash^{spec}_{type} \hat{t} \Rightarrow t$

where the relation $\vdash^{spec}_{type}$ is defined in figure 6.30.

The specialization rules for residual types are a set of rules defined for the underlined and the dash-underlined types only. These rules return the corresponding residual FCL type t for the annotated type $\hat{t}$ as we can see in figure 6.30.

6.6.8 Specialization of static types

Definition 6.14 *Let $\hat{t}$ be an annotated type, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, σ be the current store, $\mathcal{D}$ be a residual FCL program variable declarations list, and h be a next available location. Then, the specialization of the static annotated type $\hat{t}$ yields a value v, a new store σ', a possibly modified residual FCL program variable declarations list, and a new next available location h' iff:*

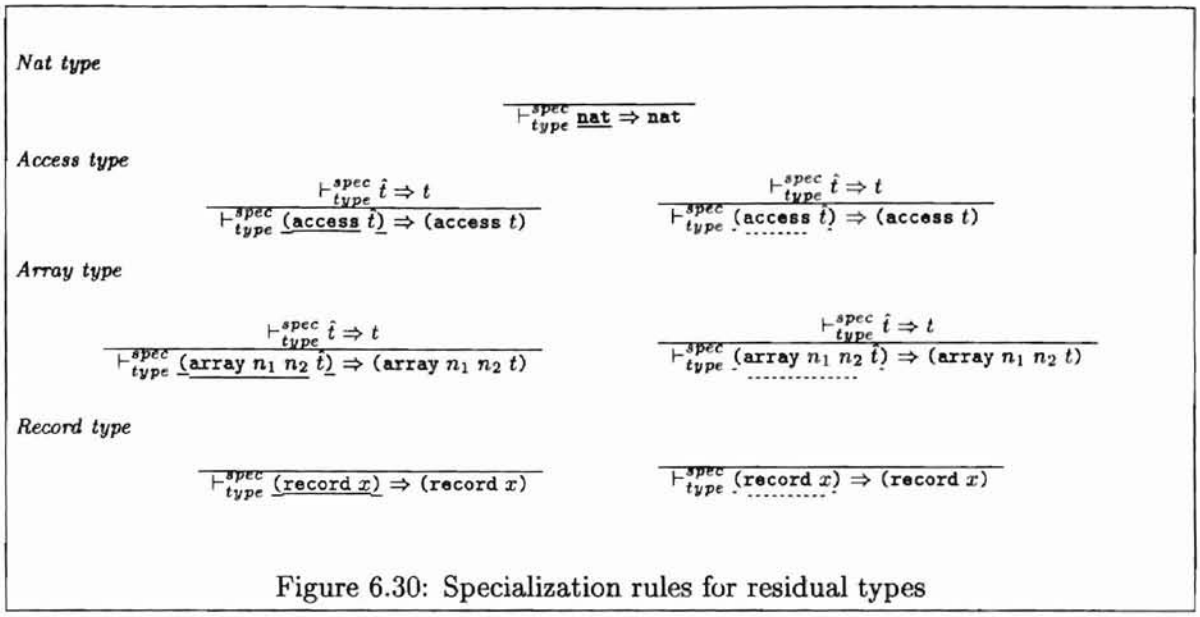


Figure 6.30: Specialization rules for residual types

$$\bar{\tau}, \sigma, \mathcal{D}, h \vdash^{spec}_{init} \hat{t} \Rightarrow \langle v, \sigma', \mathcal{D}', h' \rangle$$

where the relation $\vdash^{spec}_{init}$ *is defined in figure 6.31 and figure 6.32.*

The specialization rules for static types are a complementary set of rules for the specialization rules for residual types in the previous subsection. As expected, these rules are defined only for the non-underlined and the dash-underlined types.

The rules are evaluated like their corresponding evaluation rules in chapter 5, except with a slight difference for non-underlined array types and non-underlined record types. In those rules, we also generate a new residual FCL program variable declaration using rules in figure 6.29 and append it into the residual FCL program variable declarations list $\mathcal{D}$.

6.6.9 Specialization of left-expressions

Definition 6.15 *Let* $\widehat{le}$ *be an annotated left-expression,* $\bar{\tau}$ *be a type environment defined for all type names in the type definitions list,* ϵ *be an environment defined for all variable names in the variable declarations list,* σ *be the current store,* $\mathcal{D}$ *be a residual FCL program variable declarations list, and* h *be a next available location. Then, the specialization of the annotated left-expression* $\widehat{le}$ *yields a PE-lvalue* u*, a new store* σ'*, a possibly modified residual FCL program variable declarations list, and a new next available location* h' *iff:*

Nat type

$$\frac{}{\bar{\tau}, \sigma, \mathcal{D}, h \vdash^{spec}_{init} \mathbf{nat} \Rightarrow \langle \mathsf{nat}(0), \sigma, \mathcal{D}, h \rangle}$$

Access type

$$\frac{}{\bar{\tau}, \sigma, \mathcal{D}, h \vdash^{spec}_{init} (\mathbf{access}\ \hat{t}) \Rightarrow \langle \mathsf{loc}(nil), \sigma, \mathcal{D}, h \rangle} \qquad \frac{}{\bar{\tau}, \sigma, \mathcal{D}, h \vdash^{spec}_{init} \underline{(\mathbf{access}\ \hat{t})} \Rightarrow \langle \mathsf{loc}(nil), \sigma, \mathcal{D}, h \rangle}$$

Array type

$$\frac{\begin{array}{l} \forall i \in \{n_1, \ldots, n_2\} \ . \ \ \mathcal{D}_{i-1} \vdash^{spec}_{decl-gen} \langle h_{i-1}, \hat{t}_i \rangle \Rightarrow \mathcal{D}'_{i-1} \\ \qquad\qquad \bar{\tau}, \alpha_{i-1}, \sigma_{i-1}, \mathcal{D}'_{i-1}, h_{i-1} \vdash^{spec}_{array-init} \langle i, \hat{t}_i \rangle \Rightarrow \langle \alpha_i, \sigma_i, \mathcal{D}_i, h_i \rangle \\ \text{where} \quad \alpha_{n_1-1} = \text{empty Array-map} \\ \qquad\quad \sigma_{n_1-1} = \sigma \\ \qquad\quad \mathcal{D}_{n_1-1} = \mathcal{D} \\ \qquad\quad h_{n_1-1} = h \end{array}}{\bar{\tau}, \sigma, \mathcal{D}, h \vdash^{spec}_{init} (\mathbf{array}\ n_1\ n_2\ \hat{t}) \Rightarrow \langle \mathsf{array}(\alpha_{n_2}, n_1, n_2), \sigma_{n_2}, \mathcal{D}_{n_2}, h_{n_2} \rangle}$$

$$\frac{\begin{array}{l} \forall i \in \{n_1, \ldots, n_2\} \ . \ \bar{\tau}, \alpha_{i-1}, \sigma_{i-1}, \mathcal{D}'_{i-1}, h_{i-1} \vdash^{spec}_{array-init} \langle i, \hat{t}_i \rangle \Rightarrow \langle \alpha_i, \sigma_i, \mathcal{D}_i, h_i \rangle \\ \text{where} \quad \alpha_{n_1-1} = \text{empty Array-map} \\ \qquad\quad \sigma_{n_1-1} = \sigma \\ \qquad\quad \mathcal{D}_{n_1-1} = \mathcal{D} \\ \qquad\quad h_{n_1-1} = h \end{array}}{\bar{\tau}, \sigma, \mathcal{D}, h \vdash^{spec}_{init} \underline{(\mathbf{array}\ n_1\ n_2\ \hat{t})} \Rightarrow \langle \mathsf{array}(\alpha_{n_2}, n_1, n_2), \sigma_{n_2}, \mathcal{D}_{n_2}, h_{n_2} \rangle}$$

Figure 6.31: Specialization rules for static types (*part 1*)

$$\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{lexp} \widehat{le} \Rightarrow \langle u, \sigma', \mathcal{D}', h' \rangle$$

where the relation $\vdash^{spec}_{lexp}$ *is defined in figure 6.33 and figure 6.34.*

The specialization of variable reference, non-underlined access dereference, non-underlined array indexing, and non-underlined record indexing left-expressions are similar to their evaluation counterparts.

The first rule is the specialization rule for a get-dyn operator. The rule is evaluated by evaluating the left-expression $\widehat{le}$ yielding a location h_e, which is then used to extract a variable x from the store σ. The rule returns the variable x.

The specialization rule for get-res operator is justified similar to the specialization rule for a get-dyn operator, except that the store associated with the location h_e is a tuple containing a variable x and a value v. The rule returns a tuple containing the variable x and the location h_e.

An underlined and a dash-underlined access dereference left-expressions are special-

Record type

$$\frac{\begin{array}{l}\forall i \in \{1,\ldots,n\} \;.\; \mathcal{D}_{i-1} \vdash^{spec}_{decl-gen} \langle h_{i-1}, \hat{t}_i \rangle \Rightarrow \mathcal{D}'_{i-1} \\ \qquad \bar{\tau}, \rho_{i-1}, \sigma_{i-1}, \mathcal{D}'_{i-1}, h_{i-1} \vdash^{spec}_{decl} (x_i : \hat{t}_i) \Rightarrow \langle \rho_i, \sigma_i, \mathcal{D}_i, h_i \rangle \\ \bar{\tau}(x) = \left[\begin{array}{c} x_1 \mapsto \hat{t}_1 \\ \vdots \\ x_n \mapsto \hat{t}_n \end{array}\right] \quad \text{where} \quad \begin{array}{rcl} \rho_0 & = & \text{empty Record} \\ \sigma_0 & = & \sigma \\ \mathcal{D}_0 & = & \mathcal{D} \\ h_0 & = & h \end{array}\end{array}}{\bar{\tau}, \sigma, \mathcal{D}, h \vdash^{spec}_{init} (\texttt{record}\ x) \Rightarrow \langle \text{record}(\rho_n), \sigma_n, \mathcal{D}_n, h_n \rangle}$$

$$\frac{\begin{array}{l}\forall i \in \{1,\ldots,n\} \;.\; \bar{\tau}, \rho_{i-1}, \sigma_{i-1}, \mathcal{D}_{i-1}, h_{i-1} \vdash^{spec}_{decl} (x_i : \hat{t}_i) \Rightarrow \langle \rho_i, \sigma_i, \mathcal{D}_i, h_i \rangle \\ \bar{\tau}(x) = \left[\begin{array}{c} x_1 \mapsto \hat{t}_1 \\ \vdots \\ x_n \mapsto \hat{t}_n \end{array}\right] \quad \text{where} \quad \begin{array}{rcl} \rho_0 & = & \text{empty Record} \\ \sigma_0 & = & \sigma \\ \mathcal{D}_0 & = & \mathcal{D} \\ h_0 & = & h \end{array}\end{array}}{\bar{\tau}, \sigma, \mathcal{D}, h \vdash^{spec}_{init} \underline{(\texttt{record}\ x)} \Rightarrow \langle \text{record}(\rho_n), \sigma_n, \mathcal{D}_n, h_n \rangle}$$

Figure 6.32: Specialization rules for static types (*part 2*)

ized by evaluating the expression $\hat{e}$ which returns an FCL expression e for the underlined version or a tuple containing an FCL expression e and a non nil location h_e for the dash-underlined version. The underlined version rule returns a residual FCL access dereference left-expression and the dash-underlined version rule returns a tuple containing a residual FCL access dereference left-expression and the location h_e. In both cases, the residual FCL access dereference has the FCL expression e as its component.

To specialize an underlined and a dash-underlined array indexing left-expressions, we have to specialize annotated expressions $\hat{e}_1$ and $\hat{e}_2$. For the underlined version, the specialization returns FCL expressions e_1 and e_2 respectively. For the dash-underlined version, the $\hat{e}_1$ specialization returns a tuple containing an FCL expression e_1 and an array value with α as its array-map, while the $\hat{e}_2$ specialization returns a nat value n. The overall specialization returns a residual FCL array indexing left-expression for the underlined version or a tuple containing a residual FCL array indexing left-expression with literal nat value n as its component and a location for the dash-underlined version.

The specialization rules for the underlined and the dash-underlined record indexing left-expressions are justified by first evaluating the annotated expression $\hat{e}$ which returns a residual FCL expression e for the underlined version or a tuple containing a residual

Get variable from dynamic left-expression

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{lexp} \widehat{le} \Rightarrow \langle h_e, \sigma', \mathcal{D}', h' \rangle \qquad \sigma'(h_e) = x}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{lexp} (\textsf{get-dyn}\ \widehat{le}) \Rightarrow \langle x, \sigma', \mathcal{D}', h' \rangle}$$

Get variable and location from static-residual left-expression

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{lexp} \widehat{le} \Rightarrow \langle h_e, \sigma', \mathcal{D}', h' \rangle \qquad \sigma'(h_e) = \langle x, v \rangle}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{lexp} (\textsf{get-res}\ \widehat{le}) \Rightarrow \langle \langle x, h_e \rangle, \sigma', \mathcal{D}', h' \rangle}$$

Variable reference

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{lexp} x \Rightarrow \langle \varepsilon(x), \sigma, \mathcal{D}, h \rangle}$$

Access dereference

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle \mathsf{loc}(h_e), \sigma', \mathcal{D}', h' \rangle \qquad h_e \neq nil}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{lexp} (\textbf{deref}\ \hat{e}) \Rightarrow \langle h_e, \sigma', \mathcal{D}', h' \rangle}$$

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle e, \sigma', \mathcal{D}', h' \rangle}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{lexp} \underline{(\textbf{deref}\ \hat{e})} \Rightarrow \langle (\textbf{deref}\ e), \sigma', \mathcal{D}', h' \rangle}$$

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle \langle e, \mathsf{loc}(h_e) \rangle, \sigma', \mathcal{D}', h' \rangle \qquad h_e \neq nil}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{lexp} \underline{(\textbf{deref}\ \hat{e})} \Rightarrow \langle \langle (\textbf{deref}\ e), h_e \rangle, \sigma', \mathcal{D}', h' \rangle}$$

Figure 6.33: Specialization rules for left-expressions (*part 1*)

FCL expression e and a record value for the dash-underlined version. For the underlined version, the overall rule returns a residual FCL record indexing left-expression, while for the dash-underlined version, it returns a tuple containing a residual FCL record indexing left-expression and a location extracted from the relation ρ for the variable x.

6.6.10 Specialization of expressions

Definition 6.16 *Let $\hat{e}$ be an annotated expression, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, ε be an environment defined for all variable names in the variable declarations list, σ be the current store, $\mathcal{D}$ be a residual FCL program variable declarations list, and h be a next available location. Then, the specialization of the annotated expression $\hat{e}$ is a PE-value w, a new store σ', a possibly modified residual FCL program variable declarations list, and a new next available location h' iff:*

$$\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle w, \sigma', \mathcal{D}', h' \rangle$$

Array indexing

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{exp} \hat{e}_1 \Rightarrow \langle \mathsf{array}(\alpha,n_1,n_2),\sigma',\mathcal{D}',h'\rangle \quad \bar{\tau},\varepsilon,\sigma',\mathcal{D}',h' \vdash^{spec}_{exp} \hat{e}_2 \Rightarrow \langle \mathsf{nat}(n),\sigma'',\mathcal{D}''h''\rangle \quad n_1 \leq n \leq n_2}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{lexp} (\square\ \hat{e}_1\ \hat{e}_2) \Rightarrow \langle \alpha(n),\sigma'',\mathcal{D}'',h''\rangle}$$

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{exp} \hat{e}_1 \Rightarrow \langle e_1,\sigma',\mathcal{D}',h'\rangle \quad \bar{\tau},\varepsilon,\sigma',\mathcal{D}',h' \vdash^{spec}_{exp} \hat{e}_2 \Rightarrow \langle e_2,\sigma'',\mathcal{D}''h''\rangle}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{lexp} \underline{(\square\ \hat{e}_1\ \hat{e}_2)} \Rightarrow \langle (\square\ e_1\ e_2),\sigma'',\mathcal{D}'',h''\rangle}$$

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{exp} \hat{e}_1 \Rightarrow \langle\langle e_1,\mathsf{array}(\alpha,n_1,n_2)\rangle,\sigma',\mathcal{D}',h'\rangle \quad \bar{\tau},\varepsilon,\sigma',\mathcal{D}',h' \vdash^{spec}_{exp} \hat{e}_2 \Rightarrow \langle \mathsf{nat}(n),\sigma'',\mathcal{D}''h''\rangle \quad n_1 \leq n \leq n_2}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{lexp} \underline{(\square\ \hat{e}_1\ \hat{e}_2)} \Rightarrow \langle\langle(\square\ e_1\ n),\alpha(n)\rangle,\sigma'',\mathcal{D}'',h''\rangle}$$

Record indexing

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle \mathsf{record}(\rho),\sigma',\mathcal{D}',h'\rangle}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{lexp} (\mathtt{recmem}\ \hat{e}\ x) \Rightarrow \langle \rho(x),\sigma',\mathcal{D}',h'\rangle}$$

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle e,\sigma',\mathcal{D}',h'\rangle}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{lexp} \underline{(\mathtt{recmem}\ \hat{e}\ \mathtt{x})} \Rightarrow \langle (\mathtt{recmem}\ e\ x),\sigma',\mathcal{D}',h'\rangle}$$

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle\langle e,\mathsf{record}(\rho)\rangle,\sigma',\mathcal{D}',h'\rangle}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{lexp} \underline{(\mathtt{recmem}\ \hat{e}\ \mathtt{x})} \Rightarrow \langle\langle(\mathtt{recmem}\ e\ x),\rho(x)\rangle,\sigma',\mathcal{D}',h'\rangle}$$

Figure 6.34: Specialization rules for left-expressions (*part 2*)

where the relation $\vdash^{spec}_{exp}$ *is defined in figure 6.35, figure 6.36 and figure 6.37.*

A constant, a non-underlined binary operation, and a non-underlined nil expressions are specialized in a manner similar to their evaluation counterparts.

The specialization rules for variable reference, access dereference, array indexing and record indexing expressions are quite similar to their corresponding specialization rules for left-expressions. The only difference is in these rules, instead of returning a location or a tuple containing a location, they return a PE-value or a tuple containing a PE-value where the PE-value is extracted from the store for the location.

Lifting returns the literal of the nat value n.

A get-val operation and a get-exp operation are evaluated by extracting the result of specializing the dash-underlined expression $\hat{e}$. The get-val operation extracts the value part, while the get-exp operation extracts the expression part.

Constant

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} n \Rightarrow \langle \mathsf{nat}(n), \sigma, \mathcal{D}, h \rangle}$$

Lift

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle \mathsf{nat}(n), \sigma', \mathcal{D}', h' \rangle}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} (\mathsf{lift}\ \hat{e}) \Rightarrow \langle n, \sigma', \mathcal{D}', h' \rangle}$$

Get value

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle \langle e, \mathsf{record}(\rho) \rangle, \sigma', \mathcal{D}', h' \rangle}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} (\mathsf{get\text{-}val}\ \hat{e}) \Rightarrow \langle \mathsf{record}(\rho), \sigma', \mathcal{D}', h' \rangle}$$

Get expression

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle \langle e, v \rangle, \sigma', \mathcal{D}', h' \rangle}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} (\mathsf{get\text{-}exp}\ \hat{e}) \Rightarrow \langle e, \sigma', \mathcal{D}', h' \rangle}$$

Variable reference

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} x \Rightarrow \langle \sigma(\varepsilon(x)), \sigma, \mathcal{D}, h \rangle}$$

Binary operation

$$\frac{\begin{array}{l}\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e}_1 \Rightarrow \langle v_1, \sigma', \mathcal{D}', h' \rangle \\ \bar{\tau}, \varepsilon, \sigma', \mathcal{D}', h' \vdash^{spec}_{exp} \hat{e}_2 \Rightarrow \langle v_2, \sigma'', \mathcal{D}'', h'' \rangle \qquad \sigma'', op \vdash^{spec}_{op} \langle v_1\ ,\ v_2 \rangle \rightarrow v\end{array}}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} (op\ \hat{e}_1\ \hat{e}_2) \Rightarrow \langle v, \sigma'', \mathcal{D}'', h'' \rangle}$$

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e}_1 \Rightarrow \langle e_1, \sigma', \mathcal{D}', h' \rangle \qquad \bar{\tau}, \varepsilon, \sigma', \mathcal{D}', h' \vdash^{spec}_{exp} \hat{e}_2 \Rightarrow \langle e_2, \sigma'', \mathcal{D}'', h'' \rangle}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \underline{(op\ \hat{e}_1\ \hat{e}_2)} \Rightarrow \langle (op\ e_1\ e_2), \sigma'', \mathcal{D}'', h'' \rangle}$$

Figure 6.35: Specialization rules for expressions (*part 1*)

An underlined binary operation is evaluated by first evaluating annotated expressions $\hat{e}_1$ and $\hat{e}_2$ yielding FCL expressions e_1 and e_2 respectively. The overall rule returns a residual FCL binary operation expression with e_1 and e_2 as its left and (right) expressions.

The specialization rules for non-underlined new expressiona are defined for three different conditions based on the binding-time type of the annotated type $\hat{t}$. The first rule covers the case where the annotated type $\hat{t}$ is a non-underlined type. The rule is justified similar to the evaluation rule for new expressions.

The second rule covers the case where the annotated type $\hat{t}$ is an underlined type. This rule (and the next rule) represents the actual specialization process to remove dynamic allocation (dynamism) in the configurable system. In this case, dynamism generated via new expression is replaced with a new residual variable (static allocation) declared using rules in figure 6.29. The variable name is saved in the store σ so it can be referenced when later needed.

The third rule covers the case where the annotated type $\hat{t}$ is a dash-underlined type.

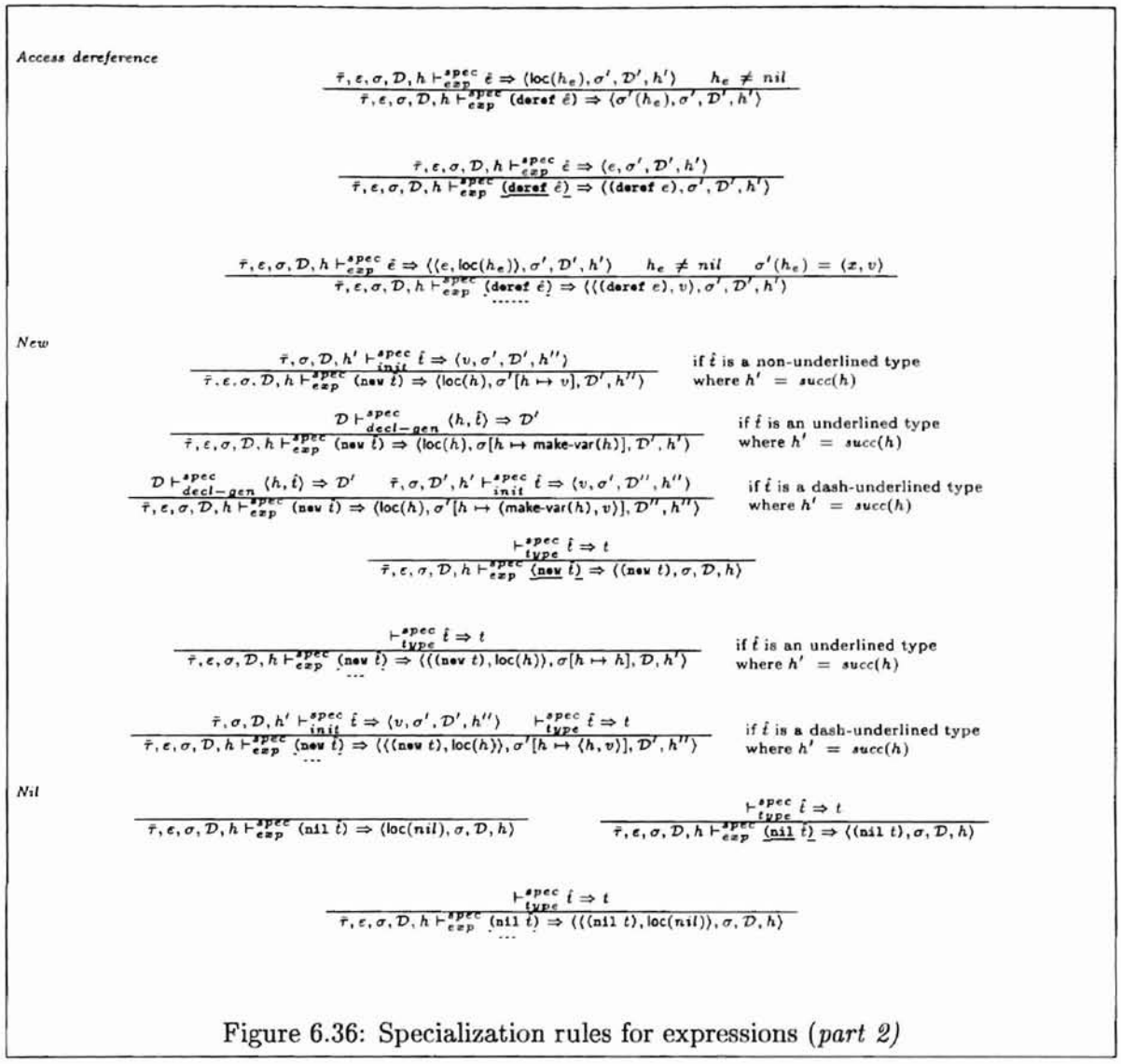


Figure 6.36: Specialization rules for expressions *(part 2)*

The rule evaluation is essentially a combination of the first and the second rule evaluations above.

The specialization rule for underline new expressions returns a residual FCL new expression.

For dash-underlined new expressions, the specialization rules are also defined for cases based on the binding-time type of the annotated type $\hat{t}$. If the annotated type $\hat{t}$ is an underlined type, then the rule returns a tuple containing a new residual FCL new expression and the next available location h. For completeness, the store associated with the location h is updated with the same location (it represents the static variable corresponded to the new

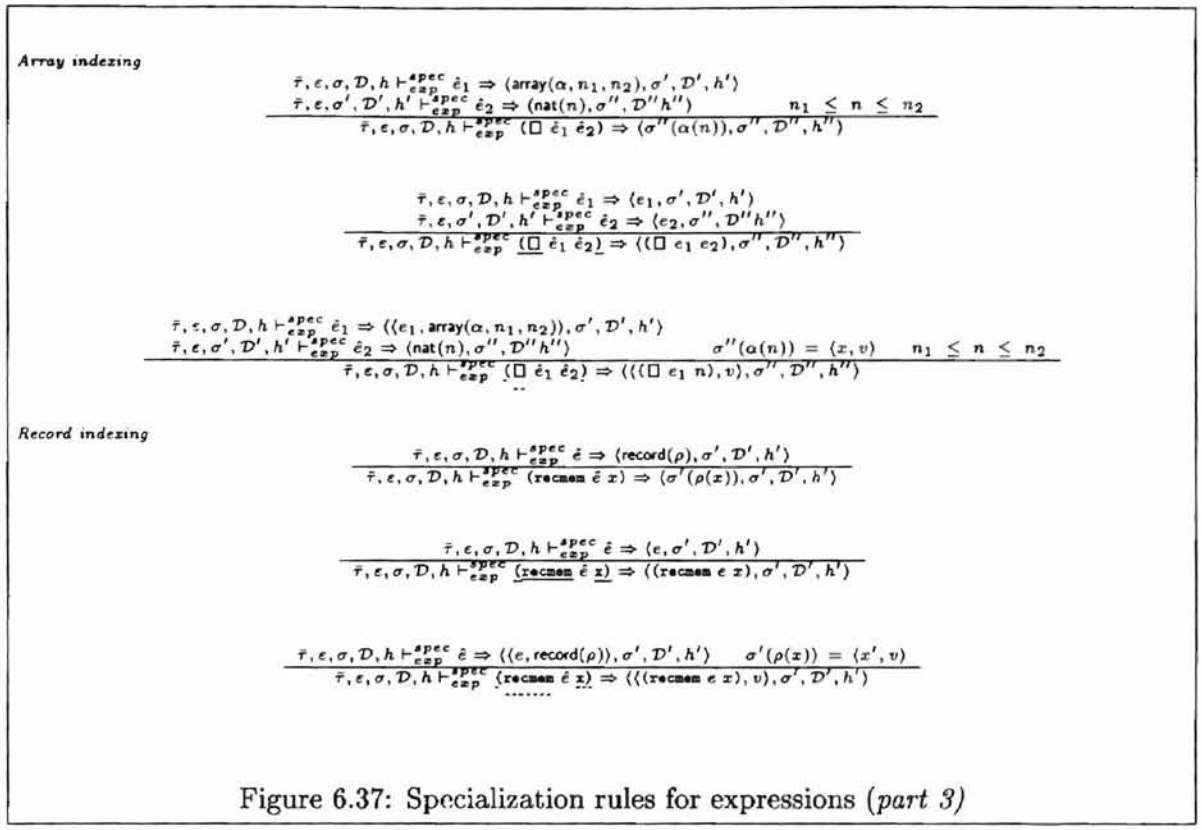


Figure 6.37: Specialization rules for expressions *(part 3)*

dynamic allocation that actually will never be dereferenced). If the annotated type $\hat{t}$ is a dash-underlined type, then the rules return the same tuple as above, while the store associated with the location h is updated with a tuple containing the location and the annotated type $\hat{t}$ initial value.

The specialization rules for underlined and dash-underlined nil expressions return a residual FCL nil expression and a tuple containing a residual FCL nil expression and a nil location respectively.

6.6.11 Specialization of operations

Definition 6.17 *Let op be an operator in an expression, v_1 and v_2 be values, and σ be the current store. Then, the specialization of operation op between v_1 and v_2 is a value v iff:*

$$\sigma, op \vdash_{op}^{spec} \langle v_1 \ , \ v_2 \rangle \rightarrow v$$

where the relation $\vdash_{op}^{spec}$ is defined in figure 6.38 and figure 6.39.

The specialization rules for an operation are textually equal with their evaluation rule

Operation specialization for $op \in \{+, -, *, /\}$

$$\frac{}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{nat}(n_1)\ ,\ \mathsf{nat}(n_2)\rangle \rightarrow \mathsf{nat}(\llbracket op \rrbracket\ n_1\ n_2)} \quad \text{if } op \in \{+, -, *, /\}$$

Operation specialization for $op \in \{>, \texttt{gte}, <, \texttt{lte}\}$

$$\frac{\mathit{true\text{-}value}(\llbracket op \rrbracket\ n_1\ n_2)}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{nat}(n_1)\ ,\ \mathsf{nat}(n_2)\rangle \rightarrow \mathsf{nat}(1)} \quad \text{if } op \in \{>, \texttt{gte}, <, \texttt{lte}\}$$

$$\frac{\mathit{false\text{-}value}(\llbracket op \rrbracket\ n_1\ n_2)}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{nat}(n_1)\ ,\ \mathsf{nat}(n_2)\rangle \rightarrow \mathsf{nat}(0)} \quad \text{if } op \in \{>, \texttt{gte}, <, \texttt{lte}\}$$

Figure 6.38: Specialization rules for operations (*part 1*)

counterparts.

6.6.12 Specialization of assignments

Definition 6.18 *Let $\hat{a}$ be an annotated assignment, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, ε be an environment defined for all variable names in the variable declarations list, σ be the current store, $\mathcal{D}$ be a residual FCL program variable declarations list, $\mathcal{A}$ be a residual FCL program assignments list, and h be a next available location. Then, the specialization of the assignment $\hat{a}$ is a a new store σ', a possibly modified residual FCL program variable declarations list, a possibly modified residual FCL program assignments list, and a new next available location h' iff:*

$$\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, \mathcal{A}, h \vdash_{assign}^{spec} \hat{a} \Rightarrow \langle \sigma', \mathcal{D}', \mathcal{A}', h' \rangle$$

where the relation $\vdash_{assign}^{spec}$ is defined in figure 6.40.

The first rule in the figure is the specialization rule for single non-underlined assignments. The rule is similar to the evaluation rule for an assignment from the previous chapter.

The second rule is the specialization rule for single non-underlined assignments. First, we specialize the annotated expression $\hat{e}$ and the annotated left-expression $\widehat{le}$ yielding a residual FCL expression e and a residual FCL left-expression le respectively. Then, using these FCL expressions, we generate a residual FCL assignment and append it into the

Operation specialization for $op \in \{=, \texttt{neq}\}$

Nat

$$\frac{}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{nat}(n)\ ,\ \mathsf{nat}(n)\rangle \rightarrow \mathsf{nat}(1)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{nat}(n_1)\ ,\ \mathsf{nat}(n_2)\rangle \rightarrow \mathsf{nat}(0)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

Loc

$$\frac{}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{loc}(h)\ ,\ \mathsf{loc}(h)\rangle \rightarrow \mathsf{nat}(1)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{loc}(h_1)\ ,\ \mathsf{loc}(h_2)\rangle \rightarrow \mathsf{nat}(0)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

Array

$$\frac{\forall i \in \{n_1, \ldots, n_2\}\ .\ \sigma, op \vdash_{op}^{spec} \langle \sigma(\alpha_1(i))\ ,\ \sigma(\alpha_2(i))\rangle \rightarrow \mathsf{nat}(1)}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{array}(\alpha_1, n_1, n_2)\ ,\ \mathsf{array}(\alpha_2, n_1, n_2)\rangle \rightarrow \mathsf{nat}(1)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{\exists i \in \{n_1, \ldots, n_2\}\ .\ \sigma, op \vdash_{op}^{spec} \langle \sigma(\alpha_1(i))\ ,\ \sigma(\alpha_2(i))\rangle \rightarrow \mathsf{nat}(0)}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{array}(\alpha_1, n_1, n_2)\ ,\ \mathsf{array}(\alpha_2, n_1, n_2)\rangle \rightarrow \mathsf{nat}(0)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

Record

$$\frac{\forall i \in \{1, \ldots, n\}\ .\ \sigma, op \vdash_{op}^{spec} \langle \sigma(\rho_1(x_i))\ ,\ \sigma(\rho_2(x_i))\rangle \rightarrow \mathsf{nat}(1)}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{record}(\rho_1)\ ,\ \mathsf{record}(\rho_2)\rangle \rightarrow \mathsf{nat}(1)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

$$\frac{\exists i \in \{1, \ldots, n\}\ .\ \sigma, op \vdash_{op}^{spec} \langle \sigma(\rho_1(x_i))\ ,\ \sigma(\rho_2(x_i))\rangle \rightarrow \mathsf{nat}(0)}{\sigma, op \vdash_{op}^{spec} \langle \mathsf{record}(\rho_1)\ ,\ \mathsf{record}(\rho_2)\rangle \rightarrow \mathsf{nat}(0)} \quad \text{if } op \in \{=, \texttt{neq}\}$$

Figure 6.39: Specialization rules for operations (*part 2*)

residual FCL program assignments list $\mathcal{A}$.

The specialization of single dash-underlined assignments is a combination of the non-underlined and the underlined assignment specializations. We update the value like in the first rule and residualize an FCL assignment like in the second rule.

The fourth and the fifth rules is the specialization rules for an empty assignment and a sequence of assignments.

6.6.13 Specialization of assignment-updates

Definition 6.19 *Let h be a location, w be a PE-value, σ be the current store, and $\mathcal{A}$ be a residual FCL program assignments list. Then, the meaning of the specialization rules for*

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle v,\sigma',\mathcal{D}',h'\rangle \quad \bar{\tau},\varepsilon,\sigma',\mathcal{D}',h' \vdash^{spec}_{lexp} \widehat{le} \Rightarrow \langle h_l,\sigma'',\mathcal{D}'',h''\rangle \quad \sigma'',\mathcal{A} \vdash^{spec}_{update} \langle h_l\ ,\ v\rangle \rightarrow \langle \sigma''',\mathcal{A}'\rangle}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},\mathcal{A},h \vdash^{spec}_{assign} (\widehat{le} := \hat{e}) \Rightarrow \langle \sigma''',\mathcal{D}'',\mathcal{A}',h''\rangle}$$

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle e,\sigma',\mathcal{D}',h'\rangle \quad \bar{\tau},\varepsilon,\sigma',\mathcal{D}',h' \vdash^{spec}_{lexp} \widehat{le} \Rightarrow \langle le,\sigma'',\mathcal{D}'',h''\rangle}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},\mathcal{A},h \vdash^{spec}_{assign} \underline{(\widehat{le} := \hat{e})} \Rightarrow \langle \sigma'',\mathcal{D}'',\mathcal{A}\triangle(le := e),h''\rangle}$$

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle\langle e,v\rangle,\sigma',\mathcal{D}',h'\rangle \quad \bar{\tau},\varepsilon,\sigma',\mathcal{D}',h' \vdash^{spec}_{lexp} \widehat{le} \Rightarrow \langle\langle le,h_l\rangle,\sigma'',\mathcal{D}'',h''\rangle \quad \sigma'',\mathcal{A} \vdash^{spec}_{update} \langle h_l\ ,\ v\rangle \rightarrow \langle \sigma''',\mathcal{A}'\rangle}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},\mathcal{A},h \vdash^{spec}_{assign} \underline{(\widehat{le} := \hat{e})} \Rightarrow \langle \sigma''',\mathcal{D}'',\mathcal{A}'\triangle(le := e),h''\rangle}$$

$$\frac{}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},\mathcal{A},h \vdash^{spec}_{assigns} () \Rightarrow \langle \sigma',\mathcal{D}',\mathcal{A}',h'\rangle}$$

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},\mathcal{A},h \vdash^{spec}_{assign} \hat{a} \Rightarrow \langle \sigma',\mathcal{D}',\mathcal{A}',h'\rangle \quad \bar{\tau},\varepsilon,\sigma',\mathcal{D}',\mathcal{A}',h' \vdash^{spec}_{assigns} (\hat{a}^*) \Rightarrow \langle \sigma'',\mathcal{D}'',\mathcal{A}'',h''\rangle}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},\mathcal{A},h \vdash^{spec}_{assigns} (\hat{a}\ \hat{a}^*) \Rightarrow \langle \sigma'',\mathcal{D}'',\mathcal{A}'',h''\rangle}$$

Figure 6.40: Specialization rules for assignments

the assignment-updates for h and w is a new store σ', and a possibly modified residual FCL program assignments list iff:

$$\sigma,\mathcal{A} \vdash^{spec}_{update} \langle h\ ,\ w\rangle \rightarrow \langle \sigma',\mathcal{A}'\rangle$$

where the relation $\vdash^{spec}_{update}$ is defined in figure 6.41 and figure 6.42.

The first rule for assignment-updates covers the case where $w \in$ Variable. In this case, we residualize an FCL assignment with the variable extracted from the store for the location h as its left-expression and the variable x as its (right) expression.

The second rule covers the case where $w \in$ Nat. In this case, we simply update the store σ associated with the location h with the nat value n.

If $w \in$ Loc, then we have to consider two different cases. The first case is applied when the store value associated with location h is another location. In this case, we update the store σ associated with the location h with the PE-value w.

The second case is applied when the store value associated with location h is a tuple containing a variable x and a location. In this case, we update the store σ associated with

Variable

$$\frac{\sigma(h) = x'}{\sigma, \mathcal{A} \vdash^{spec}_{update} \langle h\ ,\ x\rangle \rightarrow \langle \sigma, (x' := x) :: \mathcal{A}\rangle}$$

Nat

$$\frac{}{\sigma, \mathcal{A} \vdash^{spec}_{update} \langle h\ ,\ \mathsf{nat}(n)\rangle \rightarrow \langle \sigma[h \mapsto \mathsf{nat}(n)], \mathcal{A}\rangle}$$

Loc

$$\frac{\sigma(h) = \mathsf{loc}(h'')}{\sigma, \mathcal{A} \vdash^{spec}_{update} \langle h\ ,\ \mathsf{loc}(h')\rangle \rightarrow \langle \sigma[h \mapsto \mathsf{loc}(h')], \mathcal{A}\rangle}$$

$$\frac{\sigma(h) = \langle x, \mathsf{loc}(h'')\rangle}{\sigma, \mathcal{A} \vdash^{spec}_{update} \langle h\ ,\ \mathsf{loc}(h')\rangle \rightarrow \langle \sigma[h \mapsto \langle x, \mathsf{loc}(h')\rangle], \mathcal{A}\rangle}$$

$$\frac{\sigma(h) = \langle x', \mathsf{loc}(h'')\rangle}{\sigma, \mathcal{A} \vdash^{spec}_{update} \langle h\ ,\ \langle x, \mathsf{loc}(h')\rangle\rangle \rightarrow \langle \sigma[h \mapsto \langle x', \mathsf{loc}(h')\rangle, (x' := x) :: \mathcal{A}]\rangle}$$

Figure 6.41: Specialization rules for assignment-updates (*part 1)*

the location h with a tuple containing the variable x and the PE-value w.

The last rule in figure 6.41 is defined when the PE-value w is a tuple containing a variable and a location. This rule is evaluated like the previous rule. Addition to that, we generate a new FCL assignment and append it into the residual FCL program assignments list $\mathcal{A}$.

The specialization rules for assignment-updates for array and record are actually similar in style to the previous rules for location. The difference is in these rules we are also update their members. These rules can be seen in figure 6.42.

6.6.14 Specialization of jumps

Definition 6.20 *Let $\hat{j}$ be an annotated jump, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, ε be an environment defined for all variable names in the variable declarations list, σ be the current store, $\mathcal{D}$ be a residual FCL program variable declarations list, and h be a next available location. Then, the specialization of the jump $\hat{j}$ yields a set of labels $\{l_1, \ldots, l_n\}$, a residual FCL jump j, a new store σ', a possibly modified residual FCL program variable declarations list, and a new next available location h' iff:*

$$\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{jump} \hat{j} \Rightarrow \langle \{l_1, \ldots, l_n\}, j, \sigma', \mathcal{D}', h'\rangle$$

where the relation $\vdash^{spec}_{jump}$ is defined in figure 6.43.

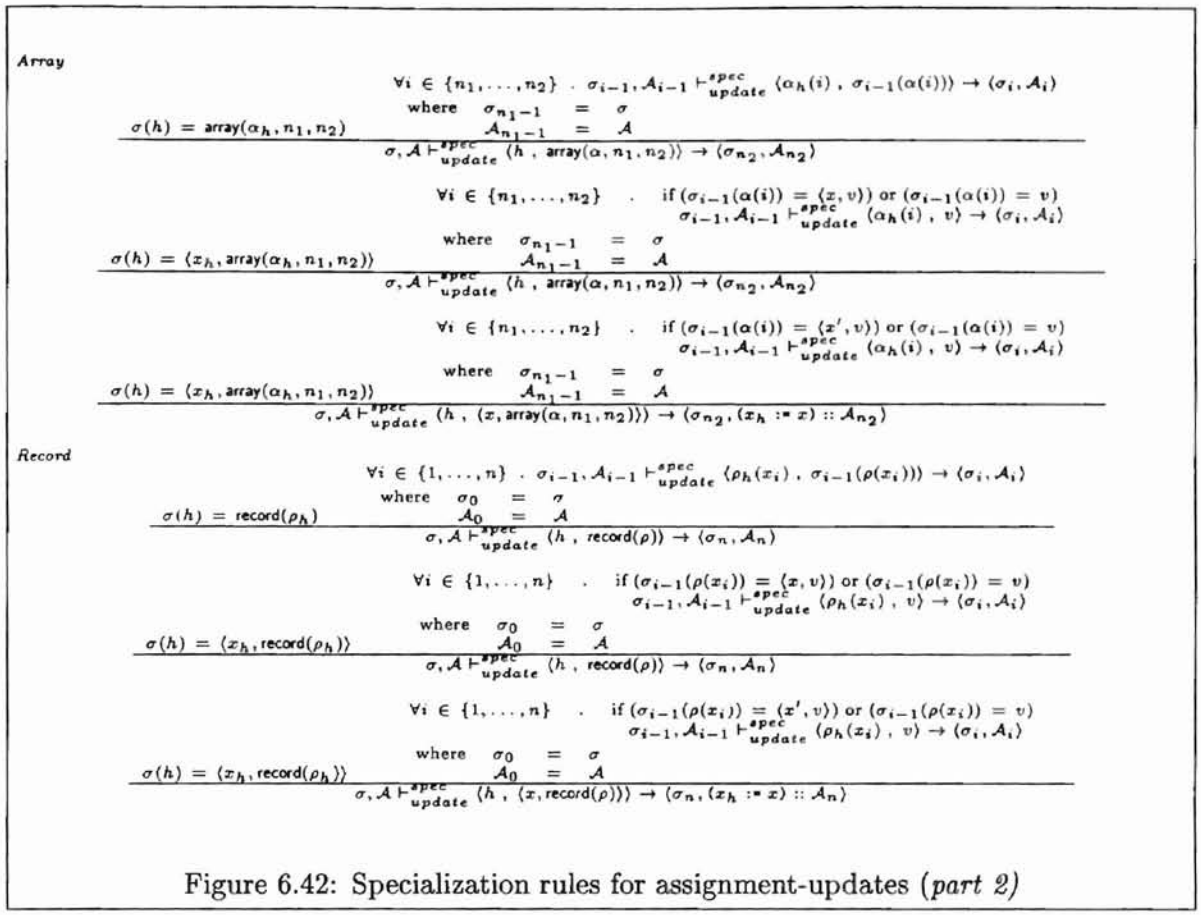


Figure 6.42: Specialization rules for assignment-updates *(part 2)*

A goto jump specialization returns its label l and an FCL goto jump that point to a new version of the block labeled l that is specialized with respect to the current store σ.

The specialization rules for if jumps are defined for three different conditions. If the result of specializing the annotated expression $\hat{e}$ is a nat value and the test result is true, then the label l_1 (label for the true branch) and a new residual FCL goto jump that point to a version of the block labeled l that is specialized with respect to the current store σ are returned. Similarly, if the test result is false, but instead of l_1, we use the label l_2 (label for the false branch). If the result of specializing the annotated expression $\hat{e}$ is an FCL expression e, then the if jump cannot be computed at specialization time. We return a set containing label l_1 and label l_2 together with a new residual FCL if jump.

The specialization rule for return jumps yields the special label halt and a new residual FCL return jump.

Goto

$$\frac{}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{jump} (\texttt{goto}\ l) \Rightarrow \langle \{l\}, (\texttt{goto}\ (l, \sigma)), \sigma, \mathcal{D}, h \rangle}$$

If

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle \mathsf{nat}(n), \sigma', \mathcal{D}', h' \rangle \qquad \textit{true-value}(n)}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{jump} (\texttt{if}\ \hat{e}\ l_1\ l_2) \Rightarrow \langle \{l_1\}, (\texttt{goto}\ (l_1, \sigma')), \sigma', \mathcal{D}', h' \rangle}$$

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle \mathsf{nat}(n), \sigma', \mathcal{D}', h' \rangle \qquad \textit{false-value}(n)}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}. h \vdash^{spec}_{jump} (\texttt{if}\ \hat{e}\ l_1\ l_2) \Rightarrow \langle \{l_2\}, (\texttt{goto}\ (l_2, \sigma')), \sigma', \mathcal{D}', h' \rangle}$$

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle e, \sigma', \mathcal{D}', h' \rangle}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{jump} \underline{(}\ \underline{\texttt{if}}\ \hat{e}\ \underline{\texttt{then}}\ l_1\ \underline{\texttt{else}}\ l_2\ \underline{)} \Rightarrow \langle \{l_1, l_2\}, (\texttt{if}\ e\ (l_1, \sigma')\ (l_2, \sigma')), \sigma', \mathcal{D}', h' \rangle}$$

Return

$$\frac{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{exp} \hat{e} \Rightarrow \langle e, \sigma', \mathcal{D}', h' \rangle}{\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{jump} \underline{(\texttt{return}\ \hat{e})} \Rightarrow \langle \{\mathsf{halt}\}, (\texttt{return}\ e), \sigma', \mathcal{D}', h' \rangle}$$

Figure 6.43: Specialization rules for jumps

6.6.15 Specialization of blocks

Definition 6.21 *Let $\hat{b}$ be a block in the annotated program, $\bar{\tau}$ be a type environment defined for all type names in the type definitions list, ε be an environment defined for all variable names in the variable declarations list, σ be the current store, $\mathcal{D}$ be a residual FCL program variable declarations list, and h be a next available location. Then, the specialization of the block $\hat{b}$ is a set of labels $\{l_1, \ldots, l_n\}$, a residual FCL program block b, a possibly modified residual FCL program variable declarations list, and a new next available location h' iff:*

$$\bar{\tau}, \varepsilon, \sigma, \mathcal{D}, h \vdash^{spec}_{block} \hat{b} \Rightarrow \langle \{l_1, \ldots, l_n\}, b, \sigma', \mathcal{D}', h' \rangle$$

where the relation $\vdash^{spec}_{jump}$ is defined in figure 6.44.

In our application, we do the *transition compression* on the fly where we compress transitions whenever the jump is a goto or a non-underlined if jump. When we specialize a block, every time we encounter these jumps, we immediately process the destination block and merge the results. This process continue until we encounter a return jump or an underlined if jump.

The first rule is the specialization rule for single blocks. It is defined if the annotated jump $\hat{j}$ specialization result is not a goto jump. This means that the annotated jump is

Block

$$\frac{\begin{array}{l}\bar{\tau},\varepsilon,\sigma,\mathcal{D},\mathcal{A},h \vdash^{spec}_{assigns} (\hat{a}^*) \Rightarrow \langle\sigma',\mathcal{D}',\mathcal{A}',h'\rangle \\ \quad \text{where } \mathcal{A} = \text{empty Assignment-list} \\ \bar{\tau},\varepsilon,\sigma',\mathcal{D}',h' \vdash^{spec}_{jump} \hat{j} \Rightarrow \langle\{l'_i \mid i \in \{1,\ldots,n\}\}, j, \sigma'', \mathcal{D}'', h''\rangle\end{array}}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{block} (l\ (\hat{a}^*)\ \hat{j}) \Rightarrow \langle\{l'_i \mid i \in \{1,\ldots,n\}\}, ((l,\sigma)\ \mathcal{A}'\ j), \sigma'', \mathcal{D}'', h''\rangle} \quad \text{if } j \neq (\text{goto } l')$$

$$\frac{\begin{array}{l}\bar{\tau},\varepsilon,\sigma,\mathcal{D},\mathcal{A},h \vdash^{spec}_{assigns} (\hat{a}^*) \Rightarrow \langle\sigma',\mathcal{D}',\mathcal{A}',h'\rangle \\ \quad \text{where } \mathcal{A} = \text{empty Assignment-list} \\ \bar{\tau},\varepsilon,\sigma',\mathcal{D}',h' \vdash^{spec}_{jump} \hat{j} \Rightarrow \langle\{l'\}, (\text{goto } l'), \sigma'', \mathcal{D}'', h''\rangle \\ \bar{\tau},\varepsilon,\sigma'',\mathcal{D}'',h'' \vdash^{spec}_{block} \Gamma(l') \Rightarrow \langle\{l''_i \mid i \in \{1,\ldots,n\}\}, ((l',\sigma'')\ \mathcal{A}''\ j), \sigma''', \mathcal{D}''', h'''\rangle\end{array}}{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{block} (l\ (\hat{a}^*)\ \hat{j}) \Rightarrow \langle\{l''_i \mid i \in \{1,\ldots,n\}\}, ((l,\sigma)\ \mathcal{A}' +\!\!+ \mathcal{A}''\ j), \sigma''', \mathcal{D}''', h'''\rangle}$$

Blocks transition

$$\frac{\bar{\tau},\varepsilon,\sigma,\mathcal{D},h \vdash^{spec}_{block} \Gamma(l) \Rightarrow \langle\{l'_i \mid i \in \{1,\ldots,n\}\}, b, \sigma', \mathcal{D}', h'\rangle}{\vdash_\Gamma \langle(l,\sigma) :: \mathcal{P}, \mathcal{S}, \mathcal{B}, \sigma, h\rangle \rightarrow \langle\text{remove-halt}((l'_1,\sigma') :: \ldots :: (l'_n,\sigma')) :: \mathcal{P}, \mathcal{S} \cup (l,\sigma), \mathcal{B}\Delta b, \sigma', \mathcal{D}', h'\rangle} \quad \text{if } (l,\sigma) \notin \mathcal{S}$$

$$\frac{}{\vdash_\Gamma \langle(l,\sigma) :: \mathcal{P}, \mathcal{S}, \mathcal{B}, \sigma, h\rangle \rightarrow \langle\mathcal{P}, \mathcal{S}, \mathcal{B}, \sigma, h\rangle} \quad \text{if } (l,\sigma) \in \mathcal{S}$$

Figure 6.44: Specialization rules for blocks

either an underlined if or a return jump. The rule returns a set of labels to execute and a specialized FCL block corresponded to this block.

The second rule is the case where the result is a goto jump which means that the annotated jump is either a goto or a non-underlined if jump. In this case, after specializing the assignment and the jump, we continue the specialization process with the destination block. The rule returns a set of labels to execute from the destination block and a specialized FCL block corresponded to this block and the destination block.

In the blocks transition rules, we use three new data structures as follow :

- a seen-before list $\mathcal{S}$ is a list containing the configurations that have already been processed,
- a pending list $\mathcal{P}$ is a list containing the configurations that are pending to be process and
- $\mathcal{B}$ is a list of residual FCL program blocks.

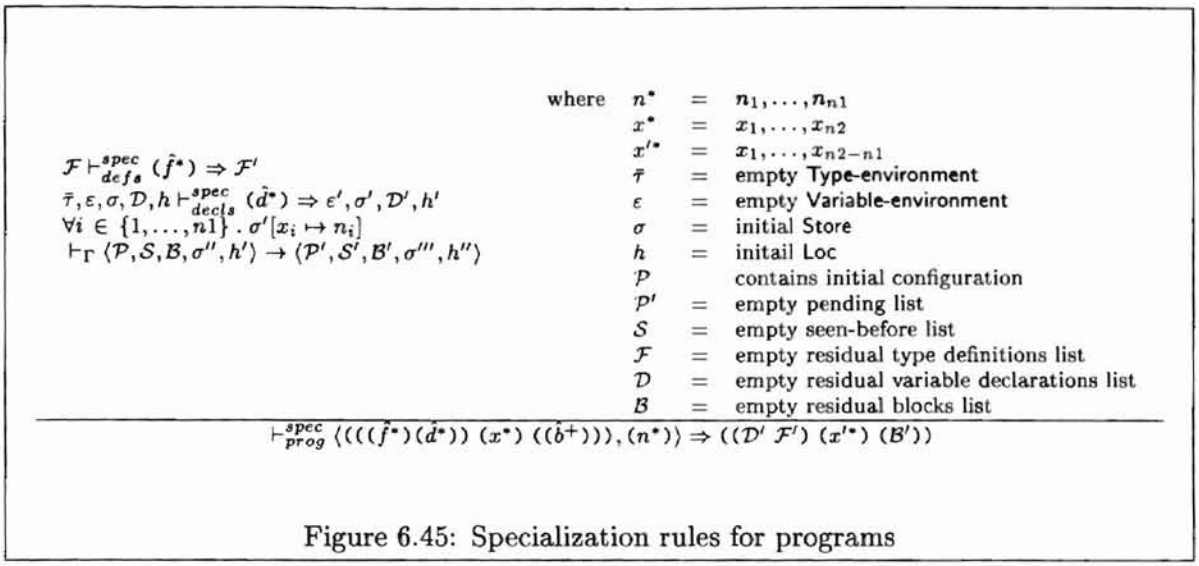


Figure 6.45: Specialization rules for programs

The specialization begins with an empty seen-before list and an empty pending list. While the pending list is not empty, we repeatedly pull the configuration off the pending list, create a specialized block for that configuration, append this block into the list of residual FCL program blocks and get a new configuration from the pending list.

The first rule for blocks transition covers the case where the next item in the pending list is not in the seen-before list (has not been processed before). In this case, the configuration state (l, σ) is removed from the pending list and the block labeled l is specialized with respect to the current store σ using of the specialization rules for a block. Destination configuration states (l'_i, σ'), where any label halt has been removed using function remove-halt, are appended into the pending list, the processed block configuration state (l, σ) is added into the seen-before list and the residual block b is inserted at the end of the residual FCL program blocks list.

The second rule is applied if the next item in the pending list is in the seen-before list (has been processed before). In this case, we simply remove this item from the pending list $\mathcal{P}$.

6.6.16 Specialization of programs

Definition 6.22 *Let $\hat{p}$ be an annotated program and (n^*) is a list of natural number represent program arguments. Then, the specialization of the annotated $\hat{p}$ is a residual FCL program p iff:*

$$\vdash^{spec}_{prog} \langle \hat{p}, (n^*) \rangle \Rightarrow p$$

where the relation $\vdash^{spec}_{prog}$ is defined in figure 6.45.

To specialize a program, we have to specialize the type definitions list which will return the residual type definitions list $(\hat{f}^*)$. Then, we specialize the variable declarations list and update the specialization result σ' for all static and eliminable variable arguments with their values into σ''.

Finally, using the blocks transition rules we specialize the programs blocks in the pending list where the initial pending list $\mathcal{P}$ contains the initial configuration with the label from the first block and the initial store σ''. The result of specializing the program is a residual FCL program where x'^* are dynamic and residual variable arguments from the original variable arguments x^*

6.7 Implementation and Specialization examples

Both of the interpreter and the specializer that we discussed before are implemented using Scheme language. We choose this language because it is a very good language for prototyping and it is easy to use.

The interpreter is divided into three different modules: parser, type-checker, and evaluator. The source code of these modules are listed in appendix A. The specializer is divided into four different modules: parser, binding-time type checker, specializer, and unparser, and their source code can be seen in appendix B.

We have tested our system with fragments from the replicated workers framework (RWF) from chapter 4, specifically the RWF structure and task types fragment in figure 4.2 and

```
(((record WorkerInfo
          ((c : (access (record CollectionInfo) S E))
           (aw : (access (record ActiveWorker) S E))
           (ap : (access (record ActivePool) S E))
           (numIn : (nat D R)))
          S R)

  (record CollectionInfo
          ((workers : (array 1 3 (access (record WorkerInfo) S E) S E))
           (pool : (access (record ActivePool) S E))
           (done : (nat D R)))
          S R)

  (record ActiveWorker
          ((StartUp : (nat D R))
           (ShutDown : (nat D R))
           (Execute : (nat D R)))
          D R)

  (record ActivePool
          ((StartUp : (nat D R))
           (ShutDown : (nat D R))
           (Get : (nat D R))
           (Put : (nat D R))
           (GetResult : (nat D R))
           (PutResult : (nat D R))
           (Finished : (nat D R))
           (Execute : (nat D R))
           (Complete : (nat D R)))
          D R))

 ((numWorkers : (nat S E))
  (numIn : (nat D R))
  (c : (access (record CollectionInfo) S E))
  (i : (nat S E))))

(numWorkers numIn)
 ...
```

Figure 6.46: The original FCL-ann fragment of the RWF structure and task types

the RWF function `Create` fragment in figure 4.5.

The first step for specializing those fragments is to convert them into FCL-ann language. Since flowchart language is a very simple language in contrast to Ada, we have to simulate some Ada contructs that not exist in the FCL. In the RWF structure and task types fragment, we simulate Ada task type constructs (*i.e.* `ActiveWorker` and `ActivePool`) as dynamic and residual record types and their entries (*e.g.* `StartUp`, `ShutDown`, *etc.*) as dynamic and residual nat variables.

The FCL-ann fragment of the RWF structure and task types is shown in figure 6.46. In the figure, we also can see that `WorkerInfo` and `CollectionInfo` are converted to static and

```
((((record workerinfo
           ((numin : nat)))

   (record collectioninfo
           ((done : nat)))

   (record activeworker
           ((startup : nat)
            (shutdown : nat)
            (execute : nat)))

   (record activepool
           ((startup : nat)
            (shutdown : nat)
            (get : nat)
            (put : nat)
            (getresult : nat)
            (putresult : nat)
            (finished : nat)
            (execute : nat)
            (complete : nat))))

  ((loc-2 : nat)
   (loc-5 : (record collectioninfo))
   (loc-12 : (record activepool))
   (loc-13 : (record workerinfo))
   (loc-18 : (record activeworker))
   (loc-19 : (record workerinfo))
   (loc-24 : (record activeworker))
   (loc-25 : (record workerinfo))
   (loc-30 : (record activeworker))))

 (loc-2)
 ...
```

Figure 6.47: The specialized FCL fragment of the RWF structure and task types

residual records where the cyclical entry calls between these two structures are annotated as static and eliminable pointers.

We specialized the fragment in figure 6.46 with respect to `numIn = 3` and the specialization result can be seen in figure 6.47.

In the figure, we can see that the record `WorkerInfo` and the record `CollectionInfo` have been specialized into new narrowed structures which contain only residual members, `numIn` for `WorkerInfo` and `done` for `CollectionInfo`.

Variables `loc-13`, `loc-19`, and `loc-25` are static forms of record `WorkerInfo` correspond to three dynamic allocations of the record when the specializer executes the orginal fragmen of function `Create` in figure 6.48. Also, variables `loc-18`, `loc-24`, and `loc-30` are static forms of record `ActiveWorker`, `loc-5` is a static form of record `CollectionInfo`, and

```
((create ((c := (new (record CollectionInfo) S E))
          ((recmem (deref c) done) := 1)
          ((recmem (deref c) pool) := (new (record ActivePool) S E))
          (i := 1))
         (goto build-test))
 (build-test ()
             (if (> i 3) cont build-loop))
 (build-loop ((([] (recmem (deref c) workers) i) := (new (record WorkerInfo) S E))
              ((recmem (deref ([] (recmem (deref c) workers) i)) c) := c)
              ((recmem (deref ([] (recmem (deref c) workers) i)) aw) :=
               (new (record ActiveWorker) S E))
              ((recmem (deref ([] (recmem (deref c) workers) i)) ap) := (recmem (deref c) pool))
              ((recmem (deref ([] (recmem (deref c) workers) i)) numIn) := numIn)
              (i := (+ i 1)))
             (goto build-test))
 (cont (((recmem (deref (recmem (deref c) pool)) StartUp) := 999)
        (i := 1))
       (goto ref-test))
 (ref-test ()
           (if (> i 3) done ref-loop))
 (ref-loop (((recmem (deref (recmem (deref ([] (recmem (deref c) workers) i)) aw)) StartUp) := 999)
            (i := (+ i 1)))
           (goto ref-test))
 (done ()
       ...
```

Figure 6.48: The original FCL-ann fragment of the RWF function `Create`

`loc-12` is a static form of record `ActivePool`.

The result of the specialization is as expected where records `WorkerInfor` and `CollectionInfo` are narrowed into new records and dynamically allocated objects are transformed into static forms.

Together with the fragment above, we also specialized the function `Create` from the left side of figure 4.5. We convert this fragment to FCL-ann language like the RWF structure and task types fragment. The FCL-ann version of function `Create` is shown in figure 6.48.

In this fragment, we have to simulate Ada boolean true values as constant `1` and Ada task type entry calls as FCL-ann assignment with the task entry as the left-expression and constant `999` as the assignment right-expression. Also, we desugar the loop in the form of `for i in 1..numWorkers loop ... end loop;` into blocks `build-test` and `build-loop`.

The result of specializing this fragment with `numIn` = 3 is shown in figure 6.49. In this figure, we can see that the loop above has been unrolled and chains of dynamically created references (*e.g.* the chain that leads from an `ActivePool` through a `WorkerInfo` record,

```
((lab-1 (((recmem loc-5 done) := 1)

        ((recmem loc-13 numin) := loc-2)
        ((recmem loc-19 numin) := loc-2)
        ((recmem loc-25 numin) := loc-2)

        ((recmem loc-12 startup) := 999)

        ((recmem loc-18 startup) := 999)
        ((recmem loc-24 startup) := 999)
        ((recmem loc-30 startup) := 999))
        ...
```

Figure 6.49: The specialized FCL fragment of the RWF function **Create**

to the **ActiveWorker** task) have been resolved, yielding the static name of the entity (*e.g.* `loc-13` and `loc-18`).

From both specialized fragments, we can see that dynamism and indirect reference in the RWF that would inhibit FSV application has been removed.

CHAPTER 7

CONCLUSION

7.1 Summary

In this thesis, we have addressed problem with dynamism in the finite state verification tools for the configurable system. The presence of of dynamism in the system makes it difficult to apply current finite state verification techniques to such system.

We attacked the problem using partial evaluation techniques, especially the offline method. We designed and implemented an offline partial evaluator prototype to eliminate dynamic allocated objects and convert them into static allocated objects. The implementation can be separated from the existing FSV tools which can significantly extend their applicability.

We developed a new approach in offline partial evaluation using *three level language* which we believe can be useful in implementing partial evaluation in complex languages that involved function calls.

7.2 Assessments

Unfortunaly, not all dynamism can be removed using our approach. When a system is allocating, dealocating and reorganizing its object structure as it executes, then the specialization will not be effective.

One way to solve this problem is have the model checker support allocation, deallocation, and garbage collection of objects.

7.3 Future work

In this thesis, we only implemented the specializer for a small language (*i.e.* FCL language). Even though this language is sufficient to illustrate the idea of removing dynamism by specialization, the language lacks features that exist in modern languages like Ada or Java. So, the next step is to extend the language to include features from modern languages (*e.g.* exceptions and object inheritance).

Also, the experimentation should be broadened to include more examples. The example of the replicated workers framework that we have in this thesis is just one of many existing examples that can be solved using partial evaluation.

BIBLIOGRAPHY

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1994. DIKU Report 94-19.

[2] G.R. Andrews. *Concurrent Programming: Principles and Practice.* Addison-Wesley, 1991.

[3] Romana Baier, Robert Glück, and Robert Zöchling. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, volume 94/9 of *Technical Report*, pages 119–132. University of Melbourne, Australia, 1994.

[4] Romana Baier, Robert Glück, and Robert Zöchling. Specialization of numerical programs with the fspec system. In W. Mackens and S.M. Rump, editors, *Software Engineering im Scientific Computing*, pages 86–90. Vieweg, 1996.

[5] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1990. DIKU Report 90-17.

[6] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[7] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[8] T. Cattel. Process control design using spin. In *Proceedings of the First SPIN Workshop*, October 1995.

[9] A. Cimatti, F. Giunchiglia, G. Mongardi, F. Torielli, and P. Traverso. Model checking safety critical software with spin: an application to a railway interlocking system. In *Proceedings of the Third SPIN Workshop*, April 1997.

[10] E.M. Clarke and E.A. Emerson. *Systhesis of Synchronization Skeletons for Branching Time Temporal Logic. In Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

[11] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[12] C. Consel. New insights into partial evaluation: The schism experiment. In Andreas Paepcke, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, March 1988. (Lecture Notes in Computer Science, vol. 300*, pages 236–246. Springer-Verlag, 1988.

[13] C. Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 264–272. ACM, 1990.

[14] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J.Noyë, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, 1998. Special issue devoted to the 1998 Symposium on Partial Evaluation (to appear).

[15] C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volanschi. A uniform approach for compile-time and run-time specialization. In *Proceeding of the 1996 International Seminar on Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 54–72, Dagstuhl Castle, Germany, February 1996.

[16] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[17] M.B. Dwyer, V. Carr, and L. Hines. Model checking graphical user interfaces using abstractions. In *LNCS 1301*, pages 244–261. The 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 1997.

[18] M.B. Dwyer and L.A. Clarke. Data flow analysis for verifying properties of concurrent programs. *Software Engineering Notes*, 19(5):62–75, December 1994. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[19] M.B. Dwyer, C.S. Pasareanu, and J.C. Corbett. Translating ada programs for model checking: A tutorial. Technical report 98-12, Kansas State University, Department of Computing and Information Sciences, 1998.

[20] W.M. Elseiady, R. Cleaveland, and J.W. Baugh Jr. Modeling and verifying active structural control systems. *Science of Computer Programming*, 29(1-2):99–122, July 1997.

[21] E.A. Emerson and J.Y. Halpern. "sometimes" and "not never" revisited: On branching time versus linear time. *Journal of ACM*, 33:151–178, 1986.

[22] Z. Har'El and R. P. Kurshan. Software for analytical devleopment of communication protocols. *AT&T Technical Journal*, 69(1):44–59, 1990.

[23] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[24] G.E. Hughes and M.J. Creswel. *Introduction to Modal Logic.* Methuen, London, 1977.

[25] M. Huth and M. Ryan. *Logic and Its Practical Applications in Computer Science.* Lecture Notes in Computer Science, DRAFT. Kansas State University and School of Computer Science, University of Birmingham, UK, 1997.

[26] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[27] J. Kramer and J. Magee. The evolving philosophers problem : Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[28] Shawn Laubach. Abstraction based program specialization. Master's thesis, Oklahoma State University, November 1998.

[29] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1991.

[30] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[31] G.N. Naumovich, L.A. Clarke, and L.J. Osterweil. Verification of communication protocols using data flow analysis. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1996.

[32] G.H. Nickel. Cellular automaton rules for solving the milne problem. *Physics Letters A*, 133(4,5):219–224, November 1988.

[33] F. Nielson and H.R. Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

[34] Erik Ruf and Daniel Weise. Using types to avoid redundant specialization. In *Proceedings of the 1991 Symposium on Partial Evaluation and Semantic-Directed Program Manipulation*, pages 321–333. ACM SIGPLAN, June 1991.

[35] D.A. Schmidt. *Denotational Semantics : A Methodology for Language Development.* Wm. C. Brown Publishers, 1986.

[36] Ravi Sethi. *Programming Languages: Concepts and Constructs.* Addison-Wesley Publishing Company, 1989.

[37] Daniel Weise, Roland Conybeare, Erik Ruf, , and Scott Seligman. Automatic online partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture.* ACM, 1991.

[38] Daniel Weise and Erik Ruf. Computing types during partial evaluation. Technical Report Fuse-Memo 91-3 revised, Stanford University, December 1988.

[39] J.M. Wing and M. Vaziri-Farahani. Model checking software systems: A case study. *Software Engineering Notes*, 20(4):128–139, October 1995. Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering.

APPENDIX A

INTERPRETER

A.1 Module for syntax

```
;;=================================================================
;; int-syntax.scm
;; syntax definition for interpreter
;;=================================================================

(require 'struct)

;;=================================================================
;; concrete syntax
;;=================================================================

;; <program>     ::= ((((<type-def>*)(<decl>*))(<var>*)(<block>+))
;; <type-def>    ::= (record <var> (<decl>+))
;; <decl>        ::= (<var> : <type>)
;; <type>        ::= nat
;;                 | (access <type>)
;;                 | (array <nat> <nat> <type>)
;;                 | (record <var>)
;; <block>       ::= (<label> (<assignment>*) <jump>)
;; <assignment>  ::= (<lexp> := <exp>)
;; <lexp>        ::= <var>
;;                 | (deref <exp>)
;;                 | ([] <exp> <exp>)
;;                 | (recmem <exp> <var>)
;; <exp>         ::= <nat>
;;                 | <var>
;;                 | (<op> <exps>)
;;                 | (deref <exp>)
;;                 | (new <type>)
;;                 | (nil <type>)
;;                 | ([] <exp> <exp>)
;;                 | (recmem <exp> <var>)
;; <jump>        ::= (goto <label>)
;;                 | (if <exp> <label> <label>)
;;                 | (return <exp>)
;; <op>          ::= + | - | * | / | = | > | gte | < | lte | neq
;; <nat>         ::= 0 | 1 | 2 | ...
;; <label>       ::= any symbol
;; <var>         ::= any alphanumeric symbol not used as a command,
;;                   number, and later lists and symbols

;;=================================================================
;; abstract syntax
;;=================================================================

(define-record program     (decls params blocks))
(define-record decls       (tdefs vdecls))
(define-record tdef        (name vdecls))
(define-record vdecl       (name type))
(define-record block       (label assigns jump))
(define-record assign      (lexp exp))
(define-record nat         (datum))
(define-record varref      (var))
(define-record app         (op exps))
```

```
(define-record deref       (exp))
(define-record new         (type))
(define-record nil         (type))
(define-record arrayref    (vexp exp))
(define-record recmem      (exp var))
(define-record goto        (label))
(define-record if          (exp then-label else-label))
(define-record return      (exp))

;;=================================================================
;; type-tag
;;=================================================================

(define-record nat-type    ())
(define-record access-type (type))
(define-record array-type  (lower upper type))
(define-record rec-type    (name))
```

A.2 Parser

```
;;=================================================================
;; int-parser.scm
;; unit to parse the source code for interpreter
;;=================================================================

(load "int-syntax.scm")

;;=================================================================
;; parser
;; converts the FCL from lists to records
;;=================================================================
;; program parsing
;; prog (list) -> prog (record)
;;-----------------------------------------------------------------

(define parse-prog
  (lambda (prog)
    (let* ((parsed-decls (parse-decls (car prog)))
           (parsed-params (cadr prog))
           (parsed-blocks (map parse-block (caddr prog)))
      (begin
        (newline)
        (display "** PARSING COMPLETED **")
        (newline)
        (newline)
        (make-program parsed-decls parsed-params parsed-blocks)))))

;;-----------------------------------------------------------------
;; declarations parsing
;; decls (list) -> decls (record)
;;-----------------------------------------------------------------

(define parse-decls
  (lambda (decls)
    (let* ((tdefs (car decls))
           (vdecls (cadr decls))
           (parsed-tdefs (map parse-tdef tdefs))
           (parsed-vdecls (map parse-vdecl vdecls)))
      (make-decls parsed-tdefs parsed-vdecls))))

;;-----------------------------------------------------------------
;; type definition parsing
;; tdef (list) -> tdef (record)
;;-----------------------------------------------------------------
```

```
(define parse-tdef
  (lambda (tdef)
    (case (car tdef)
      ((record) (let* ((name (cadr tdef))
                       (vdecls (caddr tdef))
                       (parsed-vdecls (map parse-vdecl vdecls)))
                  (make-tdef name parsed-vdecls)))
      (else (error "parse : bad type definition :" tdef)))))

;;-----------------------------------------------------------------
;; variable declaration parsing
;; vdecl (list) -> vdecl (record)
;;-----------------------------------------------------------------

(define parse-vdecl
  (lambda (vdecl)
    (let* ((name (car vdecl))
           (type (caddr vdecl))
           (parsed-type (parse-type type)))
      (make-vdecl name parsed-type))))

;;-----------------------------------------------------------------
;; type parsing
;; type (list) -> type (record)
;;-----------------------------------------------------------------

(define parse-type
  (lambda (type)
    (cond
      ((pair? type)
        (case (car type)
          ((access) (make-access-type (parse-type (cadr type))))
          ((array) (make-array-type (cadr type)
                                    (caddr type)
                                    (parse-type (cadddr type))))
          ((record) (make-rec-type (cadr type)))
          (else (error "parse : bad type :" type))))
      ((symbol? type)
       (case type
         ((nat) (make-nat-type))
         (else (error "parse : bad type :" type))))
      (else (error "parse : bad type :" type)))))

;;-----------------------------------------------------------------
;; block parsing
;; block (list) -> block (record)
;;-----------------------------------------------------------------

(define parse-block
  (lambda (block)
    (make-block (car block)
                (map parse-assign (cadr block))
                (parse-jump (caddr block)))))

;;-----------------------------------------------------------------
;; assignment parsing
;; assign (list) -> assign (record)
;;-----------------------------------------------------------------

(define parse-assign
  (lambda (assign)
    (make-assign (parse-exp (car assign))
                 (parse-exp (caddr assign)))))

;;-----------------------------------------------------------------
;; expression parsing
;; exp (list) -> exp (record)
```

```
;;-------------------------------------------------------------

(define parse-exp
  (lambda (exp)
    (if (pair? exp)
        (case (car exp)
          ((quote) (make-nat (cadr exp)))
          ((+ - * / = > gte < lte neq)
           (make-app (car exp) (map parse-exp (cdr exp))))
          ((deref) (make-deref (parse-exp (cadr exp))))
          ((new) (make-new (parse-type (cadr exp))))
          ((nil) (make-nil (parse-type (cadr exp))))
          ((recmem) (make-recmem (parse-exp (cadr exp))
                                 (caddr exp)))
          (([]) (make-arrayref (parse-exp (cadr exp))
                               (parse-exp (caddr exp))))
          (else (error "parse : not an expression :" exp)))
        (if (number? exp)
            (make-nat exp)
            (make-varref exp)))))

;;-------------------------------------------------------------
;; jump parsing
;; jump (list) -> jump (record)
;;-------------------------------------------------------------

(define parse-jump
  (lambda (jump)
    (case (car jump)
      ((goto) (make-goto (cadr jump)))
      ((if) (make-if (parse-exp (cadr jump)) (caddr jump) (cadddr jump)))
      ((return) (make-return (parse-exp (cadr jump))))
      (else (error "parse : not a jump :" jump)))))

;;-------------------------------------------------------------
;; parse-prog alias (main)
;;-------------------------------------------------------------

(define parse parse-prog)
```

A.3 Type checker

```
;;=============================================================
;; int-check.scm
;; unit to type-check for interpereter
;;=============================================================

(load "int-parser.scm")

(load "int-table.scm")

;;=============================================================
;; program type-checking
;;=============================================================

(define check-prog
  (lambda (parsed-prog)
    (variant-case parsed-prog
      (program (decls params blocks)
        (let* ((tmp (reset-type-head!))
               (tmp (reset-type-table!))
               (tmp (reset-type-env!))
               (tmp (check-decls decls))
               (tmp (check-blocks blocks))
               (tmp (newline))
```

```
            (tmp (display "** TYPE CHECK COMPLETED **"))
            (tmp (newline))
            (tmp (newline)))
          '()))
      (else (error "check : not a program :" parsed-prog)))))

;;-------------------------------------------------------------------
;; declarations type-checking
;; - builds global type-head, type-table and type-env (side-effect)
;;-------------------------------------------------------------------

(define check-decls
  (lambda (decls)
    (let ((tdefs (decls->tdefs decls))
          (vdecls (decls->vdecls decls)))
      (begin
        (set-type-head!  (check-head tdefs empty-table))
        (set-type-table! (check-tdefs tdefs empty-table))
        (set-type-env!   (check-vdecls vdecls empty-table))))))

;;-------------------------------------------------------------------
;; type definitions header scanning
;; tdefs x table -> table
;; - build list recursively for type-head
;;-------------------------------------------------------------------

(define check-head
  (lambda (tdefs table)
    (if (null? tdefs)
        table
        (let* ((tdef (car tdefs))
               (new-table (cons (tdef->name tdef) table)))
          (check-head (cdr tdefs) new-table)))))

;;-------------------------------------------------------------------
;; type definitions type-checking
;; tdefs x table -> table
;; - build table recursively for type-table
;;-------------------------------------------------------------------

(define check-tdefs
  (lambda (tdefs table)
    (if (null? tdefs)
        table
        (let* ((tdef (car tdefs))
               (name (tdef->name tdef)))
          (if (exist-table? name table)
              (error "check :" name 'exist 'in 'type 'definition)
              (let* ((vdecls (tdef->vdecls tdef))
                     (tdef-table (check-vdecls vdecls empty-table))
                     (new-table (update-table name tdef-table table)))
                (check-tdefs (cdr tdefs) new-table)))))))

;;-------------------------------------------------------------------
;; variable declarations type-checking
;; vdecls x table -> table
;; - build table recursively for type-env
;;-------------------------------------------------------------------

(define check-vdecls
  (lambda (vdecls table)
    (if (null? vdecls)
        table
        (let* ((vdecl (car vdecls))
               (name (vdecl->name vdecl)))
          (if (exist-table? name table)
              (error "check :" name 'exist 'in 'variable 'declaration)
```

```
            (let* ((type-checked (check-type (vdecl->type vdecl)))
                   (new-table (update-table name type-checked table)))
              (check-vdecls (cdr vdecls) new-table)))))))

;;-----------------------------------------------------------------
;; type type-checking
;; type -> type
;;-----------------------------------------------------------------

(define check-type
  (lambda (type)
    (variant-case type
      (nat-type () type)
      (access-type (type) (make-access-type (check-type type)))
      (array-type (lower upper type)
        (make-array-type lower upper (check-type type)))
      (rec-type (name)
        (if (exist-type-head? name)
            type
            (error "check : undefined type :" type)))
      (else (error "check : unknown type :" type)))))

;;-----------------------------------------------------------------
;; blocks type-checking
;;-----------------------------------------------------------------

(define check-blocks
  (lambda (blocks)
    (map check-block blocks)))

;;-----------------------------------------------------------------
;; (each) block type-checking
;;-----------------------------------------------------------------

(define check-block
  (lambda (block)
    (let ((assigns (block->assigns block))
          (jump    (block->jump block)))
      (begin
        (check-assigns assigns)
        (check-jump jump)))))

;;-----------------------------------------------------------------
;; assignments type-checking
;;-----------------------------------------------------------------

(define check-assigns
  (lambda (assigns)
    (map check-assign assigns)))

;;-----------------------------------------------------------------
;; (each) assignment type-checking
;;-----------------------------------------------------------------

(define check-assign
  (lambda (assign)
    (variant-case assign
      (assign (lexp exp)
        (let* ((lexp-type (check-exp lexp))
               (exp-type  (check-exp exp)))
          (check-assign-equiv lexp-type exp-type)))
      (else (error "check : not an assignment :" assign)))))

;;-----------------------------------------------------------------
;; assignment equivalent type-checking
;; type x type -> #t or error otherwise
;;-----------------------------------------------------------------
```

```
(define check-assign-equiv
  (lambda (ltype type)
    (if (not (equal-type? ltype type))
        (error "check : non equivalent type in 'assign' :" ltype type))))

;;-------------------------------------------------------------------
;; expression type-checking
;; exp -> type
;;-------------------------------------------------------------------

(define check-exp
  (lambda (exp)
    (variant-case exp
      (nat () (make-nat-type))
      (varref (var) (lookup-type-env var))
      (app (op exps)
        (let* ((exp1  (car exps))
               (type1 (check-exp exp1))
               (exp2  (cadr exps))
               (type2 (check-exp exp2)))
          (check-op op type1 type2)))
      (deref (exp)
        (let ((exp-type (check-exp exp)))
          (variant-case exp-type
            (access-type (type) type)
            (else (error "check : non pointer type in 'deref' :" exp)))))
      (new (type) (make-access-type (check-type type)))
      (nil (type) (make-access-type (check-type type)))
      (arrayref (vexp exp)
        (let* ((vexp-type (check-exp vexp))
               (exp-type  (check-exp exp)))
          (if (array-type? vexp-type)
              (if (nat-type? exp-type)
                  (array-type->type vexp-type)
                  (error "check : non nat type in 'arrayref' :" exp))
              (error "check : non array type in 'arrayref' :" vexp))))
      (recmem (exp var)
        (let* ((exp-type (check-exp exp)))
          (if (rec-type? exp-type)
              (let ((rec-env (lookup-type-table (rec-type->name exp-type))))
                (lookup-table var rec-env))
              (error "check : non record type in 'recmem' :" exp))))
      (else (error "check : not an expression :" exp)))))

;;-------------------------------------------------------------------
;; operation type-checking
;; op x type x type -> type
;;-------------------------------------------------------------------

(define check-op
  (lambda (op type1 type2)
    (case op
      ((+ - * / > gte < lte)
       (if (and (nat-type? type1)
                (nat-type? type2))
           (make-nat-type)
           (error "check : nat-type expected in 'op' :" op)))
      ((= neq)
       (if (equal-type? type1 type2)
           (make-nat-type)
           (else (error "check  : non equivalent type in 'op' :" op))))
      (else (error "check : not an operator :" op)))))

;-------------------------------------------------------------------
; check for equal type
; type x type -> #t | #f
```

```
;-----------------------------------------------------------------

(define equal-type?
  (lambda (type1 type2)
    (cond
      ((and (nat-type? type1) (nat-type? type2)) #t)
      ((and (access-type? type1) (access-type? type2))
       (equal-type? (access-type->type type1)
                    (access-type->type type2)))
      ((and (array-type? type1) (array-type? type2))
       (and (equal? (array-type->lower type1) (array-type->lower type2))
            (equal? (array-type->upper type1) (array-type->upper type2))
            (equal-type? (array-type->type type1)
                         (array-type->type type2))))
      ((and (rec-type? type1) (rec-type? type2))
       (equal? (rec-type->name type1) (rec-type->name type2)))
      (else #f))))

;;---------------------------------------------------------------
;; jump type-checking
;;---------------------------------------------------------------

(define check-jump
  (lambda (jump)
    (variant-case jump
      (goto (label) ())
      (if (exp then-label else-label)
        ;; assumption : boolean type equivalent with nat-type

        (if (not (nat-type? (check-exp exp)))
            (error "check : non nat type in 'if' :" exp)
            ()))
      (return (exp)
        ;; assumption : exp must be nat-type

        (if (not (nat-type? (check-exp exp)))
            (error "check : non nat type in 'return' :" exp)))
      (else (error "check : not a jump :" jump)))))

;;---------------------------------------------------------------
;; check-prog alias (main)
;;---------------------------------------------------------------

(define type-check check-prog)
```

A.4 Evaluator

```
;;===============================================================
;; int-eval.scm
;; unit to evaluates the program for interpreter
;;===============================================================

(load "int-check.scm")

;;===============================================================
;; halt label
;;===============================================================

(define-record halt      (val))

;;===============================================================
;; values
;;===============================================================

(define-record nat-val   (val))
```

```
(define-record loc-val   (loc))
(define-record array-val (map lower upper))
(define-record rec-val   (env))

;;-----------------------------------------------------------------
;; initial value for nat-val
;; -> val
;;-----------------------------------------------------------------

(define init-nat
  (lambda ()
    (make-nat-val 0)))

;;-----------------------------------------------------------------
;; initial value for loc-val
;; -> val
;;-----------------------------------------------------------------

(define init-loc
  (lambda ()
    (make-loc-val (make-nil-val))))

;;-----------------------------------------------------------------
;; nil value for loc
;;-----------------------------------------------------------------

(define-record nil-val     ())

;;=================================================================
;; program evaluation
;; prog x args -> val
;;=================================================================

(define eval-prog
  (lambda (parsed-prog args)
    (let* ((tmp        (reset-env!))
           (tmp        (reset-store!))
           (tmp        (next-loc! 'reset))
           (params     (program->params parsed-prog))
           (blocks     (program->blocks parsed-prog))
           (init-label (block->label (car blocks)))
           (init-env   (eval-vdecls type-env))
           (tmp        (set-env! init-env))
           (tmp        (eval-params params args))
           (return (eval-blocks init-label blocks)))
      return)))

;;-----------------------------------------------------------------
;; variable declarations evaluation
;; type-env -> env
;; - create an environment and update store using declaration info :
;;   type-env
;;-----------------------------------------------------------------

(define eval-vdecls
  (lambda (type-env)
    ;; creates a value environment (binding names to locations in
    ;; store)
    (letrec ((loop (lambda (remaining-type-env var-env)
                     (if (null? remaining-type-env)
                         var-env
                         (let* ((vdecl   (car remaining-type-env))
                                (var     (car vdecl))
                                (type    (cdr vdecl))
                                (new-loc (next-loc! 'next))
                                (new-val (eval-init-val type))
                                (new-var-env
```

```
                              (update-table var new-loc var-env)))
                          (begin
                            (update-store! new-loc new-val)
                            (loop (cdr remaining-type-env) new-var-env)))))))
      (loop type-env type-env))))

;;-----------------------------------------------------------------
;; initial value evaluation
;; - create an initial store value for a given type using declaration
;;   info : type table
;;-----------------------------------------------------------------

(define eval-init-val
  (lambda (type)
    (variant-case type
      (nat-type () (init-nat))
      (access-type (type) (init-loc))
      (array-type (lower upper type)
        (letrec ((convert (lambda (index table)
                            (if (> index upper)
                                table
                                (let* ((new-index (+ index 1))
                                       (new-table (update-table index
                                                                type
                                                                table)))
                                  (convert new-index new-table))))))
          (let* ((adecls    (convert lower empty-table))
                 (array-map (eval-vdecls adecls)))
            (make-array-val array-map lower upper))))
      (rec-type (name)
        (let* ((vdecls  (lookup-type-table name))
               (rec-env (eval-vdecls vdecls)))
          (make-rec-val rec-env)))
      (else (error "eval : unknown type :" type)))))

;;-----------------------------------------------------------------
;; parameters evaluation
;; - tag each argument as nat, then loop through parameters
;;   and update (side-effect) the store so that each parameter
;;   is bound to the corresponding argument value.
;;-----------------------------------------------------------------

(define eval-params
  (lambda (params args)
    (letrec ((loop (lambda (params tagged-args)
                     (if (not (null? params))
                         (let ((param      (car params))
                               (tagged-arg (car tagged-args)))
                           (begin
                             (update-store! (lookup-env param) tagged-arg)
                             (loop (cdr params) (cdr tagged-args))))))))
      (loop params (map make-nat-val args)))))

;;-----------------------------------------------------------------
;; blocks evaluation
;; label x blocks -> val
;;-----------------------------------------------------------------

(define eval-blocks
  (lambda (label blocks)
    ;; given label, get-block will return with the labelled block or error
    (letrec ((get-block (lambda (label tmp-blocks)
                          (if (null? tmp-blocks)
                              (error "eval : unknown label :" label)
                              (if (equal? label
                                          (block->label (car tmp-blocks)))
                                  (car tmp-blocks)
```

```
                                (get-block label (cdr tmp-blocks)))))))
      (let* ((block (get-block label blocks))
             (label (eval-block block blocks)))
        (if (halt? label)
            (halt->val label)
            (eval-blocks label blocks))))))

;;-----------------------------------------------------------------
;; (each) block evaluation
;; block x blocks -> val
;;-----------------------------------------------------------------

(define eval-block
  (lambda (block blocks)
    (let ((tmp (eval-assigns (block->assigns block))))
      (eval-jump (block->jump block) blocks))))

;;-----------------------------------------------------------------
;; assignments evaluation
;;-----------------------------------------------------------------

(define eval-assigns
  (lambda (assigns)
    (if (not (null? assigns))
        (let ((tmp (eval-assign (car assigns))))
          (eval-assigns (cdr assigns))))))

;;-----------------------------------------------------------------
;; (each) assignment evaluation
;;-----------------------------------------------------------------

(define eval-assign
  (lambda (assign)
    (variant-case assign
      (assign (lexp exp)
              (let* ((val  (eval-rexp exp))
                     (lval (eval-lexp lexp)))
                (eval-update lval val))))))

;;-----------------------------------------------------------------
;; assignment update evaluation
;;-----------------------------------------------------------------

(define eval-update
  (lambda (lval val)
    (letrec ((loop (lambda (lenv env)
                     (if (not (null? env))
                         (begin
                           (eval-update (cdar lenv)
                                        (lookup-store (cdar env)))
                           (loop (cdr lenv) (cdr env)))))))
      (variant-case val
        (nat-val ()
          (update-store! lval val))
        (loc-val ()
          (update-store! lval val))
        (array-val (map lower upper)
          (let* ((array-lval (lookup-store lval))
                 (lmap       (array-val->map array-lval)))
            (loop lmap map)))
        (rec-val (env)
          (let* ((rec-lval (lookup-store lval))
                 (lenv     (rec-val->env rec-lval)))
            (loop lenv env)))
        (else (error "eval : unknown val in 'assign update' :" val))))))

;;-----------------------------------------------------------------
```

```
;; left expression evaluation
;; exp -> loc
;;-------------------------------------------------------------------

(define eval-lexp
  (lambda (lexp)
    (variant-case lexp
      (varref (var) (lookup-env var))
      (deref (exp)
        (let* ((val (eval-rexp exp))
               (loc (loc-val->loc val)))
          (if (nil-val? loc)
              (error "eval : deref of nil val in 'lexp' :" exp)
              loc)))
      (arrayref (vexp exp)
        (let* ((val     (eval-rexp vexp))
               (ind-val (eval-rexp exp))
               (ind     (nat-val->val ind-val)))
          (if (and (>= ind (array-val->lower val))
                   (<= ind (array-val->upper val)))
              (lookup-table ind (array-val->map val))
              (error "eval : array index out of range :" ind))))
      (recmem (exp var)
        (let* ((val     (eval-rexp exp))
               (rec-env (rec-val->env val)))
          (lookup-table var rec-env)))
      (else (error "eval : unknown lexp :" lexp)))))

;;-------------------------------------------------------------------
;; right expression evaluation
;; exp -> val
;;-------------------------------------------------------------------

(define eval-rexp
  (lambda (rexp)
    (variant-case rexp
      (nat (datum) (make-nat-val datum))
      (varref (var) (lookup-store (lookup-env var)))
      (app (op exps)
        (let ((val1 (eval-rexp (car exps)))
              (val2 (eval-rexp (cadr exps))))
          (eval-op op val1 val2)))
      (deref (exp)
        (let* ((val (eval-rexp exp))
               (loc (loc-val->loc val)))
          (if (nil-val? loc)
              (error "eval : deref of nil val in 'rexp' :" exp)
              (lookup-store loc))))
      (new (type)
        (let* ((new-loc (next-loc! 'next))
               (new-val (eval-init-val type))
               (tmp     (update-store! new-loc new-val)))
          (make-loc-val new-loc)))
      (nil (type) (init-loc))
      (arrayref (vexp exp)
        (let* ((val     (eval-rexp vexp))
               (ind-val (eval-rexp exp))
               (ind     (nat-val->val ind-val)))
          (if (and (>= ind (array-val->lower val))
                   (<= ind (array-val->upper val)))
              (let ((amap (array-val->map val)))
                (lookup-store (lookup-table ind amap)))
              (error "eval : array index out of range :" ind))))
      (recmem (exp var)
        (let* ((val     (eval-rexp exp))
               (rec-env (rec-val->env val)))
          (lookup-store (lookup-table var rec-env)))))
```

```
      (else (error "eval : unknown rexp :" rexp)))))

;;-----------------------------------------------------------------
;; operation evaluation
;; op x val x val -> val
;;-----------------------------------------------------------------

(define eval-op
  (lambda (op val1 val2)
    (make-nat-val
      (case op
        ((+) (+ (nat-val->val val1) (nat-val->val val2)))
        ((-) (- (nat-val->val val1) (nat-val->val val2)))
        ((*) (* (nat-val->val val1) (nat-val->val val2)))
        ((/) (/ (nat-val->val val1) (nat-val->val val2)))
        ((>) (if (> (nat-val->val val1) (nat-val->val val2)) 1 0))
        ((gte) (if (>= (nat-val->val val1) (nat-val->val val2)) 1 0))
        ((<) (if (< (nat-val->val val1) (nat-val->val val2)) 1 0))
        ((lte) (if (<= (nat-val->val val1) (nat-val->val val2)) 1 0))
        ((=)
         (letrec ((loop (lambda (op env1 env2)
                          (if (null? env1)
                              1
                              (let ((ret (eval-op op
                                                  (lookup-store (cdar env1))
                                                  (lookup-store (cdar env2)))))
                                (if (true-val? ret)
                                    (loop op (cdr env1) (cdr env2))
                                    0))))))
           (cond
             ((and (nat-val? val1) (nat-val? val2))
              (if (equal? (nat-val->val val1) (nat-val->val val2)) 1 0))
             ((and (loc-val? val1) (loc-val? val2))
              (if (equal? (loc-val->loc val1) (loc-val->loc val2)) 1 0))
             ((and (array-val? val1) (array-val? val2))
              (loop op (array-val->map val1) (array-val->map val2)))
             ((and (rec-val? val1) (rec-val? val2))
              (loop op (rec-val->env val1) (rec-val->env val2))))))
        (else (error "eval : unknown op :" op))))))

;;-----------------------------------------------------------------
;; jump evaluation
;; jump x blocks -> label
;;-----------------------------------------------------------------

(define eval-jump
  (lambda (jump blocks)
    (variant-case jump
      (goto (label) label)
      (return (exp)
        (let ((val (eval-rexp exp)))
          (make-halt (nat-val->val val))))
      (if (exp then-label else-label)
          (if (true-val? (eval-rexp exp))
              then-label
              else-label)))))

;;-----------------------------------------------------------------
;; check if expression is true
;; val -> #t | #f
;;-----------------------------------------------------------------

(define true-val?
  (lambda (val)
    (not (equal? 0 (nat-val->val val)))))

;;-----------------------------------------------------------------
```

```
;; run the program (main)
;;-----------------------------------------------------------------

(define run
  (lambda (prog args)
    (let ((parsed-prog (parse prog)))
      (begin
        (type-check parsed-prog)
        (eval-prog parsed-prog args)))))
```

A.5 Module for tables manipulation

```
;;=================================================================
;; int-table.scm
;; functions for manipulating tables and lists
;; with side-effect for interpreter
;;=================================================================

(load "table.scm")

;;=================================================================
;; global list type-head
;;=================================================================
;; create new empty type-head
;;-----------------------------------------------------------------

(define type-head empty-list)

;;-----------------------------------------------------------------
;; reset type-head
;;-----------------------------------------------------------------

(define reset-type-head!
  (lambda ()
    (set! type-head empty-list)))

;;-----------------------------------------------------------------
;; set type-head
;;-----------------------------------------------------------------

(define set-type-head!
  (lambda (new-list)
    (set! type-head new-list)))

;;-----------------------------------------------------------------
;; lookup var existence in global type-head
;; var -> #t | #f
;;-----------------------------------------------------------------

(define exist-type-head?
  (lambda (var)
    (exist-list? var type-head)))

;;=================================================================
;; global table type-table
;;=================================================================
;; create new empty global type-table
;;-----------------------------------------------------------------

(define type-table empty-table)

;;-----------------------------------------------------------------
;; reset type-table
;;-----------------------------------------------------------------
```

```
(define reset-type-table!
  (lambda ()
    (set! type-table empty-table)))

;;-----------------------------------------------------------------
;; set type-table
;;-----------------------------------------------------------------

(define set-type-table!
  (lambda (new-table)
    (set! type-table new-table)))

;;-----------------------------------------------------------------
;; lookup decls for var in global type-table
;; var -> decls
;;-----------------------------------------------------------------

(define lookup-type-table
  (lambda (var)
    (lookup-table var type-table)))

;;-----------------------------------------------------------------
;; lookup var existence in global type-table
;; var -> #t | #f
;;-----------------------------------------------------------------
(define exist-type-table?
  (lambda (var)
    (exist-table? var type-table)))

;;-----------------------------------------------------------------
;; update var with new decls in global type-table
;;-----------------------------------------------------------------

(define update-type-table!
  (lambda (var decls)
    (set! type-table (update-table var decls type-table))))

;;=================================================================
;; global table type-env
;;=================================================================
;; create new empty global type-env
;;-----------------------------------------------------------------

(define type-env empty-table)

;;-----------------------------------------------------------------
;; reset type-env
;;-----------------------------------------------------------------

(define reset-type-env!
  (lambda ()
    (set! type-env empty-table)))

;;-----------------------------------------------------------------
;; set type-env
;;-----------------------------------------------------------------

(define set-type-env!
  (lambda (new-table)
    (set! type-env new-table)))

;;-----------------------------------------------------------------
;; lookup type for var in global type-env
;; var -> type
;;-----------------------------------------------------------------

(define lookup-type-env
```

```
  (lambda (var)
    (lookup-table var type-env)))

;;-----------------------------------------------------------------
;; update var with new type in global type-env
;;-----------------------------------------------------------------

(define update-type-env!
  (lambda (var type)
    (set! type-env (update-table var type type-env))))

;;=================================================================
;; global table environment
;;=================================================================
;; create new empty global environment
;;-----------------------------------------------------------------

(define env empty-table)

;;-----------------------------------------------------------------
;; reset env
;;-----------------------------------------------------------------

(define reset-env!
  (lambda ()
    (set! env empty-table)))

;;-----------------------------------------------------------------
;; set env
;;-----------------------------------------------------------------

(define set-env!
  (lambda (new-table)
    (set! env new-table)))

;;-----------------------------------------------------------------
;; location function
;; - next     : get current location and update location
;; - reset    : reset location to initial value (0)
;; - (number) : set location to the number
;;-----------------------------------------------------------------

(define next-loc!
  (let ((loc 0))
    (lambda (comm)
      (cond
        ((symbol? comm)
         (case comm
           ((next)
            (let ((loc-current loc)
                  (tmp (set! loc (+ loc 1))))
              (string->symbol (string-append "loc-"
                                             (number->string loc-current)))))
           ((reset)
            (let ((loc-current loc)
                  (tmp (set! loc 0)))
              loc-current))))
        ((number? comm)
         (set! loc comm))))))

;;-----------------------------------------------------------------
;; lookup location for var in global environment
;; var -> loc
;;-----------------------------------------------------------------

(define lookup-env
  (lambda (var)
```

```
    (lookup-table var env)))

;;-----------------------------------------------------------------
;; update var with new location in global environment
;;-----------------------------------------------------------------

(define update-env!
  (lambda (var loc)
    (set! env (update-table var loc env))))

;;=================================================================
;; global table store
;;=================================================================
;; create new empty global store
;;-----------------------------------------------------------------

(define store empty-table)

;;-----------------------------------------------------------------
;; reset store
;;-----------------------------------------------------------------

(define reset-store!
  (lambda ()
    (set! store empty-table)))

;;-----------------------------------------------------------------
;; set store
;;-----------------------------------------------------------------

(define set-store!
  (lambda (new-table)
    (set! store new-table)))

;;-----------------------------------------------------------------
;; lookup value for var in global store
;; loc -> value
;;-----------------------------------------------------------------

(define lookup-store
  (lambda (loc)
    (lookup-table loc store)))

;;-----------------------------------------------------------------
;; update loc with new value in global store
;;-----------------------------------------------------------------

(define update-store!
  (lambda (loc val)
    (set! store (update-table loc val store))))

;;=================================================================
;; table.scm
;; functions for manipulating tables and lists
;;=================================================================

(define empty-list '())

(define empty-table '())

;;=================================================================
;; list manipulation with no side-effect
;;=================================================================
;; check if variable exist in list
;; var x list -> #t | #f
;;-----------------------------------------------------------------
```

```
(define exist-list?
  (lambda (var list)
    (if (null? list)
#f
(if (equal? (car list) var)
    #t
    (exist-list? var (cdr list))))))

;;=================================================================
;; table manipulation with no side-effect
;;=================================================================
;; update table variable with new value if variable exist or
;; new cell for variable otherwise
;; var x val x table -> table
;;-----------------------------------------------------------------

(define update-table
  (lambda (var val table)
    (if (null? table)
(list (cons var val))
(if (eq? (caar table) var)
    (cons (cons var val) (cdr table))
    (cons (car table) (update-table var val (cdr table)))))))

;;-----------------------------------------------------------------
;; lookup variable's value in table
;; var x table -> value (if exist or error otherwise)
;;-----------------------------------------------------------------

(define lookup-table
  (lambda (var table)
    (if (null? table)
(error "lookup-table : variable not found :" var)
(if (equal? (caar table) var)
    (cdar table)
    (lookup-table var (cdr table))))))

;;-----------------------------------------------------------------
;; check if variable exist in table
;; var x table -> #t | #f
;;-----------------------------------------------------------------

(define exist-table?
  (lambda (var table)
    (if (null? table)
#f
(if (equal? (caar table) var)
    #t
    (exist-table? var (cdr table))))))

;;-----------------------------------------------------------------
;; add table with list of variable and value
;; vars x vals x table -> table
;;-----------------------------------------------------------------

(define add-table
  (lambda (vars vals table)
    (if (null? vars)
table
(cons (cons (car vars) (car vals))
      (add-table (cdr vars) (cdr vals) table)))))

;;-----------------------------------------------------------------
;; domain table
;; table -> list
;; (domain x image) -> domain
;;-----------------------------------------------------------------
```

```
(define domain-table
  (lambda (table)
    (map car table)))

;;-----------------------------------------------------------------
;; image table
;; table -> list
;; (domain x image) -> image
;;-----------------------------------------------------------------

(define image-table
  (lambda (table)
    (map cdr table)))
```

APPENDIX B

SPECIALIZER

B.1 Module for syntax

```
;;=================================================================
;; bt-syntax.scm
;; syntax definition for specializer
;;=================================================================

(require "struct)

;;=================================================================
;; concrete syntax
;;=================================================================

;; <program>      ::= (((<type-def>*)(<decl>*))(<var>*)(<block>+))
;; <type-def>     ::= (record <var> (<decl>+) <bt-val> <bt-mod>)
;; <decl>         ::= (<var> : <btype>)
;; <btype>        ::= (nat <bt-val> <bt-mod>)
;;                  | (access <btype> <bt-val> <bt-mod>)
;;                  | (array <nat> <nat> <btype> <bt-val> <bt-mod>)
;;                  | (record <var>)
;; <bt-val>       ::= S | D
;; <bt-mod>       ::= E | R
;; <block>        ::= (<label> (<assignment>*) <jump>)
;; <assignment>   ::= (<lexp> := <exp>)
;; <lexp>         ::= <var>
;;                  | (deref <exp>)
;;                  | ([] <exp> <exp>)
;;                  | (recmem <exp> <var>)
;; <exp>          ::= <nat>
;;                  | <var>
;;                  | (<op> <exps>)
;;                  | (deref <exp>)
;;                  | (new <btype> <bt-val> <bt-mod>)
;;                  | (nil <btype> <bt-val> <bt-mod>)
;;                  | ([] <exp> <exp>)
;;                  | (recmem <exp> <var>)
;; <jump>         ::= (goto <label>)
;;                  | (if <exp> <label> <label>)
;;                  | (return <exp>)
;; <op>           ::= + | - | * | / | = | > | gte | < | lte | neq
;; <nat>          ::= 0 | 1 | 2 | ...
;; <label>        ::= any symbol
;; <var>          ::= any alphanumeric symbol not used as a command,
;;                    number, and later lists and symbols

;;=================================================================
;; abstract syntax
;;=================================================================
;; source code abstract syntax
;;-----------------------------------------------------------------

(define-record program      (decls params blocks))
(define-record decls        (tdefs vdecls))
(define-record tdef         (name vdecls))
(define-record vdecl        (name type))
(define-record block        (label assigns jump))
```

```
(define-record assign       (lexp exp))
(define-record nat          (datum))
(define-record varref       (var))
(define-record app          (op exps))
(define-record deref        (exp))
(define-record new          (type))
(define-record nil          (type))
(define-record arrayref     (vexp exp))
(define-record recmem       (exp var))
(define-record goto         (label))
(define-record if           (exp then-label else-label))
(define-record return       (exp))

;;-------------------------------------------------------------------
;; annotated abstract syntax
;;-------------------------------------------------------------------

(define-record bt-tdef      (btval btmod name vdecls))
(define-record bt-vdecl     (btval btmod name btype))
(define-record bt-assign    (btval btmod lexp exp))
(define-record bt-nat       (btval btmod datum))
(define-record bt-app       (btval btmod op exps))
(define-record bt-deref     (btval btmod exp))
(define-record bt-new       (btval btmod btype))
(define-record bt-nil       (btval btmod btype))
(define-record bt-arrayref  (btval btmod vexp exp))
(define-record bt-recmem    (btval btmod exp var))
(define-record bt-goto      (btval btmod label))
(define-record bt-if        (btval btmod exp then-label else-label))
(define-record bt-return    (btval btmod exp))

(define-record get-dyn      (exp))
(define-record get-res      (exp))
(define-record lift         (exp))
(define-record get-val      (exp))
(define-record get-exp      (exp))

;;===================================================================
;; type-tag
;;===================================================================
;; source code type-tag
;;-------------------------------------------------------------------

(define-record nat-type     ())
(define-record access-type  (type))
(define-record array-type   (lower upper type))
(define-record rec-type     (name))

;;-------------------------------------------------------------------
;; annotated type-tag
;;-------------------------------------------------------------------

(define-record nat-btype    (btval btmod))
(define-record access-btype (btval btmod btype))
(define-record array-btype  (btval btmod lower upper btype))
(define-record rec-btype    (btval btmod name))

;;===================================================================
;; bt definition
;;===================================================================

(define static     's)
(define dynamic    'd)
(define eliminable 'e)
(define residual   'r)
```

B.2 Parser

```
;;=====================================================================
;; bt-parser.scm
;; unit to parse the source code for specializer
;; put initial constructs annotation
;;=====================================================================

(load "bt-syntax.scm")

;;=====================================================================
;; bt-val operation
;;=====================================================================
;; operation for static
;;---------------------------------------------------------------------

(define static?
  (lambda (btval)
    (if (equal? btval static)
        #t
        #f)))

;;---------------------------------------------------------------------
;; operation for dynamic
;;---------------------------------------------------------------------

(define dynamic?
  (lambda (btval)
    (if (equal? btval dynamic)
        #t
        #f)))

;;=====================================================================
;; bt-mod operation
;;=====================================================================
;; operation for eliminable
;;---------------------------------------------------------------------

(define eliminable?
  (lambda (btmod)
    (if (equal? btmod eliminable)
        #t
        #f)))

;;---------------------------------------------------------------------
;; operation for residual
;;---------------------------------------------------------------------

(define residual?
  (lambda (bt-mod)
    (if (equal? bt-mod residual)
        #t
        #f)))

;;=====================================================================
;; bt-parser
;; converts the FCL-ann from lists to records with annotation
;;=====================================================================
;; program parsing
;; prog (list) -> prog (record)
;;---------------------------------------------------------------------

(define bt-parse-prog
  (lambda (prog)
    (let* ((parsed-decls (bt-parse-decls (car prog)))
           (parsed-params (cadr prog))
           (parsed-blocks (map bt-parse-block (caddr prog))))
```

```
      (begin
        (newline)
        (display "** PARSING COMPLETED **")
        (newline)
        (newline)
        (make-program parsed-decls parsed-params parsed-blocks)))))

;;-------------------------------------------------------------------
;; declarations parsing
;; decls (list) -> decls (record)
;;-------------------------------------------------------------------

(define bt-parse-decls
  (lambda (decls)
    (let* ((tdefs (car decls))
           (vdecls (cadr decls))
           (parsed-tdefs (map bt-parse-tdef tdefs))
           (parsed-vdecls (map bt-parse-vdecl vdecls)))
      (make-decls parsed-tdefs parsed-vdecls))))

;;-------------------------------------------------------------------
;; type definition parsing
;; tdef (list) -> tdef (record)
;;-------------------------------------------------------------------

(define bt-parse-tdef
  (lambda (tdef)
    (case (car tdef)
      ((record) (let* ((name  (cadr tdef))
                       (vdecls (caddr tdef))
                       (btval   (cadddr tdef))
                       (btmod   (car (cddddr tdef)))
                       (parsed-vdecls (map bt-parse-vdecl vdecls))
                       (parsed-btval  (bt-parse-val btval))
                       (parsed-btmod  (bt-parse-mod btmod)))
                  (make-bt-tdef parsed-btval
                                parsed-btmod
                                name
                                parsed-vdecls)))
      (else (error "bt-parse : bad type definition :" tdef)))))

;;-------------------------------------------------------------------
;; variable declaration parsing
;; vdecl (list) -> vdecl (record)
;;-------------------------------------------------------------------

(define bt-parse-vdecl
  (lambda (vdecl)
    (let* ((name (car vdecl))
           (btype (caddr vdecl))
           (parsed-btype (bt-parse-type btype)))
      (make-bt-vdecl static eliminable name parsed-btype))))

;;-------------------------------------------------------------------
;; btype parsing
;; btype (list) -> btype (record)
;;-------------------------------------------------------------------

(define bt-parse-type
  (lambda (btype)
    (cond
      ((pair? btype)
       (case (car btype)
         ((nat) (make-nat-btype (bt-parse-val (cadr btype))
                                (bt-parse-mod (caddr btype))))
         ((access) (make-access-btype (bt-parse-val (caddr btype))
                                      (bt-parse-mod (cadddr btype))
```

```
                                    (bt-parse-type (cadr btype))))
        ((array) (make-array-btype (bt-parse-val (car (cddddr btype)))
                                   (bt-parse-mod (cadr (cddddr btype)))
                                   (cadr btype)
                                   (caddr btype)
                                   (bt-parse-type (cadddr btype))))
        ((record) (make-rec-btype static eliminable (cadr btype)))
        (else (error "bt-parse : bad type :" btype))))
      (else (error "bt-parse : bad type :" btype)))))

;;-----------------------------------------------------------------
;; bt-val parsing
;; bt-val (list) -> bt-val (record)
;;-----------------------------------------------------------------

(define bt-parse-val
  (lambda (btval)
    (cond
      ((static? btval) static)
      ((dynamic? btval) dynamic)
      (else (error "bt-parse : unknown bt-val :" btval)))))

;;-----------------------------------------------------------------
;; bt-mod parsing
;; bt-mod (list) -> bt-mod (record)
;;-----------------------------------------------------------------

(define bt-parse-mod
  (lambda (btmod)
    (cond
      ((eliminable? btmod) eliminable)
      ((residual? btmod) residual)
      (else (error "bt-parse : unknown bt-mod :" btmod)))))

;;-----------------------------------------------------------------
;; block parsing
;; block (list) -> block (record)
;;-----------------------------------------------------------------

(define bt-parse-block
  (lambda (block)
    (make-block (car block)
                (map bt-parse-assign (cadr block))
                (bt-parse-jump (caddr block)))))

;;-----------------------------------------------------------------
;; assignment parsing
;; assign (list) -> assign (record)
;;-----------------------------------------------------------------

(define bt-parse-assign
  (lambda (assign)
    (make-bt-assign static
                    eliminable
                    (bt-parse-exp (car assign))
                    (bt-parse-exp (caddr assign)))))

;;-----------------------------------------------------------------
;; expression parsing
;; exp (list) -> exp (record)
;;-----------------------------------------------------------------

(define bt-parse-exp
  (lambda (exp)
    (if (pair? exp)
        (case (car exp)
          ((quote)
```

```
          (make-bt-nat static eliminable (cadr exp)))
         ((+ - * / = > gte < lte neq)
          (make-bt-app static
                       eliminable
                       (car exp)
                       (map bt-parse-exp (cdr exp))))
         ((deref)
          (make-bt-deref static eliminable (bt-parse-exp (cadr exp))))
         ((new)
          (make-bt-new (bt-parse-val (caddr exp))
                       (bt-parse-mod (cadddr exp))
                       (bt-parse-type (cadr exp))))
         ((nil)
          (make-bt-nil (bt-parse-val (caddr exp))
                       (bt-parse-mod (cadddr exp))
                       (bt-parse-type (cadr exp))))
         (([])
          (make-bt-arrayref static
                            eliminable
                            (bt-parse-exp (cadr exp))
                            (bt-parse-exp (caddr exp))))
         ((recmem)
          (make-bt-recmem static
                          eliminable
                          (bt-parse-exp (cadr exp))
                          (caddr exp)))
         (else (error "bt-parse : not an expression :" exp)))
       (if (number? exp)
           (make-bt-nat static eliminable exp)
           (make-varref exp)))))

;;-----------------------------------------------------------------
;; jump parsing
;; jump (list) -> jump (record)
;;-----------------------------------------------------------------

(define bt-parse-jump
  (lambda (jump)
    (case (car jump)
      ((goto)
       (make-bt-goto static eliminable (cadr jump)))
      ((if)
       (make-bt-if static
                   eliminable
                   (bt-parse-exp (cadr jump))
                   (caddr jump)
                   (cadddr jump)))
      ((return)
       (make-bt-return static eliminable (bt-parse-exp (cadr jump))))
      (else (error "bt-parse : not a jump :" jump)))))

;;-----------------------------------------------------------------
;; parse-prog alias (main)
;;-----------------------------------------------------------------

(define bt-parse bt-parse-prog)
```

B.3 Binding-time type checker

```
;;=================================================================
;; bt-check.scm
;; unit to bt-type check for specializer
;; check if the bt-type is well-formed
;;=================================================================
```

```
(load "bt-parser.scm")

(load "bt-table.scm")

(load "bt-lib.scm")

;;=================================================================
;; bt operation
;;=================================================================
;; get bt-val from btype
;; btype -> bt-val
;;-----------------------------------------------------------------

(define get-bt-val
  (lambda (btype)
    (variant-case btype
      (nat-btype (btval) btval)
      (access-btype (btval) btval)
      (array-btype (btval) btval)
      (rec-btype (btval) btval)
      (else (error "bt-check : unknown btype :" btype)))))

;;-----------------------------------------------------------------
;; get bt-mod from btype
;; btype -> bt-mod
;;-----------------------------------------------------------------

(define get-bt-mod
  (lambda (btype)
    (variant-case btype
      (nat-btype (btmod) btmod)
      (access-btype (btmod) btmod)
      (array-btype (btmod) btmod)
      (rec-btype (btmod) btmod)
      (else (error "bt-check : unknown btype :" btype)))))

;;-----------------------------------------------------------------
;; set ast constructs bt-val and bt-mod with given arguments
;;-----------------------------------------------------------------

(define set-bt!
  (lambda (ast val mod)
    (begin
      (vector-set! ast 1 val)
      (vector-set! ast 2 mod))))

;;-----------------------------------------------------------------
;; lattice operations on bt
;; bt-val x bt-mod x (bt-val x bt-mod) -> #t | #f
;;-----------------------------------------------------------------

(define bt-leq?
  (lambda (btval1 btmod1 btval2 btmod2)
    (or (and (static? btval1) (eliminable? btmod1))
        (and (static? btval1) (residual? btmod1)
             (or (and (static? btval2) (residual? btmod2))
                 (and (dynamic? btval2) (residual? btmod2))))
        (and (dynamic? btval1) (residual? btmod1)
             (and (dynamic? btval2) (residual? btmod2))))))

;;=================================================================
;; insert get-val, get-exp, lift, get-dyn, get-res
;;=================================================================
;; insert get-val
;;-----------------------------------------------------------------

(define insert-get-val!
```

```
  (lambda (arg exp)
    (begin
      (vector-set! arg 3 (make-get-val exp))
      (set-bt! arg static eliminable))))

;;-------------------------------------------------------------------
;; insert get-exp
;;-------------------------------------------------------------------

(define insert-get-exp!
  (lambda (arg exp)
    (begin
      (vector-set! arg 3 (make-get-exp exp))
      (set-bt! arg dynamic residual))))

;;-------------------------------------------------------------------
;; insert lift
;;-------------------------------------------------------------------

(define insert-lift!
  (lambda (arg)
    (variant-case arg
      (bt-assign (exp)
        (vector-set! arg 4 (make-lift exp)))
      (bt-arrayref (exp)
        (vector-set! arg 4 (make-lift exp)))
      (bt-return (exp)
        (vector-set! arg 3 (make-lift exp))))))

;;-------------------------------------------------------------------
;; insert get-dyn
;;-------------------------------------------------------------------

(define insert-get-dyn!
  (lambda (assign)
    (vector-set! assign 3 (make-get-dyn (bt-assign->lexp assign)))))

;;-------------------------------------------------------------------
;; insert get-res
;;-------------------------------------------------------------------

(define insert-get-res!
  (lambda (assign)
    (vector-set! assign 3 (make-get-res (bt-assign->lexp assign)))))

;;===================================================================
;; environment operation
;;===================================================================
;; all bt-val in environment are dynamic
;; table -> #t | #f
;;-------------------------------------------------------------------

(define all-dynamic?
  (lambda (table)
    (all-true? (map dynamic? (map get-bt-val (image-table table))))))

;;-------------------------------------------------------------------
;; exist residual bt-mod in environment
;; table -> #t | #f
;;-------------------------------------------------------------------

(define exist-residual?
  (lambda (table)
    (if (null? table)
        #f
        (let* ((decl   (car table))
               (btype  (cdr decl))
```

```
            (bt-mod (get-bt-mod btype)))
        (if (residual? bt-mod)
            #t
            (exist-residual? (cdr table)))))))

;;=================================================================
;; program bt-type checking
;; prog -> prog
;;=================================================================

(define bt-check-prog
  (lambda (parsed-prog)
    (variant-case parsed-prog
      (program (decls params blocks)
        (let* ((tmp (reset-btype-head!))
               (tmp (reset-btype-table!))
               (tmp (reset-btype-env!))
               (tmp (bt-check-decls decls))
               (tmp (bt-check-params params))
               (tmp (bt-check-blocks blocks))
               (tmp (newline))
               (tmp (display "** BTYPE CHECK COMPLETED **"))
               (tmp (newline))
               (tmp (newline))
               )
          ()))
      (else (error "bt-check : not a program :" parsed-prog)))))

;;-----------------------------------------------------------------
;; declarations bt-type checking
;; - builds global btype-head, btype-table and btype-env (side-effect)
;;-----------------------------------------------------------------

(define bt-check-decls
  (lambda (decls)
    (let ((tdefs (decls->tdefs decls))
          (vdecls (decls->vdecls decls)))
      (begin
        (set-btype-head!  (bt-check-head tdefs empty-table))
        (set-btype-table! (bt-check-tdef tdefs empty-table))
        (set-btype-env!   (bt-check-vdecl vdecls empty-table))))))

;;-----------------------------------------------------------------
;; type definition header scanning
;; tdefs x table -> table
;; - build table recursively for btype-head
;;-----------------------------------------------------------------

(define bt-check-head
  (lambda (tdefs table)
    (if (null? tdefs)
        table
        (let* ((tdef      (car tdefs))
               (name      (bt-tdef->name tdef))
               (btval     (bt-tdef->btval tdef))
               (btmod     (bt-tdef->btmod tdef))
               (bt        (cons btval btmod))
               (new-table (update-table name bt table)))
          (bt-check-head (cdr tdefs) new-table))))))

;;-----------------------------------------------------------------
;; type definitions bt-type checking
;; tdefs x table -> table
;; - build table recursively for btype-table
;;-----------------------------------------------------------------

(define bt-check-tdef
```

```
  (lambda (tdefs table)
    (if (null? tdefs)
        table
        (let* ((tdef (car tdefs))
               (name (bt-tdef->name tdef)))
          (if (exist-table? name table)
              (error "bt-check : variable" name 'exist 'in 'type 'definition)
              (let* ((btval      (bt-tdef->btval tdef))
                     (btmod      (bt-tdef->btmod tdef))
                     (vdecls     (bt-tdef->vdecls tdef))
                     (tdef-table (bt-check-vdecl vdecls empty-table)))
                (begin
                  (cond
                    ((and (dynamic? btval) (residual? btmod))
                     ;; all variables bt-type in vdecls are dynamic
                     (if (not (all-dynamic? tdef-table))
                         (error "bt-check : bad type definition :" tdef)))
                    ((and (static? btval) (residual? btmod))
                     ;; exist variables bt-type in vdecls is residual
                     (if (not (exist-residual? tdef-table))
                         (error "bt-check : bad type definition :" tdef))
                     )
                    ((and (static? btval) (eliminable? btmod))
                     ;;  do nothing
                     '())
                    (else (error "bt-check : bad btype" 'in tdef)))
                  (let ((new-table (update-table name tdef-table table)))
                    (bt-check-tdef (cdr tdefs) new-table)))))))))

;;-----------------------------------------------------------------
;; variable declarations bt-type checking
;; vdecls x table -> table
;; - build table recursively for btype-env
;;-----------------------------------------------------------------

(define bt-check-vdecl
  (lambda (vdecls table)
    (if (null? vdecls)
        table
        (let* ((vdecl (car vdecls))
               (name  (bt-vdecl->name vdecl)))
          (if (exist-table? name table)
              (error "bt-check : record" name 'exist 'in 'variable
                     'declaration)
              (let* ((btype-checked (bt-check-type (bt-vdecl->btype vdecl)))
                     (new-table     (update-table name btype-checked table))
                     (btval         (get-bt-val btype-checked))
                     (btmod         (get-bt-mod btype-checked)))
                (begin
                  (set-bt! vdecl btval btmod)
                  (bt-check-vdecl (cdr vdecls) new-table))))))))

;;-----------------------------------------------------------------
;; btype bt-type checking
;; btype -> btype
;; - recursively change declaration structure to environments
;;-----------------------------------------------------------------

(define bt-check-type
  (lambda (type)
    (variant-case type
      (nat-btype (btval btmod)
        (if (or (and (static? btval) (eliminable? btmod))
                (and (dynamic? btval) (residual? btmod)))
            type
            (error "bt-check : btype is not well-formed :" type)))
      (access-btype (btval btmod btype)
```

```
        (let* ((btype-checked (bt-check-type btype)))
          (if (bt-leq? btval btmod
                       (get-bt-val btype-checked) (get-bt-mod btype-checked))
              (make-access-btype btval btmod btype-checked)
              (error "bt-check : btype is not well-formed :" type))))
      (array-btype (btval btmod lower upper btype)
        (let* ((btype-checked (bt-check-type btype)))
          (if (bt-leq? btval btmod
                       (get-bt-val btype-checked) (get-bt-mod btype-checked))
              (make-array-btype btval btmod lower upper btype-checked)
              (error "bt-check : btype is not well-formed :" type))))
      (rec-btype (name)
        (let* ((bt    (lookup-btype-head name))
               (btval (car bt))
               (btmod (cdr bt))
               (tmp   (set-bt! type btval btmod)))
          (make-rec-btype btval btmod name)))
      (else (error "bt-check : unknown btype :" type)))))

;;-----------------------------------------------------------------
;; parameter bt-type checking
;;-----------------------------------------------------------------

(define bt-check-params
  (lambda (params)
    (if (not (null? params))
        (let* ((param (car params))
               (btype (lookup-btype-env param)))
          (if (nat-btype? btype)
              (bt-check-params (cdr params))
              (error "bt-check : non nat-btype parameter :" param))))))

;;-----------------------------------------------------------------
;; blocks bt-type checking
;;-----------------------------------------------------------------

(define bt-check-blocks
  (lambda (blocks)
    (map bt-check-block blocks)))

;;-----------------------------------------------------------------
;; (each) block bt-type checking
;;-----------------------------------------------------------------

(define bt-check-block
  (lambda (block)
    (let ((assigns (block->assigns block))
          (jump    (block->jump block)))
      (begin
        (bt-check-assigns assigns)
        (bt-check-jump jump)))))

;;-----------------------------------------------------------------
;; assignments bt-type checking
;;-----------------------------------------------------------------

(define bt-check-assigns
  (lambda (assigns)
    (map bt-check-assign assigns)))

;;-----------------------------------------------------------------
;; (each) assignment bt-type checking
;;-----------------------------------------------------------------

(define bt-check-assign
  (lambda (assign)
    (variant-case assign
```

```
      (bt-assign (lexp exp)
        (let* ((lexp-btype (bt-check-lexp assign lexp))
               (exp-btype  (bt-check-rexp exp))
               (btype      (bt-check-assign-equiv lexp-btype exp-btype))
               (btval      (get-bt-val btype))
               (btmod      (get-bt-mod btype)))
          (begin
            (set-bt! assign btval btmod)
            (if (nat-btype? btype)
                (if (and (dynamic? btval) (residual? btmod)
                         (static? (get-bt-val exp-btype))
                         (eliminable? (get-bt-mod exp-btype)))
                    (insert-lift! assign))))))
      (else (error "check : not an assignment :" assign)))))

;-----------------------------------------------------------------
; assignment equivalent bt-type checking
;-----------------------------------------------------------------

(define bt-check-assign-equiv
  (lambda (lbtype btype)
    (let* ((lbtval (get-bt-val lbtype))
           (lbtmod (get-bt-mod lbtype))
           (btval  (get-bt-val btype))
           (btmod  (get-bt-mod btype)))
      (cond
        ((and (nat-btype? lbtype) (nat-btype? btype))
         (cond
           ((and (and (static? lbtval) (eliminable? lbtmod))
                 (and (static? btval) (eliminable? btmod)))
            (make-nat-btype static eliminable))
           ((and (dynamic? lbtval) (residual? lbtmod))
            (make-nat-btype dynamic residual))
           (else (error "bt-check : not well-formed assign expressions :"
                        lbtype btype))))
        ((or (and (access-btype? lbtype) (access-btype? btype))
             (and (array-btype? lbtype) (array-btype? btype))
             (and (rec-btype? lbtype) (rec-btype? btype)))
         (if (equal-btype? lbtype btype)
             lbtype
             (error "bt-check : non equivalent btype in 'assign' :"
                    ltype type)))
        (else (error "bt-check : non equivalent btype in 'assign' :"
                     lbtype btype))))))

;;-----------------------------------------------------------------
;; left expression bt-type checking
;; lexp -> btype
;;-----------------------------------------------------------------
(define bt-check-lexp
  (lambda (assign lexp)
    (variant-case lexp
      (varref (var)
        (let* ((btype (lookup-btype-env var))
               (bt-val (get-bt-val btype))
               (bt-mod (get-bt-mod btype)))
          (begin
            (cond
              ((and (static? bt-val) (residual? bt-mod))
               ;; insert get-res for lexp
               (insert-get-res! assign))
              ((and (dynamic? bt-val) (residual? bt-mod))
               ;; insert get-dyn for lexp
               (insert-get-dyn! assign)))
            btype)))
      (bt-deref (exp)
        (let ((exp-btype (bt-check-rexp exp)))
```

```
    (variant-case exp-btype
      (access-btype (btval btmod btype)
        (begin
          ;; set bt-deref bt according to exp bt
          (set-bt! lexp btval btmod)
          (let* ((bt-val (get-bt-val btype))
                 (bt-mod (get-bt-mod btype)))
            (cond
              ((and (static? btval) (eliminable? btmod))
               (cond
                 ((and (static? bt-val) (residual? bt-mod))
                  ;; insert get-res for lexp
                  (insert-get-res! assign))
                 ((and (dynamic? bt-val) (residual? bt-mod))
                  ;; insert get-dyn for lexp
                  (insert-get-dyn! assign))))
              ((and (static? btval) (residual? btmod))
               (if (and (dynamic? bt-val) (residual? bt-mod))
                   ;; insert get-exp for bt-deref exp
                   (insert-get-exp! lexp exp)))))
            btype))
        (else (error "bt-check : non pointer type in 'deref' :" exp)))))
(bt-arrayref (vexp exp)
  (let ((vexp-btype (bt-check-rexp vexp))
        (exp-btype  (bt-check-rexp exp)))
    (variant-case vexp-btype
      (array-btype (btval btmod btype)
        (if (nat-btype? exp-btype)
            (let ((exp-btval (nat-btype->btval exp-btype))
                  (exp-btmod (nat-btype->btmod exp-btype)))
              (begin
                ;; set bt-arrayref bt according to vexp bt
                (set-bt! lexp btval btmod)
                (let* ((bt-val (get-bt-val btype))
                       (bt-mod (get-bt-mod btype)))
                  (cond
                    ((and (static? btval) (eliminable? btmod))
                     (if (and (static? exp-btval)
                              (eliminable? exp-btmod))
                         (cond
                           ((and (static? bt-val) (residual? bt-mod))
                            ;; insert get-res for lexp
                            (insert-get-res! assign))
                           ((and (dynamic? bt-val) (residual? bt-mod))
                            ;; insert get-dyn for lexp
                            (insert-get-dyn! assign)))
                         (error "bt-check : non SE nat in 'arrayref' :"
                                exp)))
                    ((and (static? btval) (residual? btmod))
                     (if (and (static? exp-btval)
                              (eliminable? exp-btmod))
                         (if (and (dynamic? bt-val) (residual? bt-mod))
                             ;; insert get-exp for bt-arrayref vexp
                             (insert-get-exp! lexp vexp))
                         (error "bt-check : non SE nat in 'arrayref' :"
                                exp)))
                    ((and (dynamic? btval) (residual? btmod))
                     (if (and (static? exp-btval)
                              (eliminable? exp-btmod))
                         ;; lift bt-arrayref exp
                         (insert-lift! lexp)))))
                  btype))
            (error "bt-check : non nat-btype in 'arrayref' :" exp)))
        (else (error "bt-check : non array-btype in 'arrayref' :" vexp)))))
(bt-recmem (exp var)
  (let ((exp-btype (bt-check-rexp exp)))
    (variant-case exp-btype
```

```
          (rec-btype (btval btmod name)
            (let* ((rec-env (lookup-btype-table (rec-btype->name exp-btype)))
                   (btype   (lookup-table var rec-env)))
              (begin
                ;; set bt-recmem bt according to exp bt
                (set-bt! lexp btval btmod)
                (let* ((bt-val (get-bt-val btype))
                       (bt-mod (get-bt-mod btype)))
                  (cond
                    ((and (static? btval) (eliminable? btmod))
                     (cond
                       ((and (static? bt-val) (residual? bt-mod))
                        ;; insert get-res for lexp
                        (insert-get-res! assign))
                       ((and (dynamic? bt-val) (residual? bt-mod))
                        ;; insert get-dyn for lexp
                        (insert-get-dyn! assign))))
                    ((and (static? btval) (residual? btmod))
                     (cond
                       ((and (static? bt-val) (eliminable? bt-mod))
                        ;; insert get-val for bt-recmem exp
                        (insert-get-val! lexp exp))
                       ((and (dynamic? bt-val) (residual? bt-mod))
                        ;; insert get-exp for bt-recmem exp
                        (insert-get-exp! lexp exp))))))
                btype)))
          (else (error "bt-check : non record-btype in 'recmem' :" exp)))))
      (else (error "bt-check : not a lexp :" lexp)))))

;;-------------------------------------------------------------------
;; right expression bt-type checking
;; rexp -> btype
;;-------------------------------------------------------------------
(define bt-check-rexp
  (lambda (rexp)
    (variant-case rexp
      (bt-nat () (make-nat-btype static eliminable))
      (varref (var) (lookup-btype-env var))
      (bt-app (btval btmod op exps)
        (let* ((btype (bt-check-op op exps))
               (btval (get-bt-val btype))
               (btmod (get-bt-mod btype)))
          (begin
            (set-bt! rexp btval btmod)
            btype)))
      (bt-deref (exp)
        (let ((exp-btype (bt-check-rexp exp)))
          (variant-case exp-btype
            (access-btype (btval btmod btype)
              (begin
                ;; set bt-deref bt according to exp bt
                (set-bt! rexp btval btmod)
                (let* ((bt-val (get-bt-val btype))
                       (bt-mod (get-bt-mod btype)))
                  (if (and (static? btval) (residual? btmod))
                      (if (and (dynamic? bt-val) (residual? bt-mod))
                          ;; insert get-exp for bt-deref exp
                          (insert-get-exp! rexp exp))))
                btype))
            (else (error "bt-check : non pointer type in 'deref' :" exp)))))
      (bt-new (btval btmod btype)
        (let ((btype-checked (bt-check-type btype)))
          (if (bt-leq? btval btmod
                       (get-bt-val btype-checked) (get-bt-mod btype-checked))
              (make-access-btype btval btmod btype-checked)
              (error "bt-check : btype is not well-formed in 'new' :" btype))))
      (bt-nil (btval btmod btype)
```

```
    (let ((btype-checked (bt-check-type btype)))
      (if (bt-leq? btval btmod
                   (get-bt-val btype-checked) (get-bt-mod btype-checked))
          (make-access-btype btval btmod btype-checked)
          (error "bt-check : btype is not well-formed in 'nil' :" btype))))
  (bt-arrayref (vexp exp)
    (let ((vexp-btype (bt-check-rexp vexp))
          (exp-btype  (bt-check-rexp exp)))
      (variant-case vexp-btype
        (array-btype (btval btmod btype)
          (if (nat-btype? exp-btype)
              (let ((exp-btval (nat-btype->btval exp-btype))
                    (exp-btmod (nat-btype->btmod exp-btype)))
                (begin
                  ;; set bt-arrayref bt according to vexp bt
                  (set-bt! rexp btval btmod)
                  (let* ((bt-val (get-bt-val btype))
                         (bt-mod (get-bt-mod btype)))
                    (cond
                      ((and (static? btval) (eliminable? btmod))
                       (if (not (and (static? exp-btval)
                                     (eliminable? exp-btmod)))
                           (error "bt-check : non SE nat in 'arrayref' :"
                                  exp)))
                      ((and (static? btval) (residual? btmod))
                       (if (and (static? exp-btval)
                                (eliminable? exp-btmod))
                           (if (and (dynamic? bt-val) (residual? bt-mod))
                               ;; insert get-exp for bt-arrayref vexp
                               (insert-get-exp! rexp vexp))
                           (error "bt-check : non SE nat in 'arrayref' :"
                                  exp)))
                      ((and (dynamic? btval) (residual? btmod))
                       (if (and (static? exp-btval)
                                (eliminable? exp-btmod))
                           ;; lift bt-arrayref exp
                           (insert-lift! rexp)))))
                  btype))
              (error "bt-check : non nat-btype in 'arrayref' :" exp)))
        (else (error "bt-check : non array-btype in 'arrayref' :" vexp)))))
  (bt-recmem (exp var)
    (let ((exp-btype (bt-check-rexp exp)))
      (variant-case exp-btype
        (rec-btype (btval btmod name)
          (let* ((rec-env (lookup-btype-table (rec-btype->name exp-btype)))
                 (btype   (lookup-table var rec-env)))
            (begin
              ;; set bt-recmem bt according to exp bt
              (set-bt! rexp btval btmod)
              (let* ((bt-val (get-bt-val btype))
                     (bt-mod (get-bt-mod btype)))
                (if (and (static? btval) (residual? btmod))
                    (cond
                      ((and (static? bt-val) (eliminable? bt-mod))
                       ;; insert get-val for bt-recmem exp
                       (insert-get-val! rexp exp))
                      ((and (dynamic? bt-val) (residual? bt-mod))
                       ;; insert get-exp for bt-recmem exp
                       (insert-get-exp! rexp exp)))))
              btype)))
        (else (error "bt-check : non record-btype in 'recmem' :" exp)))))
  (else (error "bt-check : not a rexp :" rexp))))

;;-----------------------------------------------------------------
;; operation bt-type checking
;; op x exps -> btype
;;-----------------------------------------------------------------
```

```
(define bt-check-op
  (lambda (op exps)
    (let* ((exp1   (car exps))
           (btype1 (bt-check-rexp exp1))
           (btval1 (get-bt-val btype1))
           (btmod1 (get-bt-mod btype1))
           (exp2   (cadr exps))
           (btype2 (bt-check-rexp exp2))
           (btval2 (get-bt-val btype2))
           (btmod2 (get-bt-mod btype2)))
      (case op
        ((+ - * / > gte < lte)
         (if (and (nat-btype? btype1) (nat-btype? btype2))
             (if (and (and (static? btval1) (eliminable? btmod1))
                      (and (static? btval2) (eliminable? btmod2)))
                 (make-nat-btype static eliminable)
                 (begin
                   (cond
                     ((and (static? btval1) (eliminable? btmod1))
                      ;; lift exps firts argument
                      (set-car! exps (make-lift exp1)))
                     ((and (static? btval2) (eliminable? btmod2))
                      ;; lift exps second argument
                      (set-car! (cdr exps) (make-lift exp2))))
                   (make-nat-btype dynamic residual)))
             (error "bt-check : nat-btype expected in 'op' :" op)))
        ((= neq)
         (cond
           ((and (nat-btype? btype1) (nat-btype? btype2))
            (if (and (and (static? btval1) (eliminable? btmod1))
                     (and (static? btval2) (eliminable? btmod2)))
                (make-nat-btype static eliminable)
                (begin
                  (cond
                    ((and (static? btval1) (eliminable? btmod1))
                     ;; lift exps firts argument
                     (set-car! exps (make-lift exp1)))
                    ((and (static? btval2) (eliminable? btmod2))
                     ;; lift exps second argument
                     (set-car! (cdr exps) (make-lift exp2))))
                  (make-nat-btype dynamic residual))))
           ((and (access-btype? btype1) (access-btype? btype2))
            (if (equal-btype? (access-btype->btype btype1)
                              (access-btype->btype btype2))
                (cond
                  ((and (and (static? btval1) (eliminable? btmod1))
                        (and (static? btval2) (eliminable? btmod2)))
                   (make-nat-btype static eliminable))
                  ((and (equal? btval1 btval2)
                        (and (residual? btmod1) (residual? btmod2)))
                   (make-nat-btype dynamic residual))
                  (else (error "bt-check  : non equivalent btype in 'op' :"
                               op)))
                (else (error "bt-check  : non equivalent btype in 'op' :" op))
                ))
           ((and (array-btype? btype1) (array-btype? btype2))
            (let ((array-btype1 (array-btype->btype btype1))
                  (array-btype2 (array-btype->btype btype2)))
              (if (and (equal? (array-btype->lower btype1)
                               (array-btype->lower btype2))
                       (equal? (array-btype->upper btype1)
                               (array-btype->upper btype2))
                       (equal-btype? array-btype1 array-btype2))
                  (cond
                    ((and (and (static? btval1) (eliminable? btmod1))
                          (and (static? btval2) (eliminable? btmod2)))
```

```
                    (if (eliminable? (get-bt-val array-btype1))
                      (make-nat-btype static eliminable)
                      (error "bt-check : non eliminable array btype"
                             array-btype1 'in "op" ': op)))
                    ((and (equal? btval1 btval2)
                          (and (residual? btmod1) (residual? btmod2)))
                     (make-nat-btype dynamic residual))
                    (else (error "bt-check  : non equivalent btype in 'op' :"
                                 op)))
                  (else (error "bt-check  : non equivalent btype in 'op' :"
                               op)))))
            ((and (rec-btype? btype1) (rec-btype? btype2))
             (let ((rec-name1 (rec-btype->name btype1))
                   (rec-name2 (rec-btype->name btype2)))
               (if (equal? rec-name1 rec-name2)
                 (cond
                   ((eliminable? btmod1)
                    (let ((rec-env (lookup-btype-table rec-name1)))
                      (if (all-true? (map eliminable?
                                          (map get-bt-mod (map cdr rec-env))))
                        (make-nat-btype static eliminable)
                        (error "bt-check : non eliminable record members"
                               rec-name1 'in "op" ': op))))
                   ((residual? bmod1)
                    (make-nat-btype dynamic residual))
                   (else (error "bt-check  : non equivalent btype in 'op' :"
                                op)))
                 (else (error "bt-check  : non equivalent btype in 'op' :"
                              op)))))
            (else (error "bt-check  : non equivalent btype in 'op' :" op))))
          (else (error "bt-check : not an operator :" op))))))

;-----------------------------------------------------------------
; equivalence check for type in btype
; btype x btype -> #t | #f
;-----------------------------------------------------------------

(define equal-btype?
  (lambda (btype1 btype2)
    (let* ((btval1 (get-bt-val btype1))
           (btmod1 (get-bt-mod btype1))
           (btval2 (get-bt-val btype2))
           (btmod2 (get-bt-mod btype2)))
      (if (and (equal? btval1 btval2)
               (equal? btmod1 btmod2))
          (cond
            ((and (nat-btype? btype1) (nat-btype? btype2)) #t)
            ((and (access-btype? btype1) (access-btype? btype2))
             (equal-btype? (access-btype->btype btype1)
                           (access-btype->btype btype2)))
            ((and (array-btype? btype1) (array-btype? btype2))
             (and (equal? (array-btype->lower btype1)
                          (array-btype->lower btype2))
                  (equal? (array-btype->upper btype1)
                          (array-btype->upper btype2))
                  (equal-btype? (array-btype->btype btype1)
                                (array-btype->btype btype2))))
            ((and (rec-btype? btype1) (rec-btype? btype2))
             (equal? (rec-btype->name btype1) (rec-btype->name btype2)))
            (else #f))
          #f))))

;-----------------------------------------------------------------
; jump bt-type checking
;-----------------------------------------------------------------

(define bt-check-jump
```

```
  (lambda (jump)
    (variant-case jump
      ;; goto is always static and eliminable
      (bt-goto (label) ())
      (bt-if (exp then-label else-label)
        (let ((exp-btype (bt-check-rexp exp)))
          (if (nat-btype? exp-btype)
              (let ((btval (nat-btype->btval exp-btype))
                    (btmod (nat-btype->btmod exp-btype)))
                (if (and (dynamic? btval) (residual? btmod))
                    ;; set bt-if bt according to exp
                    (set-bt! jump btval btmod)))
              (error "bt-check : non nat-btype in 'if' :" exp))))
      (bt-return (exp)
        (let ((exp-btype (bt-check-rexp exp)))
          (if (nat-btype? exp-btype)
              (let ((btval (nat-btype->btval exp-btype))
                    (btmod (nat-btype->btmod exp-btype)))
                (begin
                  ;; set bt-return bt to DR
                  (set-bt! jump dynamic residual)
                  (if (and (static? btval) (eliminable? btmod))
                      ;; lift bt-return exp
                      (insert-lift! jump))))
              (error "bt-check : non nat-btype in 'return' :" exp))))
      (else (error "bt-check : not a jump :" jump)))))

;-----------------------------------------------------------------
; bt-check-prog alias (main)
;-----------------------------------------------------------------

(define bt-check bt-check-prog)
```

B.4 Specializer

```
;;=================================================================
;; bt-spec.scm
;; unit to specialize the program
;;=================================================================

(load "bt-check.scm")
(load "bt-unparser.scm")

;;=================================================================
;; halt label
;;=================================================================

(define-record halt ())

;;=================================================================
;; global label
;;=================================================================

(define next-label!
  (let ((lab 1))
    (lambda (comm)
      (case comm
        ((next)
         (let ((lab-tmp lab)
               (tmp (set! lab (+ lab 1))))
           (string->symbol (string-append "lab-" (number->string lab-tmp)))))
        ((reset)
         (let ((lab-tmp lab)
               (tmp (set! lab 1)))
           lab-tmp))))))
```

```
;;================================================================
;; global table pending-list
;;================================================================
;; pending-item record definition
;;----------------------------------------------------------------

(define-record pending-item (old-label new-label saved-store))

;;----------------------------------------------------------------
;; create new empty global pending-list
;;----------------------------------------------------------------

(define pending-list empty-table)

;;----------------------------------------------------------------
;; reset pending-list
;;----------------------------------------------------------------

(define reset-pending-list!
  (lambda ()
    (set! pending-list empty-table)))

;;----------------------------------------------------------------
;; insert new pending-item into (head) pending-list
;;----------------------------------------------------------------

(define insert-pending-list!
  (lambda (old-label new-label store)
    (let* ((pending-item (make-pending-item old-label new-label store)))
      (set! pending-list (cons pending-item pending-list)))))

;;----------------------------------------------------------------
;; return next entry (head) from pending-list
;;----------------------------------------------------------------

(define next-pending-list!
  (lambda ()
    (if (null? pending-list)
        #f
        (let ((item (car pending-list))
              (tmp  (set! pending-list (cdr pending-list))))
          item))))

;;================================================================
;; global table seenB4-list
;;================================================================
;; seenB4-item record definition
;;----------------------------------------------------------------

(define-record seenB4-item (old-label new-label saved-store))

;;----------------------------------------------------------------
;; create new empty global seenB4-list
;;----------------------------------------------------------------

(define seenB4-list empty-table)

;;----------------------------------------------------------------
;; reset seenB4-list
;;----------------------------------------------------------------

(define reset-seenB4-list!
  (lambda ()
    (set! seenB4-list empty-table)))

;;----------------------------------------------------------------
```

```
;; insert new seenB4-item into (head) seenB4-list
;;-----------------------------------------------------------------

(define insert-seenB4-list!
  (lambda (old-label new-label store)
    (let* ((seenB4-item (make-seenB4-item old-label new-label store)))
      (set! seenB4-list (cons seenB4-item seenB4-list)))))

;;-----------------------------------------------------------------
;; searh for seenB4-item in seenB4-list
;; label x store -> label | #f
;;-----------------------------------------------------------------

(define exist-seenB4-item?
  (lambda (label store)
    (letrec ((loop (lambda (list)
                     (if (null? list)
                         #f
                         (let* ((item (car list))
                                (old-label (seenB4-item->old-label item))
                                (saved-store (seenB4-item->saved-store item)))
                           (if (and (equal? label old-label)
                                    (equal? store saved-store))
                               (seenB4-item->new-label item)
                               (loop (cdr list))))))))
      (loop seenB4-list))))

;;=================================================================
;; global table residual tipe definition
;;=================================================================
;; create new empty table residual-tdefs
;;-----------------------------------------------------------------

(define residual-tdefs empty-table)

;;-----------------------------------------------------------------
;; reset residual-tdefs
;;-----------------------------------------------------------------

(define reset-residual-tdefs!
  (lambda ()
    (set! residual-tdefs empty-table)))

;;-----------------------------------------------------------------
;; update residual-tdefs
;; - insert a new tipe definition in front of residual-tdefs
;;-----------------------------------------------------------------

(define insert-residual-tdefs!
  (lambda (tdef)
    (begin
      (set! residual-tdefs (cons tdef residual-tdefs)))))

;;-----------------------------------------------------------------
;; retrieved residual-tdefs (reversed)
;;-----------------------------------------------------------------

(define retrieve-residual-tdefs
  (lambda ()
    (reverse residual-tdefs)))

;;=================================================================
;; global table residual variable declarations
;;=================================================================
;; create new empty table residual-vdecls
;;-----------------------------------------------------------------
```

```
(define residual-vdecls empty-table)

;;-------------------------------------------------------------
;; reset residual-vdecls
;;-------------------------------------------------------------

(define reset-residual-vdecls!
  (lambda ()
    (set! residual-vdecls empty-table)))

;;-------------------------------------------------------------
;; update residual-vdecls
;; - insert a new variable declaration in front of residual-vdecls
;;-------------------------------------------------------------

(define insert-residual-vdecls!
  (lambda (vdecl)
    (begin
      (set! residual-vdecls (cons vdecl residual-vdecls)))))

;;-------------------------------------------------------------
;; retrieved residual-vdecls (reversed)
;;-------------------------------------------------------------

(define retrieve-residual-vdecls
  (lambda ()
    (reverse residual-vdecls)))

;;=============================================================
;; global table residual blocks
;;=============================================================
;; create new empty global residual-blocks
;;-------------------------------------------------------------

(define residual-blocks empty-table)

;;-------------------------------------------------------------
;; reset residual-blocks
;;-------------------------------------------------------------

(define reset-residual-blocks!
  (lambda ()
    (set! residual-blocks empty-table)))

;;-------------------------------------------------------------
;; update residual-blocks
;; - insert new block in front of residual-blocks
;;-------------------------------------------------------------

(define insert-residual-blocks!
  (lambda (block)
    (begin
      (set! residual-blocks (cons block residual-blocks)))))

;;-------------------------------------------------------------
;; retrieve residual-blocks (reversed)
;;-------------------------------------------------------------

(define retrieve-residual-blocks
  (lambda ()
    (reverse residual-blocks)))

;;=============================================================
;; global table residual assignments
;;=============================================================
;; create new empty residual-assigns
;;-------------------------------------------------------------
```

```
(define residual-assigns empty-table)

;;-----------------------------------------------------------------
;; reset residual-assigns
;;-----------------------------------------------------------------

(define reset-residual-assigns!
  (lambda ()
    (set! residual-assigns empty-table)))

;;-----------------------------------------------------------------
;; update residual-assigns
;; - insert new assignment in front of residual-assigns
;;-----------------------------------------------------------------

(define insert-residual-assigns!
  (lambda (assign)
    (begin
      (set! residual-assigns (cons assign residual-assigns)))))

;;-----------------------------------------------------------------
;; retrieve current-assignments (reversed)
;;-----------------------------------------------------------------

(define retrieve-residual-assigns
  (lambda ()
    (reverse residual-assigns)))

;;=================================================================
;; values
;;=================================================================

(define-record nat-val   (val))
(define-record loc-val   (loc))
(define-record array-val (map lower upper))
(define-record rec-val   (env))

;;-----------------------------------------------------------------
;; initial value for nat-val
;; -> val
;;-----------------------------------------------------------------

(define init-nat
  (lambda ()
    (make-nat-val 0)))

;;-----------------------------------------------------------------
;; initial value for loc-val
;; -> val
;;-----------------------------------------------------------------

(define init-loc
  (lambda ()
    (make-loc-val (make-nil-val))))

;;-----------------------------------------------------------------
;; nil value for loc
;;-----------------------------------------------------------------

(define-record nil-val ())

;;=================================================================
;; program specialization
;;=================================================================

(define spec-prog
```

```
  (lambda (parsed-prog args)
    (let* ((tmp        (reset-env!))
           (tmp        (reset-store!))
           (tmp        (next-loc! 'reset))
           (tmp        (next-label! 'reset))
           (tmp        (reset-pending-list!))
           (tmp        (reset-seenB4-list!))
           (tmp        (reset-residual-tdefs!))
           (tmp        (reset-residual-vdecls!))
           (tmp        (reset-residual-blocks!))
           (decls      (program->decls parsed-prog))
           (tdefs      (decls->tdefs decls))
           (vdecls     (decls->vdecls decls))
           (params     (program->params parsed-prog))
           (blocks     (program->blocks parsed-prog))
           (init-label (block->label (car blocks)))

           (tmp        (spec-tdefs tdefs))
           (init-env   (spec-vdecls vdecls empty-table))
           (tmp        (set-env! init-env))
           (res-params (spec-params params args))
           (tmp (insert-pending-list! init-label (next-label! 'next) store))
           (tmp (spec-blocks blocks))

           (tmp (newline))
           (tmp (display "** BTYPE CHECK COMPLETED **"))
           (tmp (newline))
           (tmp (newline))
           )
      (unparse (make-program (make-decls (retrieve-residual-tdefs)
                                         (retrieve-residual-vdecls))
                             res-params
                             (retrieve-residual-blocks))))))

;;-------------------------------------------------------------------
;; type definitions specialization
;;-------------------------------------------------------------------

(define spec-tdefs
  (lambda (tdefs)
    (if (not (null? tdefs))
        (let* ((tdef   (car tdefs))
               (name   (bt-tdef->name tdef))
               (btmod  (bt-tdef->btmod tdef))
               (vdecls (bt-tdef->vdecls tdef)))
          (begin
            (if (residual? btmod)
                (let* ((res-vdecls
                        (letrec ((loop (lambda (vdecls table)
                                         (if (null? vdecls)
                                             (reverse table)
                                             (let* ((vdecl (car vdecls))
                                                    (name (bt-vdecl->name vdecl))
                                                    (type (bt-vdecl->btype vdecl))
                                                    (new-table (spec-cdecl name
                                                                           type
                                                                           table)))
                                               (loop (cdr vdecls) new-table))))))
                          (loop vdecls empty-table))))
                  (insert-residual-tdefs! (make-tdef name res-vdecls))))
            (spec-tdefs (cdr tdefs)))))))

;;-------------------------------------------------------------------
;; declaration generation specialization
;;-------------------------------------------------------------------

(define spec-cdecl
```

```
  (lambda (var btype res-vdecls)
    (let ((btmod (get-bt-mod btype)))
      (if (residual? btmod)
          (let* ((type (spec-type btype))
                 (vdecl (make-vdecl var type))
                 (new-res-vdecls (cons vdecl res-vdecls)))
            new-res-vdecls)
          res-vdecls))))

;;-------------------------------------------------------------------
;; variable declarations specialization
;;-------------------------------------------------------------------

(define spec-vdecls
  (lambda (vdecls var-env)
    (if (null? vdecls)
        var-env
        (let* ((vdecl (car vdecls))
               (type (bt-vdecl->btype vdecl))
               (res-vdecl (spec-cdecl (next-loc! 'get) type residual-vdecls))
               (tmp (set! residual-vdecls res-vdecl))
               (new-var-env (spec-vdecl vdecl var-env)))
          (spec-vdecls (cdr vdecls) new-var-env)))))

;;-------------------------------------------------------------------
;; (each) variable declaration specialization
;;-------------------------------------------------------------------

(define spec-vdecl
  (lambda (vdecl var-env)
    (let* ((var (bt-vdecl->name vdecl))
           (type (bt-vdecl->btype vdecl))
           (btval (bt-vdecl->btval vdecl))
           (btmod (bt-vdecl->btmod vdecl))
           (new-loc (next-loc! 'next))
           (new-var-env (update-table var new-loc var-env)))
      (begin
        (cond
          ((and (static? btval) (eliminable? btmod))
           (let ((new-val (spec-init-val type)))
             (update-store! new-loc new-val)))
          ((and (dynamic? btval) (residual? btmod))
           (update-store! new-loc (make-varref new-loc)))
          ((and (static? btval) (residual? btmod))
           (let ((new-val (spec-init-val type)))
             (update-store! new-loc (cons (make-varref new-loc) new-val)))))
        new-var-env))))

;;-------------------------------------------------------------------
;; initial value specialization
;;-------------------------------------------------------------------

(define spec-init-val
  (lambda (type)
    (variant-case type
      (nat-btype (btval btmod)
        (if (and (static? btval) (eliminable? btmod))
            (init-nat)))
      (access-btype (btval)
        (if (static? btval)
            (init-loc)))
      (array-btype (btval btmod lower upper btype)
        (if (static? btval)
          (let* ((bt-val (get-bt-val btype))
                 (bt-mod (get-bt-mod btype))
                 (amap (letrec ((loop (lambda (index amap)
                                        (if (> index upper)
```

```
                                    amap
                                    (let* ((new-index (+ index 1))
                                           (vdecl (make-bt-vdecl bt-val
                                                                 bt-mod
                                                                 index
                                                                 btype)))
                                      (begin
                                        (if (eliminable? btmod)
                                          (let ((res-vdecl (spec-cdecl
                                                             (next-loc! 'get)
                                                             btype
                                                             residual-vdecls)))
                                            (set! residual-vdecls res-vdecl)))
                                        (let ((new-amap (spec-vdecl vdecl
                                                                    amap)))
                                          (loop new-index new-amap))))))))
                          (loop lower empty-table))))
              (make-array-val amap lower upper))))
      (rec-btype (btval btmod name)
        (if (static? btval)
          (let* ((decls (lookup-btype-table name))
                 (renv (letrec ((loop (lambda (decls renv)
                                        (if (null? decls)
                                          renv
                                          (let* ((decl (car decls))
                                                 (var (car decl))
                                                 (btype (cdr decl))
                                                 (bt-val (get-bt-val btype))
                                                 (bt-mod (get-bt-mod btype))
                                                 (vdecl (make-bt-vdecl bt-val
                                                                       bt-mod
                                                                       var
                                                                       btype)))
                                            (begin
                                              (if (eliminable? btmod)
                                                (let ((res-vdecl (spec-cdecl
                                                                   (next-loc! 'get)
                                                                   btype
                                                                   residual-vdecls)))
                                                  (set! residual-vdecls res-vdecl)))
                                              (let ((new-renv (spec-vdecl vdecl
                                                                          renv)))
                                                (loop (cdr decls) new-renv))))))))
                         (loop decls empty-table))))
            (make-rec-val renv))))
        (else (error "eval : unknown type :" type)))))

;;-----------------------------------------------------------------
;; spec-type
;;-----------------------------------------------------------------

(define spec-type
  (lambda (type)
    (variant-case type
      (nat-btype (btval btmod)
        (if (and (dynamic? btval) (residual? btmod))
            (make-nat-type)))
      (access-btype (btmod btype)
        (if (residual? btmod)
            (make-access-type (spec-type btype))))
      (array-btype (btmod lower upper btype)
        (if (residual? btmod)
            (make-array-type lower upper (spec-type btype))))
      (rec-btype (btmod name)
        (if (residual? btmod)
            (make-rec-type name)))
      (else (error "spec : unknown btype :" type)))))
```

```
;;-----------------------------------------------------------------
;; parameters specialization
;;-----------------------------------------------------------------

(define spec-params
  (lambda (params args)
    (letrec ((loop (lambda (params tagged-args)
                     (if (null? params)
                         ()
                         (let* ((param (car params))
                                (btype (lookup-btype-env param))
                                (btval (get-bt-val btype))
                                (btmod (get-bt-mod btype)))
                           (if (and (static? btval) (eliminable? btmod))
                               (begin
                                 (update-store! (lookup-env param)
                                                (car tagged-args))
                                 (loop (cdr params) (cdr tagged-args)))
                               (let ((var (lookup-store (lookup-env param))))
                                 (cons (varref->var var)
                                       (loop (cdr params) tagged-args)))))))))
      (loop params (map make-nat-val args)))))

;;-----------------------------------------------------------------
;; blocks specialization
;;-----------------------------------------------------------------

(define spec-blocks
  (lambda (blocks)
    (letrec ((loop (lambda ()
                (if (not (null? pending-list))
                    (let* ((item        (next-pending-list!))
                           (old-label   (pending-item->old-label item))
                           (new-label   (pending-item->new-label item))
                           (saved-store (pending-item->saved-store item))
                           (tmp         (insert-seenB4-list! old-label
                                                             new-label
                                                             saved-store))
                           (tmp         (reset-residual-assigns!))
                           (tmp         (set-store! saved-store))
                           (jump  (spec-block (get-block old-label blocks)
                                              blocks))
                           (block (make-block new-label
                                              (retrieve-residual-assigns)
                                              jump))
                           (tmp   (insert-residual-blocks! block)))
                      (loop))))))
      (loop))))

;;-----------------------------------------------------------------
;; get block
;;-----------------------------------------------------------------

(define get-block
  (lambda (label blocks)
    (if (null? blocks)
        (error "spec : unknown label :" label)
        (if (equal? label (block->label (car blocks)))
            (car blocks)
            (get-block label (cdr blocks))))))

;;-----------------------------------------------------------------
;; (each) block specialization
;;-----------------------------------------------------------------

(define spec-block
```

```
  (lambda (block blocks)
    (let* ((tmp (spec-assigns (block->assigns block)))
           (jump-return (spec-jump (block->jump block) blocks))
           (jump-label (car jump-return))
           (jump-code (cdr jump-return)))
      (variant-case jump-code
        (goto ()
          (spec-block (get-block jump-label blocks) blocks))
        (if (exp then-label else-label)
            (let* ((seen-then (exist-seenB4-item? then-label store))
                   (seen-else (exist-seenB4-item? else-label store))
                   (new-then (if seen-then
                                 seen-then
                                 (let* ((label (next-label! 'next))
                                        (tmp (insert-pending-list! then-label
                                                                   label
                                                                   store)))
                                   label)))
                   (new-else (if seen-else
                                 seen-else
                                 (let* ((label (next-label! 'next))
                                        (tmp (insert-pending-list! else-label
                                                                   label
                                                                   store)))
                                   label))))
              (make-if exp new-then new-else)))
        (return ()
          jump-code)))))

;;---------------------------------------------------------------------
;; assignments specialization
;;---------------------------------------------------------------------

(define spec-assigns
  (lambda (assigns)
    (if (not (null? assigns))
        (begin
          (spec-assign (car assigns))
          (spec-assigns (cdr assigns))))))

;;---------------------------------------------------------------------
;; (each) assignment specialization
;;---------------------------------------------------------------------

(define spec-assign
  (lambda (assign)
    (variant-case assign
      (bt-assign (btval btmod lexp exp)
         (let* ((val  (spec-rexp exp))
                (lval (spec-lexp lexp)))
           (cond
             ((and (static? btval) (eliminable? btmod))
              (spec-update lval val))
             ((and (dynamic? btval) (residual? btmod))
              (insert-residual-assigns! (make-assign lval val)))
             ((and (static? btval) (residual? btmod))
              (begin
                (insert-residual-assigns! (make-assign (car lval) (car val)))
                (spec-update (cdr lval) (cdr val)))))))))))

;;---------------------------------------------------------------------
;; assignment update specialization
;;---------------------------------------------------------------------

(define spec-update
  (lambda (loc val)
    (letrec ((loop (lambda (lenv env)
```

```
                    (if (not (null? env))
                        (begin
                          (spec-update (cdar lenv)
                                       (lookup-store (cdar env)))
                          (loop (cdr lenv) (cdr env))))))
             (loop-res (lambda (lenv env)
                         (if (not (null? env))
                             (let ((val (if (pair? (lookup-store (cdar env)))
                                            (cdr (lookup-store (cdar env)))
                                            (lookup-store (cdar env)))))
                               (begin
                                 (if (not (varref? val))
                                     (spec-update (cdar lenv) val))
                                 (loop-res (cdr lenv) (cdr env))))))))
      (if (pair? val)
          (variant-case (cdr val)
            (loc-val ()
              (let ((lval (lookup-store loc)))
                (begin
                  (insert-residual-assigns! (make-assign (car lval) (car val)))
                  (update-store! loc (cons (car lval) (cdr val)))
                  )))
            (array-val (map lower upper)
              (let* ((lval (lookup-store loc))
                     (lmap (array-val->map (cdr lval)))
                     (tmp  (insert-residual-assigns!
                             (make-assign (car lval) (car val)))))
                (loop-res lmap map)))
            (rec-val (env)
              (let* ((lval (lookup-store loc))
                     (lenv (rec-val->env (cdr lval)))
                     (tmp  (insert-residual-assigns!
                              (make-assign (car lval) (car val)))))
                (loop-res lenv env))))
          (variant-case val
            (varref (var)
              (insert-residual-assigns! (make-assign (lookup-store loc) val)))
            (nat-val ()
              (update-store! loc val))
            (loc-val ()
              (let ((lval (lookup-store loc)))
                (if (pair? lval)
                    (update-store! loc (cons (car lval) val))
                    (update-store! loc val))))
            (array-val (map lower upper)
              (let ((lval (lookup-store loc))
                (if (pair? lval)
                    (let ((lmap (array-val->map (cdr lval))))
                      (loop-res lmap map))
                    (let ((lmap (array-val->map lval)))
                      (loop lmap map))))))
            (rec-val (env)
              (let ((lval (lookup-store loc)))
                (if (pair? lval)
                    (let ((lenv (rec-val->env (cdr lval))))
                      (loop-res lenv env))
                    (let ((lenv (rec-val->env lval)))
                      (loop lenv env)))))
            (else (error "spec : unknown val in 'assign update' :" val)))))))

;;----------------------------------------------------------------
;; left expression specialization
;;----------------------------------------------------------------

(define spec-lexp
  (lambda (lexp)
    (variant-case lexp
```

```
      (varref (var) (lookup-env var))
      (get-dyn (exp)
        (let ((loc (spec-lexp exp)))
          (lookup-store loc)))
      (get-res (exp)
        (let* ((loc  (spec-lexp exp))
               (val (lookup-store loc)))
          (cons (car val) loc)))
      (bt-deref (btval btmod exp)
        (let ((val (spec-rexp exp)))
          (cond
            ((and (static? btval) (eliminable? btmod))
             (let ((loc (loc-val->loc val)))
               (if (nil-val? loc)
                   (error "spec : deref of nil val in 'lexp' :" exp)
                   loc)))
            ((and (dynamic? btval) (residual? btmod))
             (make-deref val))
            ((and (static? btval) (residual? btmod))
             (let ((loc (loc-val->loc (cdr val))))
               (if (nil-val? loc)
                   (error "spec : deref of nil val in 'lexp' :" exp)
                   (cons (make-deref (car val)) loc)))))))
      (bt-arrayref (btval btmod vexp exp)
        (let ((val     (spec-rexp vexp))
              (ind-val (spec-rexp exp)))
          (cond
            ((and (static? btval) (eliminable? btmod))
             (let ((ind  (nat-val->val ind-val)))
               (if (and (>= ind (array-val->lower val))
                        (<= ind (array-val->upper val)))
                   (lookup-table ind (array-val->map val))
                   (error "spec : array index out of range :" ind))))
            ((and (dynamic? btval) (residual? btmod))
             (make-arrayref val ind-val))
            ((and (static? btval) (residual? btmod))
             (let ((ind (nat-val->val ind-val)))
               (if (and (>= ind (array-val->lower val))
                        (<= ind (array-val->upper val)))
                   (cons (make-arrayref (car val) ind)
                         (lookup-table ind (array-val->map (cdr val))))
                   (error "spec : array index out of range :" ind)))))))
      (bt-recmem (btval btmod exp var)
        (let ((val (spec-rexp exp)))
          (cond
            ((and (static? btval) (eliminable? btmod))
             (lookup-table var (rec-val->env val)))
            ((and (dynamic? btval) (residual? btmod))
             (make-recmem val var))
            ((and (static? btval) (residual? btmod))
             (cons (make-recmem (car val) var)
                   (lookup-table var (rec-val->env (cdr val))))))))
      (else (error "spec : unknown lexp :" lexp)))))

;;-----------------------------------------------------------------
;; expression specialization
;;-----------------------------------------------------------------

(define spec-rexp
  (lambda (rexp)
    (variant-case rexp
      (bt-nat (datum) (make-nat-val datum))
      (lift (exp)
        (let ((val (spec-rexp exp)))
          (make-nat (nat-val->val val))))
      (get-val (exp)
        (let ((val (spec-rexp exp)))
```

```
    (cdr val)))
(get-exp (exp)
  (let ((val (spec-rexp exp)))
    (car val)))
(varref (var) (lookup-store (lookup-env var)))
(bt-app (btval btmod op exps)
  (let ((val1 (spec-rexp (car exps)))
        (val2 (spec-rexp (cadr exps))))
    (cond
      ((and (static? btval) (eliminable? btmod))
       (spec-op op val1 val2))
      ((and (dynamic? btval) (residual? btmod))
       (make-app op (list val1 val2))))))
(bt-deref (btval btmod exp)
  (let ((val (spec-rexp exp)))
    (cond
      ((and (static? btval) (eliminable? btmod))
       (let ((loc (loc-val->loc val)))
         (if (nil-val? loc)
             (error "spec : deref of nil val in 'rexp' :" exp)
             (lookup-store loc))))
      ((and (dynamic? btval) (residual? btmod))
       (make-deref val))
      ((and (static? btval) (residual? btmod))
       (let ((loc (loc-val->loc (cdr val))))
         (if (nil-val? loc)
             (error "spec : deref of nil val in 'rexp' :" exp)
             (cons (make-deref (car val)) (cdr (lookup-store loc)))))))))
(bt-new (btval btmod btype)
  (cond
    ((and (static? btval) (eliminable? btmod))
     (let ((bt-val (get-bt-val btype))
           (bt-mod (get-bt-mod btype))
           (loc    (next-loc! 'next)))
       (begin
         (cond
           ((and (static? bt-val) (eliminable? bt-mod))
            (let ((val (spec-init-val btype)))
              (update-store! loc val)))
           ((and (dynamic? bt-val) (residual? bt-mod))
            (let* ((res-vdecl (spec-cdecl loc btype residual-vdecls))
                   (tmp       (set! residual-vdecls res-vdecl)))
              (update-store! loc (make-varref loc))))
           ((and (static? bt-val) (residual? bt-mod))
            (let* ((res-vdecl (spec-cdecl loc btype residual-vdecls))
                   (tmp       (set! residual-vdecls res-vdecl))
                   (val       (spec-init-val btype)))
              (update-store! loc (cons (make-varref loc) val)))))
         (make-loc-val loc))))
    ((and (dynamic? btval) (residual? btmod))
     (let ((type (spec-type btype)))
       (make-new type)))
    ((and (static? btval) (residual? btmod))
     (let ((bt-val (get-bt-val btype))
           (bt-mod (get-bt-mod btype))
           (loc (next-loc! 'next))
           (type (spec-type btype)))
       (begin
         (cond
           ((and (dynamic? bt-val) (residual? bt-mod))
            (update-store! loc (make-varref loc)))
           ((and (static? bt-val) (residual? bt-mod))
            (let ((val (spec-init-val btype)))
              (update-store! loc (cons (make-varref loc) val)))))
         (cons (make-new type) (make-loc-val loc)))))))
(bt-nil (btval btmod btype)
  (cond
```

```
          ((and (static? btval) (eliminable? btmod))
           (init-loc))
          ((and (dynamic? btval) (residual? btmod))
           (let ((type (spec-type btype)))
             (make-nil type)))
          ((and (static? btval) (residual? btmod))
           (let ((type (spec-type btype)))
             (cons (make-nil type) (init-loc))))))
      (bt-arrayref (btval btmod vexp exp)
        (let ((val     (spec-rexp vexp))
              (ind-val (spec-rexp exp)))
          (cond
            ((and (static? btval) (eliminable? btmod))
             (let ((ind  (nat-val->val ind-val)))
               (if (and (>= ind (array-val->lower val))
                        (<= ind (array-val->upper val)))
                   (let ((map (array-val->map val)))
                     (lookup-store (lookup-table ind map)))
                   (error "spec : array index out of range :" ind))))
            ((and (dynamic? btval) (residual? btmod))
             (make-arrayref val ind-val))
            ((and (static? btval) (residual? btmod))
             (let ((ind (nat-val->val ind-val)))
               (if (and (>= ind (array-val->lower val))
                        (<= ind (array-val->upper val)))
                   (let ((map (array-val->map (cdr val))))
                     (cons (make-arrayref (car val) ind)
                           (lookup-table ind map)))
                   (error "spec : array index out of range :" ind)))))))
      (bt-recmem (btval btmod exp var)
        (let ((val (spec-rexp exp)))
          (cond
            ((and (static? btval) (eliminable? btmod))
             (lookup-store (lookup-table var (rec-val->env val))))
            ((and (dynamic? btval) (residual? btmod))
             (make-recmem val var))
            ((and (static? btval) (residual? btmod))
             (let* ((rec-val (lookup-store
                              (lookup-table var (rec-val->env (cdr val))))))
               (cons (make-recmem (car val) var) (cdr rec-val)))))))
      (else (error "spec : unknown rexp :" rexp)))))

;;-----------------------------------------------------------------
;; operation specialization
;;-----------------------------------------------------------------

(define spec-op
  (lambda (op val1 val2)
    (make-nat-val
      (case op
        ((+) (+ (nat-val->val val1) (nat-val->val val2)))
        ((-) (- (nat-val->val val1) (nat-val->val val2)))
        ((*) (* (nat-val->val val1) (nat-val->val val2)))
        ((/) (/ (nat-val->val val1) (nat-val->val val2)))
        ((>) (if (> (nat-val->val val1) (nat-val->val val2)) 1 0))
        ((gte) (if (>= (nat-val->val val1) (nat-val->val val2)) 1 0))
        ((<) (if (< (nat-val->val val1) (nat-val->val val2)) 1 0))
        ((lte) (if (<= (nat-val->val val1) (nat-val->val val2)) 1 0))
        ((=)
         (letrec ((loop (lambda (op env1 env2)
                          (if (null? env1)
                              1
                              (let ((ret (eval-op op
                                                  (lookup-store (cdar env1))
                                                  (lookup-store (cdar env2)))))
                                (if (true-val? ret)
                                    (loop op (cdr env1) (cdr env2))
```

```
                                    0))))))
          (cond
            ((and (nat-val? val1) (nat-val? val2))
             (if (equal? (nat-val->val val1) (nat-val->val val2)) 1 0))
            ((and (loc-val? val1) (loc-val? val2))
             (if (equal? (loc-val->loc val1) (loc-val->loc val2)) 1 0))
            ((and (array-val? val1) (array-val? val2))
             (loop op (array-val->map val1) (array-val->map val2)))
            ((and (rec-val? val1) (rec-val? val2))
             (loop op (rec-val->env val1) (rec-val->env val2))))))
        (else (error "spec : unknown op :" op))))))

;;-------------------------------------------------------------
;; jump specialization
;;-------------------------------------------------------------

(define spec-jump
  (lambda (jump blocks)
    (variant-case jump
      (bt-goto (btval btmod label)
        (cons label (make-goto label)))
      (bt-if (btval btmod exp then-label else-label)
        (cond
          ((and (static? btval) (eliminable? btmod))
             (if (true-val? (spec-rexp exp))
                 (cons then-label (make-goto then-label))
                 (cons else-label (make-goto else-label))))
          ((and (dynamic? btval) (residual? btmod))
           (let* ((exp-code (spec-rexp exp)))
             (cons (cons then-label else-label)
                   (make-if exp-code then-label else-label))))))
      (bt-return (btval btmod exp)
        (cons (make-halt) (make-return (spec-rexp exp))))
      (else (error "spec : not a jump :" jump)))))

;;-------------------------------------------------------------
;; check if expression is true
;; val -> #t | #f
;;-------------------------------------------------------------

(define true-val?
  (lambda (val)
    (not (equal? 0 (nat-val->val val)))))

;;-------------------------------------------------------------
;; run the program (main)
;;-------------------------------------------------------------

(define spec
  (lambda (prog args)
    (let ((parsed-prog (bt-parse prog)))
      (begin
        (bt-check parsed-prog)
        (spec-prog parsed-prog args)))))
```

B.5 Unparser

```
;;=============================================================
;; bt-unparser.scm
;; unit to unparse the residual code to original source code
;;=============================================================

(load "bt-syntax.scm")

;;=============================================================
```

```
;; bt-unparser
;; converts the flow-chart language from records to lists
;; without annotation
;;=================================================================
;; program unparsing
;; prog (record) -> prog (list)
;;-----------------------------------------------------------------

(define bt-unparse-prog
  (lambda (prog)
    (if (program? prog)
        (list (bt-unparse-decls (program->decls prog))
              (program->params prog)
              (map bt-unparse-block (program->blocks prog)))
        (error "bt-unparse : not a parsed program :" prog))))

;;-----------------------------------------------------------------
;; declarations unparsing
;; decls (record) -> decls (list)
;;-----------------------------------------------------------------

(define bt-unparse-decls
  (lambda (decls)
    (if (decls? decls)
        (list (map bt-unparse-tdef (decls->tdefs decls))
              (map bt-unparse-vdecl (decls->vdecls decls)))
        (error "bt-unparse : not a parsed declarations :" decls))))

;;-----------------------------------------------------------------
;; type definition unparsing
;; tdef (record) -> tdef (list)
;;-----------------------------------------------------------------

(define bt-unparse-tdef
  (lambda (tdef)
    (if (tdef? tdef)
        (let ((name (tdef->name tdef))
              (vdecls (map bt-unparse-vdecl (tdef->vdecls tdef))))
          (list 'record name vdecls))
        (error "bt-unparse : not a parsed type definition :" tdef))))

;;-----------------------------------------------------------------
;; variable declaration unparsing
;; vdecl (record) -> vdecl (list)
;;-----------------------------------------------------------------

(define bt-unparse-vdecl
  (lambda (vdecl)
    (if (vdecl? vdecl)
        (let ((name (vdecl->name vdecl))
              (type (bt-unparse-type (vdecl->type vdecl))))
          (list name ': type))
        (error "bt-unparse : not a parsed variable declaration :" vdecl))))

;;-----------------------------------------------------------------
;; type unparsing
;; type (record) -> type (list)
;;-----------------------------------------------------------------

(define bt-unparse-type
  (lambda (btype)
    (variant-case btype
      (nat-type () 'nat)
      (access-type (type) (list 'access (bt-unparse-type type)))
      (array-type (lower upper type)
        (list 'array lower upper (bt-unparse-type type)))
      (rec-type (name) (list 'record name))
```

```
      (else (error "bt-unparse : not a parsed type :" btype)))))

;;-----------------------------------------------------------------
;; block unparsing
;; block (record) -> block (list)
;;-----------------------------------------------------------------

(define bt-unparse-block
  (lambda (block)
    (if (block? block)
        (let ((label   (block->label block))
              (assigns (block->assigns block))
              (jump    (block->jump block)))
          (list label (map bt-unparse-assign assigns) (bt-unparse-jump jump)))
        (error "bt-unparse : not a parsed block :" block))))

;;-----------------------------------------------------------------
;; assignment unparsing
;; assign (record) -> assign (list)
;;-----------------------------------------------------------------

(define bt-unparse-assign
  (lambda (assign)
    (variant-case assign
      (assign (lexp exp)
              (list (bt-unparse-exp lexp) ':= (bt-unparse-exp exp)))
      (else (error "bt-unparse : not a parsed assignment :" assign)))))

;;-----------------------------------------------------------------
;; expression unparsing
;; exp (record) -> exp (list)
;;-----------------------------------------------------------------

(define bt-unparse-exp
  (lambda (exp)
    (variant-case exp
      (nat (datum) datum)
      (varref (var) var)
      (app (op exps) (cons op (map bt-unparse-exp exps)))
      (deref (exp) (list 'deref (bt-unparse-exp exp)))
      (new (type) (list 'new (bt-unparse-type type)))
      (nil (type) (list 'nil (bt-unparse-type type)))
      (arrayref (vexp exp)
        (list '[] (bt-unparse-exp vexp) (bt-unparse-exp exp)))
      (recmem (exp var)
        (list 'recmem (bt-unparse-exp exp) var))
      (else (error "bt-unparse : not a parsed expression :" exp)))))

;;-----------------------------------------------------------------
;; jump unparsing
;; jump (record) -> jump (list)
;;-----------------------------------------------------------------

(define bt-unparse-jump
  (lambda (jump)
    (variant-case jump
      (goto (label) (list 'goto label))
      (if (exp then-label else-label)
          (list 'if (bt-unparse-exp exp) then-label else-label))
      (return (exp) (list 'return (bt-unparse-exp exp)))
      (else (error "bt-unparse : not a parsed jump :" jump)))))

;;-----------------------------------------------------------------
;; bt-unparse-prog alias (main)
;;-----------------------------------------------------------------

(define unparse bt-unparse-prog)
```

B.6 Module for tables manipulation

```
;;=====================================================================
;; bt-table.scm
;; functions for manipulating tables
;; with side-effect for specializer
;;=====================================================================

(load "table.scm")

;;=====================================================================
;; global table btype-head
;;=====================================================================
;; create new empty btype-head
;;---------------------------------------------------------------------

(define btype-head empty-table)

;;---------------------------------------------------------------------
;; reset btype-head
;;---------------------------------------------------------------------

(define reset-btype-head!
  (lambda ()
    (set! btype-head empty-table)))

;;---------------------------------------------------------------------
;; set btype-head
;;---------------------------------------------------------------------

(define set-btype-head!
  (lambda (new-table)
    (set! btype-head new-table)))

;;---------------------------------------------------------------------
;; lookup bt for var in global btype-head
;; var -> bt
;;---------------------------------------------------------------------

(define lookup-btype-head
  (lambda (var)
    (lookup-table var btype-head)))

;;=====================================================================
;; global table btype-table
;;=====================================================================
;; create new empty global btype-table
;;---------------------------------------------------------------------

(define btype-table empty-table)

;;---------------------------------------------------------------------
;; reset btype-table
;;---------------------------------------------------------------------

(define reset-btype-table!
  (lambda ()
    (set! btype-table empty-table)))

;;---------------------------------------------------------------------
;; set btype-table
;;---------------------------------------------------------------------

(define set-btype-table!
  (lambda (new-table)
    (set! btype-table new-table)))
```

```
;;-----------------------------------------------------------------
;; lookup decls for var in global btype-table
;; var -> decls
;;-----------------------------------------------------------------

(define lookup-btype-table
  (lambda (var)
    (lookup-table var btype-table)))

;;-----------------------------------------------------------------
;; lookup var existence in global btype-table
;; var -> #t | #f
;;-----------------------------------------------------------------
(define exist-btype-table?
  (lambda (var)
    (exist-table? var btype-table)))

;;-----------------------------------------------------------------
;; update var with new decls in global btype-table
;;-----------------------------------------------------------------

(define update-btype-table!
  (lambda (var decls)
    (set! btype-table (update-table var decls btype-table))))

;;=================================================================
;; global table btype-env
;;=================================================================
;; create new empty global btype-env
;;-----------------------------------------------------------------

(define btype-env empty-table)

;;-----------------------------------------------------------------
;; reset btype-env
;;-----------------------------------------------------------------

(define reset-btype-env!
  (lambda ()
    (set! btype-env empty-table)))

;;-----------------------------------------------------------------
;; set btype-env
;;-----------------------------------------------------------------

(define set-btype-env!
  (lambda (new-table)
    (set! btype-env new-table)))

;;-----------------------------------------------------------------
;; lookup btype for var in global btype-env
;; var -> btype
;;-----------------------------------------------------------------

(define lookup-btype-env
  (lambda (var)
    (lookup-table var btype-env)))

;;-----------------------------------------------------------------
;; update var with new btype in global btype-env
;;-----------------------------------------------------------------

(define update-btype-env!
  (lambda (var btype)
    (set! btype-env (update-table var btype btype-env))))

;;=================================================================
```

```
;; global table environment
;;=================================================================
;; create new empty global environment
;;-----------------------------------------------------------------

(define env empty-table)

;;-----------------------------------------------------------------
;; reset env
;;-----------------------------------------------------------------

(define reset-env!
  (lambda ()
    (set! env empty-table)))

;;-----------------------------------------------------------------
;; set env
;;-----------------------------------------------------------------

(define set-env!
  (lambda (new-table)
    (set! env new-table)))

;;-----------------------------------------------------------------
;; location function
;; - get      : get current location
;; - next     : get current location and update location
;; - reset    : reset location to initial value (0)
;; - (number) : set location to the number
;;-----------------------------------------------------------------

(define next-loc!
  (let ((loc 1))
    (lambda (comm)
      (cond
        ((symbol? comm)
         (case comm
           ((get)
            (string->symbol (string-append "loc-"
                                           (number->string loc))))
           ((next)
            (let ((loc-current loc)
                  (tmp (set! loc (+ loc 1))))
              (string->symbol (string-append "loc-"
                                             (number->string loc-current)))))
           ((reset)
            (let ((loc-current loc)
                  (tmp (set! loc 1)))
              loc-current))))
        ((number? comm)
         (set! loc comm))))))

;;-----------------------------------------------------------------
;; lookup location fo var in global environment
;; var -> loc
;;-----------------------------------------------------------------

(define lookup-env
  (lambda (var)
    (lookup-table var env)))

;;-----------------------------------------------------------------
;; update var with new location in global environment
;;-----------------------------------------------------------------

(define update-env!
  (lambda (var loc)
```

```
    (set! env (update-table var loc env))))

;;=====================================================================
;; global table store
;;=====================================================================
;; create new empty global store
;;---------------------------------------------------------------------

(define store empty-table)

;;---------------------------------------------------------------------
;; reset store
;;---------------------------------------------------------------------

(define reset-store!
  (lambda ()
    (set! store empty-table)))

;;---------------------------------------------------------------------
;; set store
;;---------------------------------------------------------------------

(define set-store!
  (lambda (new-table)
    (set! store new-table)))

;;---------------------------------------------------------------------
;; lookup value for var in global store
;; loc -> value
;;---------------------------------------------------------------------

(define lookup-store
  (lambda (loc)
    (lookup-table loc store)))

;;---------------------------------------------------------------------
;; update loc with new value in global store
;;---------------------------------------------------------------------

(define update-store!
  (lambda (loc val)
    (set! store (update-table loc val store))))

;;=====================================================================
;; table.scm
;; functions for manipulating tables and lists
;;=====================================================================

(define empty-list '())

(define empty-table '())

;;=====================================================================
;; list manipulation with no side-effect
;;=====================================================================
;; check if variable exist in list
;; var x list -> #t | #f
;;---------------------------------------------------------------------

(define exist-list?
  (lambda (var list)
    (if (null? list)
        #f
        (if (equal? (car list) var)
            #t
            (exist-list? var (cdr list))))))
```

```
;;=================================================================
;; table manipulation with no side-effect
;;=================================================================
;; update table variable with new value if variable exist or
;; new cell for variable otherwise
;; var x val x table -> table
;;-----------------------------------------------------------------

(define update-table
  (lambda (var val table)
    (if (null? table)
        (list (cons var val))
        (if (eq? (caar table) var)
            (cons (cons var val) (cdr table))
            (cons (car table) (update-table var val (cdr table)))))))

;;-----------------------------------------------------------------
;; lookup variable's value in table
;; var x table -> value (if exist or error otherwise)
;;-----------------------------------------------------------------

(define lookup-table
  (lambda (var table)
    (if (null? table)
        (error "lookup-table : variable not found :" var)
        (if (equal? (caar table) var)
            (cdar table)
            (lookup-table var (cdr table))))))

;;-----------------------------------------------------------------
;; check if variable exist in table
;; var x table -> #t | #f
;;-----------------------------------------------------------------

(define exist-table?
  (lambda (var table)
    (if (null? table)
        #f
        (if (equal? (caar table) var)
            #t
            (exist-table? var (cdr table))))))

;;-----------------------------------------------------------------
;; add table with list of variable and value
;; vars x vals x table -> table
;;-----------------------------------------------------------------

(define add-table
  (lambda (vars vals table)
    (if (null? vars)
        table
        (cons (cons (car vars) (car vals))
              (add-table (cdr vars) (cdr vals) table)))))

;;-----------------------------------------------------------------
;; domain table
;; table -> list
;; (domain x image) -> domain
;;-----------------------------------------------------------------

(define domain-table
  (lambda (table)
    (map car table)))

;;-----------------------------------------------------------------
;; image table
;; table -> list
```

```
;; (domain x image) -> image
;;-------------------------------------------------------------------

(define image-table
  (lambda (table)
    (map cdr table)))
```

B.7 Miscellaneous functions

```
;;===================================================================
;; bt-lib.scm
;;===================================================================
;; all true
;; - test to see that every value in a list is true
;;   necessary because "apply" doesn't work with "and"
;;-------------------------------------------------------------------

(define all-true?
  (lambda (l)
    (foldr l #t (lambda (arg1 arg2) (and arg1 arg2)))))

;;-------------------------------------------------------------------
;; help in function all-true
;;-------------------------------------------------------------------

(define foldr
  (lambda (list base fun)
    (if (null? list)
        base
        (fun (car list) (foldr (cdr list) base fun)))))
```

VITA

Muhammad Wahyu Nanda

Candidate for the Degree of

Master of Science

Thesis: PARTIAL EVALUATION FOR CONCURRENT SOFTWARE FINITE STATE VERIFICATION

Major Field: Computer Science

Biographical:

Personal Data: Born in Pekanbaru, Riau, Indonesia, on Aril 27, 1967, the third son of Rumzi Bey Rasjad and Hajatun Nismah.

Education: Graduated from SMA Cendana Rumbai in Rumbai, Pekanbaru, Riau, Indonesia in May 1985; received Sarjana Teknik degree in Engineering Informatics from Institut Teknologi Bandung, Bandung, Jawa Barat, Indonesia in October 1994. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December 1999.

Experience: Employed as an animator by PT. Mutiara Multimedia, Jakarta, Indonesia, 1990 to 1991; employed as a computer graphics specialist by PT. Spektragrafik Intermedia, Jakarta, Indonesia, 1994 to 1995; employed as a computer graphics specialist by PT. Pyramid Image, Jakarta, Indonesia, 1995 to 1996; employed by Oklahoma State University, Department of Computer Science as graduate research assistant during Summer 1997 and Summer 1998; employed by Oklahoma State University, Department of Computer Science as graduate teaching assistant, 1997 to 1998; employed by Sabre as programmer/analyst, since 1998.