Binding-time analysis for a JVM core language

Peter Bertelsen (Lc1761)

`http://www.dina.kvl.dk/~pmb`

Department of Mathematics and Physics
Royal Veterinary and Agricultural University
Copenhagen, Denmark

April 4, 1999

Contents

1 Introduction

This report describes a *binding-time analysis* for a simple bytecode 'core language'. The core language, which is presented in section 2, is an abstraction over a small subset of the Java Virtual Machine (JVM) instruction set [13]. The JVM is an abstract machine for executing Java bytecode programs[1].

The binding-time analysis presented here was developed during a special course on constraint-based static analyses under the supervision of assistant professor Jesper Jørgensen and associate professor Peter Sestoft, Department of Mathematics and Physics, Royal and Veterinary University of Copenhagen, Denmark.

This work is funded by the DART research project, which in turn is funded by the Danish Research Councils.

1.1 Binding-time analysis and off-line partial evaluation

A *partial evaluator* for a programming language takes a program and some of its parameters and generates a specialized program. The resulting *residual* program behaves just the same as the original program, except that it does not expect the parameters that were used by the partial evaluator. Moreover, the specialized program may execute significantly faster than the original program.

An *off-line partial evaluator* is guided by the *binding-time* annotations made by a *binding-time analysis* [12]. The purpose of the binding-time analysis is thus to determine which parts of a program may be evaluated in case only some of the program parameters are known.

In other words, a binding-time analysis calculates the *binding-time* for each variable (and basic operation) in a program. The input to the binding-time analysis is a program and a classification of its parameters: which parameters will be known and which parameters will be unknown at specialization-time. The known values are also referred to as *static* values and unknown values are referred to as *dynamic* values.

The binding-time analysis propagates this information throughout the program and determines the binding-time of each variable and of each basic operation to be either static or dynamic[2]. Informally, all operations depending only on static parameters should be annotated as static, whereas all operations depending (directly or indirectly) on dynamic parameters should be annotated as dynamic.

Given the values of the static program parameters, an off-line partial evaluator specializes the program as indicated by the binding-time annotations determined by the binding-time analysis, keeping track of static and dynamic values as it goes. The result of this is a specialized program in which all static parts have been evaluated, using the values of the static parameters.

1.2 Our approach

The binding-time analysis described in this report is *constraint-based* [11, 9]: given a program written in our core language and a binding-time typing of the program's parameters, we generate a set of constraints that must hold for any valid binding-time annotation of that program. For example, we might generate a constraint asserting that an instruction loading the value of a specific variable must be annotated as dynamic in case that variable turns out to be dynamic.

By means of the constraint solver developed by Birkedal and Welinder [8] we find an optimal solution to the generated set of constraints. An optimial solution is one that is as static as possible, i.e., allowing for as much specialization as possible of the resulting annotated program. Based on this solution we derive an annotated version of the original program, including special instructions for handling binding-time coercions (also known as *lifting*).

[1] Throughout this report, we sometimes refer to programs written in the instruction set of the JVM as "Java bytecode" programs.

[2] Our analysis also handles *partially static arrays* and thus may annotate certain instructions as being partially static. This is described in more detail in section 3.

The steps involved in our binding-time analysis may be summarized as follows:

- constraint generation (see section 3.5)
- constraint solving (see section 3.6)
- translation to annotated code, with insertion of lift instructions (see section 3.7).

The result of our analysis is thus a program written in the annotated core language, which corresponds to the core language, except that most instructions are annotated as static or dynamic. Furthermore, the annotated core language has special instructions for lifting stack operands and local variables to a more dynamic context.

1.3 Why focus on Java bytecode?

We are interested in binding-time analysis and program specialization at the level of Java bytecode for a number of reasons, including:

- Java bytecode (i.e., Java class files) is a general-purpose, platform independent software distribution format.
- The semantics of Java bytecode is fairly well-defined [13, 4].
- Java bytecode is more flexible than Java source code; specifically, some valid Java bytecode programs cannot be expressed in Java source code.
- Other programming languages than Java may be compiled to Java bytecode, e.g. Standard ML [14, 5].
- Development of Java software based on generic components may benefit from automatic specialization.
- Optimization techniques usually deployed in connection with partial evaluation may be carried over to optimization of Java bytecode programs in general.
- We already have expertise in Java bytecode and its semantics.

Unfortunately, optimization at the level of Java bytecode has the important disadvantage that some optimizations are impossible in a bytecode-to-bytecode translation. For example, array bounds checking can in general not be optimized away since the JVM *must* check array bounds at runtime[3].

However, optimization of Java bytecode programs, e.g. by program specialization, may still be worthwhile. For example, layers of interpretation in a Java bytecode program may be specialized away to yield a significantly faster specialized program, loops in the program may be unrolled, etc. (as is the case for other programming languages).

1.4 Report outline

The remainder of this report is structured as follows:

- In section 2 we present our simple core language and define the kinds of values that the instructions of the core language operate on. We briefly explain the semantics of each instruction and describe how the core language corresponds to the actual instruction set of the JVM. Additionally, we present some assumptions concerning well-typedness that our binding-time analysis relies upon.

[3] In order to minimize the runtime overhead of array bounds checking, arrays may be 'unfolded' (where possible) by storing the components of a 'known' array in local variables. This is described in more detail in section 5.1.

- In section 3, which is the central part of this report, we present our binding-time analysis. First, we define the domain of binding-time types that are used in the analysis and present our annotated core language.

  We then describe the binding-time constraints that are used in our analysis and present a set of inference rules that must hold for any well-annotated program written in the annotated core language. Based on the inference rules, we define an algorithm for generating binding-time constraints and explain how the constraint solver developed by Birkedal and Welinder [8] can be used to solve the generated constraints. Given a solution to the generated constraints, we describe how the original core language program may be translated into a well-annotated core language program.

  Finally, we outline our implementation of the presented binding-time analysis, which is based on Birkedal and Welinder's implementation of their constraint solver, and explain how the 'user interface' of our implementation works.

- In section 4 we demonstrate our implementation of the binding-time analysis through a few examples. For each sample program, we show the original core language program and the annotated program that is obtained from the binding-time analysis for a specific classification of the input variables.

- In section 5 we outline a possible approach to specialization of programs written in the annotated core language of section 3. At the time of writing, a specializer had not yet been implemented. Instead, we describe the flavour of partial evaluation that our binding-time analysis is meant to support.

- Section 6 concludes, describes related work and lists plans and ideas for further work towards partial evaluation of Java bytecode.

- Appendix A summarizes the inference rules defined in section 3.4.

2 The JVM core language

The abstract instruction set *Instr* considered in this report comprises the following instructions:

```
arraylength
arrayload
arraystore
binop ⊙
const k
dup
goto l
if r l
label l
load j
newarray d
pop
return
store j
swap
```

where $\odot \in \{\mathsf{add}, \mathsf{sub}, \mathsf{mul}, \mathsf{div}, \mathsf{rem}\}$, $k \in \mathit{Int} \cup \{\mathtt{null}\}$, $l \in \mathit{Label}$, $r \in \{\mathsf{eq}, \mathsf{ge}, \mathsf{gt}, \mathsf{le}, \mathsf{lt}, \mathsf{ne}\}$, $j \in \mathit{Index}$ and $d \in \mathcal{N}$.

Int is the set of integers, `null` is a special reference value (see section 2.1), *Label* is the domain of abstract labels, *Index* is the ordered domain of abstract local variable indices and $\mathcal{N}$ is the set of natural numbers. The least local variable index is denoted j_{min}:

$$\forall j \in \mathit{Index} \,.\, j_{min} \sqsubseteq j$$

A core language program is a sequence of *Instr* instructions:

$$\mathit{Prog} = \mathit{Instr}^*$$

We use the notation $c :: cs$ to denote the instruction sequence consisting of the instruction c followed by the instruction sequence cs. The empty instruction sequence is denoted *nil*.

2.1 Values

The instructions in the core language operate only on values of type *Val*:

$$\mathit{Val} = \mathit{Int} \cup \mathit{Ref} \cup \{\mathtt{null}\}$$

where *Ref* is the type of array references (see below) and `null` is a special 'reference' value that never refers to anything.

The instructions in the abstract instruction set manipulate an operand stack, a local variable environment[4] and a heap:

$$\begin{aligned} \mathit{Stack} &= \mathit{Val}^* \\ \mathit{Locals} &= \mathit{Index} \xrightarrow{\text{fin}} \mathit{Val} \\ \mathit{Heap} &= \mathit{Ref} \xrightarrow{\text{fin}} \mathit{Val}^* \end{aligned}$$

The operand stack, which is of type *Stack*, is a sequence of operands of type *Val*. Most instructions manipulate only the topmost stack operand(s). Initially, when a core language program starts executing, the operand stack is empty.

[4]The variables supported by the core language correspond to the local variables of a Java bytecode method. This is why we refer to the variables of the core language as "local variables" although the core language does not support any non-local variables.

The local variable environment, which is of type *Locals*, maps a local variable index j to the value of local variable j. The parameters to a core language program are passed in local variables. Initially, only those local variables that correspond to program parameters have a value; any other local variable is undefined until it has been stored into.

The heap, which is of type *Heap*, maps an array reference r to an array (vector) of components. The components of an array are numbered from 0 through $n-1$, where n is the length of the array. The special reference value `null` is never in the domain of the heap.

A one-dimensional array is represented as an object of type Val^* in the heap; each component of a one-dimensional array object is an integer value. A multi-dimensional array is represented as a 'tree' of array objects in the heap: the components of the 'root' array are references to arrays of the next dimension (or `null`).

More generally, the components of an array object of dimension d, where $d > 1$, are references to array objects of dimension $d-1$ (or `null`). Note that the tree of array objects constituting a multi-dimensional array need not be balanced and need not include any array objects at 'lower' dimensions than the root array object (since any of its components may be `null`).

2.2 Semantics of instructions

Informally, the semantics of each abstract instruction is as follows:

- `arraylength` removes the topmost stack operand, which must be a reference to an array, and pushes the length of the array onto the stack.

- `arrayload` removes the two topmost stack operands[5], which must be an integer i and an array reference, and pushes the value of component i in the array onto the stack. The index i must be non-negative and less than the length of the array.

- `arraystore` removes the three topmost stack operands, which must be a value v, an integer i and an array reference; the value v is stored into component i in the array. As for `arrayload`, the index i must be non-negative and less than the length of the array.

- `binop` $\odot$ removes the two topmost stack operands v_1 and v_2, which must both be integers, performs the calculation corresponding to the operator $\odot$ and pushes the result onto the stack. The meaning of each operator is as follows:

$\odot$	Result
add	$v_2 + v_1$
sub	$v_2 - v_1$
mul	$v_2 \cdot v_1$
div	$\lfloor v_2/v_1 \rfloor$
rem	$v_2 - v_1 \cdot \lfloor v_2/v_1 \rfloor$

  For $\odot \in \{\mathsf{div}, \mathsf{rem}\}$ the divisor v_1 must be non-zero.

- `const` k pushes the constant k onto the stack.

- `dup` pushes a copy of the topmost stack operand onto the stack.

- `goto` l branches unconditionally to label l.

- `if` r l removes the two topmost stack operands v_1 and v_2 and branches to label l in case the relation r is satisfied. The meaning of each relation is as follows:

[5] The stack operands to each instruction are mentioned in the order they appear on the stack, starting with the top-most stack operand.

Relation	Satisfied for
eq	$v_2 = v_1$
ge	$v_2 \geq v_1$
gt	$v_2 > v_1$
le	$v_2 \leq v_1$
lt	$v_2 < v_1$
ne	$v_2 \neq v_1$

For ge, gt, le and lt, v_1 and v_2 must be integers. For eq and ne, either v_1 and v_2 must be integers or both must be array references (but need not refer to arrays of the same dimensionality).

If the relation r is not satisfied, execution continues with the instruction immediately following the `if` instruction.

- `label` l represents the label l, but otherwise has no effect.
- `load` j pushes the value of local variable j onto the stack. Local variable j must either hold a value of a program parameter or previously have been assigned a value by means of a `store` j instruction (see below).
- `newarray` d removes the topmost stack operand, which must be a non-negative integer n, and pushes a fresh reference to a new array of length n onto the stack. The component type of the new array is determined as follows: if $d = 1$, a one-dimensional array with all components equal to 0 is created; otherwise an array with all components equal to `null` is created[6].
- `pop` discards the topmost stack operand.
- `return` terminates the program and passes the topmost stack operand as the result of the program (any other stack operands are discarded).
- `store` j removes the topmost stack operand and stores it into local variable j.
- `swap` exchanges the two topmost stack operands.

For all instructions but `goto`, `if` and `return`, execution always proceeds with the next instruction when the current instruction has completed. Only `goto` and `return` may appear as the last instruction in a core language program.

Each of the instructions in the abstract instruction set fails if the required number of operands is not available on the operand stack, if the operands are not of the required types or if an operand has an invalid value. When an instruction fails, the semantics of the entire program is undefined.

2.3 Relation to JVM instructions

The abstract instruction set *Instr* is an abstraction[7] over a subset of the JVM instruction set [13].

A program written in the core language corresponds to a single method in the actual JVM instruction set. In other words, the following features of Java bytecode are not supported in the core language:

- classes, fiels and methods
- class instances (objects)
- exception handling

[6]The 'lower' dimensions of a multi-dimensional array must be initialized explicitly by storing into each component of the 'root' array object.

[7]This abstraction is similar to the abstract instruction set defined in the *SML-JVM toolkit* [3].

Apart from this, the relation between the actual JVM instruction set and our core language is fairly straightforward. Some of the instructions in the abstract instruction set correspond directly to the same instructions in the JVM instruction set: `arraylength`, `dup`, `pop` and `swap`.

The remaining abstract instructions correspond to the actual JVM instructions as follows:

Abstract instruction	Corresponding JVM instructions
`arrayload`	`aaload` or `iaload`
`arraystore`	`aastore` or `iastore`
`binop` $\odot$	`iadd`, `isub`, `imul`, `idiv` or `irem`
`const` k	`iconst_<i>`, `bipush` k, `sipush` k, `ldc` i, `ldc_w` i or `aconst_null`
`goto` l	`goto` δ or `goto_w` δ
`if` r l	`if_acmpeq` δ, `if_acmpne` δ, `if_icmpeq` δ, `if_icmpge` δ, `if_icmpgt` δ, `if_icmple` δ, `if_icmplt` δ or `if_icmpne` δ
`load` j	`aload` j' or `iload` j'
`newarray` t	`anewarray` i or `newarray` t'
`return`	`areturn` or `ireturn`
`store` j	`astore` j' or `istore` j'

The argument i to each of the concrete JVM instructions `ldc`, `ldc_w` and `anewarray` is an index into the constant pool of the class file. The constant pool of a Java class file holds (large) program constants and symbolic references to classes, interfaces, fields and methods.

The argument δ to each the concrete JVM instructions `goto`, `goto_w`, `if_acmpeq`, `if_acmpne`, `if_icmpeq`, `if_icmpge`, `if_icmpgt`, `if_icmple`, `if_icmplt`, and `if_icmpne` is an instruction offset (rather than a target label).

The argument j' to each of the concrete JVM instructions `aload`, `iload`, `astore` and `istore` is a non-negative integer corresponding to the abstract local variable index j.

The argument t' to the concrete JVM instruction `newarray` is an integer specifying a simple component type[8].

The abstract instruction `label` l serves as an explicit label marker in the abstract core language; it has no counterpart in the actual JVM instruction set.

Some of the concrete JVM instructions mentioned above will throw a runtime exception in certain situations:

- `aaload`, `aastore`, `iaload` and `iastore` will throw an `ArrayIndexOutOfBoundsException` in case the index operand is not within the bounds of the array being accessed;
- `idiv` and `irem` will throw an `ArithmeticException` in case the divisor is 0; and
- `anewarray` and `newarray` t' will throw a `NegativeArraySizeException` in case the length operand is negative.

Since exceptions are not supported in our core language, the semantics of the corresponding abstract instructions in those special situations has been left undefined (cf. section 2.2).

2.4 Assumptions

In the following sections, we assume for simplicity that the programs being analyzed are valid with respect to the JVM specification [13]: for any core language program, a corresponding sequence of concrete JVM instructions must constitute a valid Java bytecode method and must be accepted by a conforming Java bytecode verifier.

This implies that the following features of Java bytecode carry over to our core language:

- static typing of stack operands and local variables is possible;

[8] In a translation to/from our abstract instruction set, the argument t' to the concrete JVM instruction `newarray` must be 10, meaning `int`, since the only simple type in the abstract core language is *Int*.

- for any instruction in a program, the operand stack must always have the same depth, regardless of which path of control is chosen to reach that instruction;
- the maximum operand stack depth of a program may be determined statically;
- a program uses a fixed set of local variables;
- a local variable cannot be loaded from before a value has been stored into it, unless the variable already holds a method parameter[9].

Additionally, each label l referenced from an `if` r l or `goto` l instruction must be defined by exactly one `label` l instruction in the program. This extra requirement is due to the use of abstract labels as branch targets in the core language (rather than instruction offsets as in the actual JVM instruction set).

We assume for simplicity that a core language program does not contain any unreachable code. This implies that any `goto` l or `return` instruction in the program must either be the last instruction or be followed by a `label` l' instruction, where l' is referenced from a reachable `if` r l' or `goto` l' instruction in the program.

[9]The parameters to a method in the actual JVM instruction set are passed in local variables.

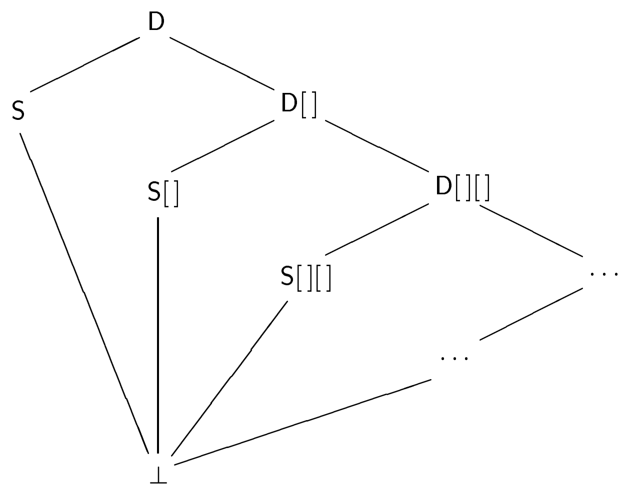

Figure 1: The binding-time domain BT

3 Binding-time analysis

In the following sections we present our binding-time analysis for the core language described in section 2.

We define the domain of binding-time types that are used in the analysis and present a slightly different version of our core language: the *annotated* core language. We describe the binding-time constraints that are used in our analysis and present a set of inference rules that define well-annotatedness for the annotated core language.

From the inference rules we derive a constraint generation algorithm and show how the constraints generated by this algorithm may be solved using the constraint solver developed by Birkedal and Welinder [8]. Based on a solution to the generated constraints, we describe how a core language program may be translated into our annotated core language.

Finally, our implementation of the presented binding-time analysis is outlined, along with a description of its 'user interface'.

3.1 The binding-time domain

The binding-time types used in our binding-time analysis are taken from the domain BT, which is shown in figure 1. This domain forms a complete, infinite lattice in which $\perp$ is the least element and D is the greatest element[10].

The type D is used to indicate that a value or an instruction is *dynamic*. A dynamic value is unknown at specialization time, whereas a dynamic instruction must be residualized by the specializer (i.e., will appear in the residual program).

The type S is used to indicate that a (simple) value or an instruction is *static*. A static value is an integer known at specialization time, whereas a static instruction is to be evaluated by the specializer.

A type of the form $b[\,]$, where $b \in BT$ and $b \neq \perp$, is used to indicate that a value or an instruction is *partially static*. A value of binding-time type $b[\,]$ is a reference to an array whose components have binding-time type b; the length and position of the array will be known at specialization time[11]. An instruction of binding-time type $b[\,]$ may be (partially) evaluated by the specializer; this is described in more detail in section 5.2.

[10] Although the domain BT is infinite, we use only a finite subset of it for any given program, corresponding to the data types actually used in the program.

[11] All components of a partially static array are considered to have the same binding-time type.

3.2 Annotated core language

The result of the binding-time analysis presented here is a program written in the annotated core language. The instruction set Instr_{BT} of the annotated core language, which is quite similar to the core language defined in section 2, comprises the following instructions:

$\texttt{arraylength}_b$
$\texttt{arrayload}_b$
$\texttt{arraystore}_b$
$\texttt{binop}_b\ \odot$
$\texttt{const}\ k$
$\texttt{dup}_b$
$\texttt{goto}\ l$
$\texttt{if}_b\ r\ l$
$\texttt{label}\ l$
$\texttt{load}_b\ j$
$\texttt{newarray}_b\ d$
$\texttt{pop}_b$
$\texttt{return}$
$\texttt{store}_b\ j$
$\texttt{swap}_b$
$\texttt{liftTop}$
$\texttt{lift2nd}$
$\texttt{lift}\ j$

where $b \in BT$.

Most of the instructions in Instr_{BT} are annotated with a binding-time type b. As indicated in the previous section, an instruction annotated with D must be residualized, whereas an instruction annotated with b, where $b \neq \mathsf{D}$, should be (partially) evaluated by the specializer.

The instructions `const` k, `goto` l, `label` l and `return` of the annotated core language are not annotated with a binding-time type. The intention is that `const` k should always be evaluated by the specializer, whereas `goto` l, `label` l and `return` should always be residualized[12].

The instruction set Instr_{BT} includes special instructions for explicitly *lifting* static values to a more dynamic context: `liftTop` lifts the top-most stack operand, `lift2nd` lifts the second top-most stack operand and `lift` j lifts the value of local variable j, where $j \in \mathit{Index}$.

The intention is that a lift instruction should cause a known value (a constant) to be considered unknown at specialization time; instead of using that value in the specialization process, an instruction for creating the value should be inserted in the residual program. This is described in more detail in section 5.2.

Note that the instruction set of the annotated core language includes instructions for lifting only the two topmost stack operands. It would generally not be possible to generate residual code for lifting an operand at a 'deeper' level in the stack since the actual JVM instruction set does not include instructions for inserting an operand 'below' the two topmost operands in the stack.

Lifting of local variables is more flexible: we may lift a local variable j at any point in the code. This does not have any net effect on the residual operand stack.

3.3 Notation

In this section we define the constraint operators and other special notation used in the following sections. The constraint operators defined here are similar to those defined by Birkedal and Welinder [8] and may be translated into their constraint system (see section 3.6).

We use b, b', b_i and b'_i, where $i = 1, 2, \ldots$, to describe concrete terms from the domain BT of binding-time types.

[12]Superfluous `goto` instructions in the residual program may be eliminated by path compression (cf. section 5.2).

A constraint of the form $b_1 \lhd\rhd b_2$ asserts that b_2 must be D in case b_1 is D *and vice versa*[13]:

$$b_1 \lhd\rhd b_2 \;\equiv\; b_1 = \mathsf{D} \Leftrightarrow b_2 = \mathsf{D}$$

A constraint of the form $(b_1, b_2) \lhd\rhd b_3$ asserts that b_3 must be D in case both b_1 and b_2 are D *and vice versa*:

$$(b_1, b_2) \lhd\rhd b_3 \;\equiv\; b_1 = b_2 = \mathsf{D} \Leftrightarrow b_3 = \mathsf{D}$$

This kind of constraint is used in connection with the `swap` instruction (see rule 15 on page 16).

A constraint of the form $b_1[\,] \leq b_2$ asserts that either b_2 is the binding-time type of an array whose components have binding time type b_1 or else b_1 and b_2 must both be D:

$$b_1[\,] \leq b_2 \;\equiv\; b_1[\,] = b_2 \vee b_1 = b_2 = \mathsf{D}$$

Constraints of this kind are used in connection with the array manipulation instructions.

A constraint of the form $b_1 \rightsquigarrow b_2$ asserts that a value of binding-time type b_1 may be *lifted* to binding-time type b_2:

$$b_1 \rightsquigarrow b_2 \;\equiv\; (b_1 = \mathsf{S} \wedge b_2 = \mathsf{D}) \vee b_1 = b_2$$

Note that only simple values, i.e., those having binding-time type S, may be lifted to a more dynamic context. Array references cannot be lifted[14]: a lift constraint involving array references can only be satisfied by equality or by values having binding-time type D.

In the following section, we use σ (with various superscripts and/or subscripts) to describe a *binding-time stack*:

$$\sigma : \mathit{Stack}_{BT}$$

where

$$\mathit{Stack}_{BT} = BT^*$$

A binding-time stack is a sequence of binding-time types that represents the binding-time types of the corresponding stack operands at a particular point in a core language program.

Similarly, we use ρ (with various superscripts and/or subscripts) to describe a *local variable binding-time environment*, representing the binding-time types of the local variables at a particular point in a core language program:

$$\rho : \mathit{Locals}_{BT}$$

where

$$\mathit{Locals}_{BT} = \mathit{Index} \overset{\text{fin}}{\to} BT$$

A local variable binding-time environment maps each local variable index j to the binding-time type of that variable.

We define the lift constraint operator for binding-time stacks as follows:

$$\sigma_1 \rightsquigarrow \sigma_2 \;\equiv\; \begin{cases} b_1 \rightsquigarrow b_2 \wedge b_1' \rightsquigarrow b_2' \wedge \sigma_1' = \sigma_2' & \text{if } \sigma_1 = b_1 :: b_1' :: \sigma_1' \wedge \sigma_2 = b_2 :: b_2' :: \sigma_2' \\ b_1 \rightsquigarrow b_2 & \text{if } \sigma_1 = b_1 :: \mathit{nil} \wedge \sigma_2 = b_2 :: \mathit{nil} \\ \mathit{true} & \text{if } \sigma_1 = \sigma_2 = \mathit{nil} \\ \mathit{false} & \text{otherwise} \end{cases}$$

The constraint $\sigma_1 \rightsquigarrow \sigma_2$ states that it must be possible to lift the two top-most operands in the stack represented by σ_1 'point-wise' to the two top-most stack operands represented σ_2 and that the remainders of the two stacks (if any) must have the same binding-time types (point-wise). Note that the definition of $\sigma_1 \rightsquigarrow \sigma_2$ captures the fact that we can lift only the two top-most stack operands (cf. section 3.2).

[13]The constraint $b_1 \lhd\rhd b_2$ corresponds to the combination of $b_1 \rhd b_2$ and $b_2 \rhd b_1$, where $\rhd$ is the constraint operator used in other literature on partial evaluation, e.g. [11, 9, 8]. See also section 3.6.

[14]If an array reference could (hypothetically) be lifted, e.g. from binding-time type $\mathsf{S}[\,]$ to $\mathsf{D}[\,]$, problems with duplication of the array in the residual program might arise. To avoid this, we disallow lifting of array references to a more dynamic context.

We define the lift constraint operator for local variable binding-time environments as follows[15]:

$$\rho_1 \leadsto \rho_2 \;\equiv\; dom(\rho_1) \supseteq dom(\rho_2) \wedge \forall j \in dom(\rho_2)\,.\,(\rho_1(j) \leadsto \rho_2(j))$$

The constraint $\rho_1 \leadsto \rho_2$ states that for each local variable j in ρ_2, it must be possible to lift the value of local variable j from the binding-time type specified in ρ_1 to the binding-time type specified in ρ_2.

3.4 Valid binding-time annotations

In this section we present a system of inference rules that define well-annotatedness for programs written in the annotated core language. The inference rules defined below following are summarized in appendix A.

A *label environment* λ maps each program label l to the binding-time stack and local variable binding-time environment for the corresponding program point[16]:

$$\lambda : Label \overset{\text{fin}}{\to} (Stack_{BT} \times Locals_{BT})$$

In each of the following inference rules, the conclusion has the form:

$$\lambda, \sigma, \rho \;\vdash\; c \;:\; \sigma', \rho'$$

The meaning of such an assertion is: given the label environment λ, the binding-time stack σ and the local variable binding-time environment ρ, execution of the instruction c will lead to the resulting binding-time stack σ' and local variable binding-time environment ρ'.

The inference rule for an abstract `arraylength` instruction ensures that the instruction is annotated as D iff the binding-time type b_1 of the array operand is D and that the binding-time type b_2 of the result being pushed onto the operand stack (corresponding to the length of the array) is the same as the binding-time type b of the instruction itself:

$$\lambda, (b_1 :: \sigma), \rho \;\vdash\; \texttt{arraylength}_b \;:\; (b_2 :: \sigma), \rho \qquad \text{if } b_1 \lhd\!\rhd b \wedge b = b_2 \tag{1}$$

In other words, if the stack operand to the `arraylength` instruction is partially static then the instruction may be fully evaluated at specialization-time since the length of the array will be known.

The inference rule for an `arrayload` instruction ensures that the binding-time type b_1 of the index operand is D iff the binding-time type b_2 of the array reference operand is D, that the binding-time type b of the instruction itself is the same as that of the array reference and that the binding-time type b_3 of the result being pushed onto the operand stack corresponds to the binding-time type of components in the array:

$$\lambda, (b_1 :: b_2 :: \sigma), \rho \;\vdash\; \texttt{arrayload}_b \;:\; (b_3 :: \sigma), \rho \qquad \text{if } b_1 \lhd\!\rhd b_2 \wedge b_2 = b \wedge b_3[\,] \leq b_2 \tag{2}$$

Similarly, the inference rule for an `arraystore` instruction ensures that the binding-time type b_2 of the index operand is D iff the binding-time type b_3 of the array reference operand is D, that the binding-time type b of the instruction itself is the same as that of the array reference and that the binding-time type b_1 of the operand being stored into the array corresponds to the binding-time type of components in the array:

$$\lambda, (b_1 :: b_2 :: b_3 :: \sigma), \rho \;\vdash\; \texttt{arraystore}_b \;:\; \sigma, \rho \qquad \text{if } b_2 \lhd\!\rhd b_3 \wedge b_3 = b \wedge b_1[\,] \leq b_3 \tag{3}$$

The inference rule for a `binop` instruction ensures that the binding-time types b_1 and b_2 of the operands are the same as the binding-time type b of the instruction itself and that the binding-time

[15] The definition of $\rho_1 \leadsto \rho_2$ should possibly take only live variables in ρ_2 into account.

[16] The program point corresponding to l is the program point at which l is represented by a `label` l instruction.

type b_3 of the result being pushed onto the operand stack is also the same as the binding-time type of the instruction itself:

$$\lambda, (b_1 :: b_2 :: \sigma), \rho \vdash \mathtt{binop}_b \odot \, : \, (b_3 :: \sigma), \rho \qquad \text{if } b_1 = b_2 = b = b_3 \tag{4}$$

The inference rule for a `const` instruction simply states that a value with binding-time type S is being pushed onto the operand stack:

$$\lambda, \sigma, \rho \vdash \mathtt{const}\ k \, : \, (\mathsf{S} :: \sigma), \rho \tag{5}$$

The inference rule for a `dup` instruction ensures that the instruction is annotated as D iff the binding-time type b_1 of the stack operand is D and that the resulting duplicates being pushed onto the operand stack have the same binding-time types as the operand:

$$\lambda, (b_1 :: \sigma), \rho \vdash \mathtt{dup}_b \, : \, (b_2 :: b_3 :: \sigma), \rho \qquad \text{if } b_1 \lhd\!\rhd b \land b_1 = b_2 = b_3 \tag{6}$$

The inference rule for a `goto` instruction ensures that the binding-time stack σ and local variable binding-time environment ρ correspond to the binding-time stack and local variable binding-time environment at the branch target l:

$$\lambda, \sigma, \rho \vdash \mathtt{goto}\ l \, : \, \sigma', \rho' \qquad \text{if } \lambda(l) = (\sigma, \rho) \tag{7}$$

Note that there are no requirements on the resulting binding-time stack σ' and local variable binding-time environment ρ' after an unconditional branch[17].

The inference rule for an `if` instruction ensures that the instruction is annotated as D iff the binding-time types b_1 and b_2 of the two stack operands are D and that the resulting binding-time stack σ and local variable binding-time environment ρ correspond to the binding-time stack and local variable binding-time environment at the branch target l:

$$\lambda, (b_1 :: b_2 :: \sigma), \rho \vdash \mathtt{if}_b\ r\ l \, : \, \sigma, \rho \qquad \text{if } b_1 \lhd\!\rhd b \land b_2 \lhd\!\rhd b \land \lambda(l) = (\sigma, \rho) \tag{8}$$

Note that the two stack operands to an `if` instruction need not have the same binding-time type; for example, the operands could be two references to partially static arrays of different dimensionality[18].

The inference rule for a `label` instruction ensures that the binding-time stack σ and the local variable binding-time environment ρ is the same before and after the instruction, and that σ and ρ are consistent with the binding-time stack and local variable binding-time environment for label l according to λ:

$$\lambda, \sigma, \rho \vdash \mathtt{label}\ l \, : \, \sigma, \rho \qquad \text{if } \lambda(l) = (\sigma, \rho) \tag{9}$$

The inference rule for a `load` instruction ensures that the instruction is annotated as D iff the binding-time type $\rho(j)$ of the local variable j being loaded is D and that the binding-time type b_1 of the result being pushed onto the operand stack is the same as the binding-time type of local variable j:

$$\lambda, \sigma, \rho \vdash \mathtt{load}_b\ j \, : \, (b_1 :: \sigma), \rho \qquad \text{if } \rho(j) = b_1 \land b_1 \lhd\!\rhd b \tag{10}$$

The inference rule for a `newarray` instruction ensures that the instruction is annotated as D iff the binding-time type b_1 of the stack operand is D, that the binding-time type b_2 of the result being pushed onto the operand stack is the same as the binding-time type b of the instruction itself and that b is a binding-time type of a partially static array or D:

$$\lambda, (b_1 :: \sigma), \rho \vdash \mathtt{newarray}_b\ d \, : \, (b_2 :: \sigma), \rho \qquad \text{if } b_1 \lhd\!\rhd b \land \exists b_3 . b_3[\,] \leq b \land b = b_2 \tag{11}$$

[17]Informally, there is no relation between a `goto` instruction and the (`label`) instruction immediately following it (if any).

[18]The two operands to an `if` instruction need not have the same type. Either both operands must be integers, or both operands must be of array types (but need not have the same component type, cf. section 2.2).

In other words, a `newarray` instruction must be annotated with a binding-time type corresponding to the array that it will create. This is necessary to guide the specializer: if a `newarray` instruction was simply annotated with S or D (as is any other annotated instruction), then the specializer could not properly unfold a partially static array being created by the instruction.

In order to properly unfold a partially static array created by a `newarray` instruction, the specializer must know 'how static' the components of the array are supposed to be, i.e., whether the binding-time type of the array components is D or not. We describe specialization of partially static arrays in more detail in section 5.1.

The inference rule for a `pop` instruction ensures that the instruction is annotated as D iff the binding-time type b_1 of the stack operand to be discarded is D:

$$\lambda, (b_1 :: \sigma), \rho \vdash \texttt{pop}_b \ : \ \sigma, \rho \qquad \text{if } b_1 \lhd\!\rhd b \tag{12}$$

The inference rule for the `return` instruction simply states that the binding-time type of the stack operand to be returned as the result of the program must be D:

$$\lambda, (\mathsf{D} :: \sigma), \rho \vdash \texttt{return} \ : \ \sigma', \rho' \tag{13}$$

The operand to a `return` must be dynamic since a value must be returned from the residual program.

Note that there are no requirements on the resulting binding-time stack σ' and local variable binding-time environment ρ' after a `return` instruction[19].

The inference rule for a `store` instruction ensures that the instruction is annotated as D iff the binding-time type b_1 of the stack operand is D and that the resulting binding-time type of the local variable j being stored into is the same as b_1:

$$\lambda, (b_1 :: \sigma), \rho \vdash \texttt{store}_b \ j \ : \ \sigma, \rho[j \mapsto b_1] \qquad \text{if } b_1 \lhd\!\rhd b \tag{14}$$

The inference rule for a `swap` instruction ensures that the instruction is annotated as D iff the binding-time types b_1 and b_2 of the stack operands both are D and that the binding-time types b_3 and b_4 of the results being pushed onto the stack correspond to b_1 and b_2 (swapped):

$$\lambda, (b_1 :: b_2 :: \sigma), \rho \vdash \texttt{swap}_b \ : \ (b_4 :: b_3 :: \sigma), \rho \qquad \text{if } (b_1, b_2) \lhd\!\rhd b \wedge b_1 = b_3 \wedge b_2 = b_4 \tag{15}$$

Note that `swap` may be fully evaluated at specialization time in case $b_1 \neq \mathsf{D} \vee b_2 \neq \mathsf{D}$.

The inference rule for the special `liftTop` instruction simply ensures that the topmost stack operand has binding-time type S and that the result being pushed onto the operand stack has binding-time type D:

$$\lambda, (\mathsf{S} :: \sigma), \rho \vdash \texttt{liftTop} \ : \ (\mathsf{D} :: \sigma), \rho \tag{16}$$

The inference rule for the special `lift2nd` instruction ensures that the second topmost stack operand has binding-time type S, that the second topmost operand on the resulting operand stack has binding-time type D and that the binding-time type of the topmost stack operand is the same before and after the instruction:

$$\lambda, (b_1 :: \mathsf{S} :: \sigma), \rho \vdash \texttt{lift2nd} \ : \ (b_2 :: \mathsf{D} :: \sigma), \rho \qquad \text{if } b_1 = b_2 \tag{17}$$

The inference rule for a `lift` j instruction ensures that the binding-time type of the variable to be lifted is S before the instruction and that the variable has binding-time type D after the instruction:

$$\lambda, \sigma, \rho[j \mapsto \mathsf{S}] \vdash \texttt{lift} \ j \ : \ \sigma, \rho[j \mapsto \mathsf{D}] \tag{18}$$

Each of the above inference rules deal only with a single instruction. In order to define well-annotatedness of annotated core language programs, we introduce the following two inference rules dealing with sequences of instructions:

$$\frac{\lambda, \sigma, \rho \vdash c \ : \ \sigma_1, \rho_1 \qquad \lambda, \sigma_1, \rho_1 \vdash cs \ : \ \sigma_2, \rho_2}{\lambda, \sigma, \rho \vdash c :: cs \ : \ \sigma_2, \rho_2} \tag{19}$$

[19] As for a `goto` instruction, there is no relation between a `return` instruction and the instruction immediately following it (if any).

$$\lambda, \sigma, \rho \vdash nil : \sigma, \rho \tag{20}$$

The former rule states that a non-empty instruction sequence is well-annotated iff the first instruction as well as the remainder of the sequence is well-annotated; this rule may be applied recursively, in connection with the latter rule, to prove that a program is well-annotated.

The latter rule states that the binding-time stack σ and the local variable binding-time environment ρ are unchanged by an empty instruction sequence.

We are now in a position to define what a well-annotated core language program is: a program p is well-annotated with respect to a specific label environment λ and local variable binding-time environment ρ iff it holds that

$$\exists \lambda, \sigma_1, \rho_1 . (\lambda, nil, \rho \vdash p : \sigma_1, \rho_1)$$

where ρ is the initial local variable binding-time environment, holding the binding-time types of the program parameters.

3.5 Constraint generation

The inference rules presented in the previous section define well-annotatedness for an annotated core language program. They do, however, not describe how to achieve a valid annotated core language program from a given core language program. Specifically, it has not been specified where lift instructions should be inserted or how to annotate each instruction of the original core language program with a valid binding-time type[20].

In this section, we define a function $\mathcal{C}[\![\cdot]\!]$ for generating binding-time constraints from a core language program. For a given core language program, any valid solution to the generated set of constraints may be used for translating the program into a well-annotated core language program (as described in section 3.7).

The below definition of the constraint generation function is derived from the inference rules of the previous section in the following way: for each core language instruction, the function $\mathcal{C}[\![\cdot]\!]$ generates one or more binding-time constraints, corresponding to the side-conditions specified in the inference rule for the corresponding annotated core language instruction.

For example, the inference rule (12) for the `pop` instruction has the following side-condition:

$$\ldots \quad \text{if } b_1 \triangleleft\!\triangleright b$$

whereas the $\mathcal{C}[\![\cdot]\!]$ function generates the following set of binding-time constraints for a `pop` instruction:

$$\{\beta_1 \triangleleft\!\triangleright \beta\}$$

A valid solution S to the set C of generated constraints substitutes each binding-time variable β with a binding-time type b, where $b \in BT$, such that all contraints in C are satisfied (cf. section 3.3). For example, a valid solution S to the above constraint set must satisfy:

$$S(\beta_1) \triangleleft\!\triangleright S(\beta)$$

Strictly speaking, the binding-time constraints used in this and the following sections are different from the binding-time constraints defined in section 3.3 and used in the inference rules of the previous section: whereas the constraints used hitherto were constraints on concrete binding-time types from the domain BT, the binding-time constraints generated for a given core language program are constraints on *binding-time variables* β, β_1, β_2, etc. Despite the slightly different meaning of binding-time constraints, we use the same notation as before for the binding-time constraints generated by the $\mathcal{C}[\![\cdot]\!]$ function.

In the following definition of the $\mathcal{C}[\![\cdot]\!]$ function, we also take the liberty of using the symbols λ, σ and ρ, etc., with a sligthly different meaning:

[20] A valid annotated core language program may always be obtained by annotating each instruction of the original core language program as dynamic (where relevant). We are, however, mainly interested in non-trivial solutions.

- We use σ (with various superscripts and/or subscripts) to describe a *stack of binding-time variables* $\beta_1 :: \beta_2 :: \cdots :: nil$. Each binding-time variable in the stack represents the as of yet unknown binding-time type of the corresponding stack operand at a particular point in a core language program.
- We use ρ (with various superscripts and/or subscripts) to describe a *binding-time variable environment* which maps each local variable index j to a binding-time variable representing the as of yet unknown binding-time type of local variable number j at a particular point in a core language program.
- The *label environment* λ now maps each program label l to the binding-time variable stack and binding-time variable environment for the program point labelled l.

The constraint generation function $\mathcal{C}[\![\cdot]\!]$ takes three parameters: a sequence of core language instructions, a binding-time variable stack and a binding-time variable environment. The label environment λ is assumed to have been built already and is not changed by the $\mathcal{C}[\![\cdot]\!]$ function[21].

It is also assumed that the relevant number of variables have already been associated with each core language instruction. For an instruction c annotated as $c_{\beta}^{\beta_1,\beta_2,\cdots}$, the binding-time variable (β) written as a subscript to the instruction represents the (unknown) binding-time type of the instruction itself. The variables $(\beta_1, \beta_2, \ldots)$ written as superscripts to the instruction are used in determining where lift instructions should be inserted in the resulting annotated program; this is described in section 3.7.

We define $\mathcal{C}[\![\cdot]\!]$ as follows:

$$\mathcal{C}[\![\mathtt{arraylength}_{\beta}^{\beta_2} :: cs]\!]\,(\beta_1 :: \sigma)\,\rho \;=\; \{\beta_1 \lhd\rhd \beta, \beta \leadsto \beta_2\} \,\cup\, \mathcal{C}[\![cs]\!]\,(\beta_2 :: \sigma)\,\rho$$

$$\begin{aligned}&\mathcal{C}[\![\mathtt{arrayload}_{\beta}^{\beta_1,\beta_3,\beta_4,\beta_5} :: cs]\!]\,(\beta_1 :: \beta_2 :: \sigma)\,\rho \;=\\ &\qquad \{\beta_1 \leadsto \beta_4, \beta_4 \lhd\rhd \beta_2, \beta_2 = \beta, \beta_3[\,] \le \beta_2, \beta_3 \leadsto \beta_5\} \,\cup\, \mathcal{C}[\![cs]\!]\,(\beta_5 :: \sigma)\,\rho\end{aligned}$$

$$\begin{aligned}&\mathcal{C}[\![\mathtt{arraystore}_{\beta}^{\beta_1,\beta_2,\beta_4,\beta_5} :: cs]\!]\,(\beta_1 :: \beta_2 :: \beta_3 :: \sigma)\,\rho \;=\\ &\qquad \{\beta_1 \leadsto \beta_4, \beta_2 \leadsto \beta_5, \beta_5 \lhd\rhd \beta_3, \beta_3 = \beta, \beta_4[\,] \le \beta_3\} \,\cup\, \mathcal{C}[\![cs]\!]\,\sigma\,\rho\end{aligned}$$

$$\mathcal{C}[\![\mathtt{binop}_{\beta}^{\beta_1,\beta_2,\beta_3}\ \odot :: cs]\!]\,(\beta_1 :: \beta_2 :: \sigma)\,\rho \;=\; \{\beta_1 \leadsto \beta, \beta_2 \leadsto \beta, \beta \leadsto \beta_3\} \,\cup\, \mathcal{C}[\![cs]\!]\,(\beta_3 :: \sigma)\,\rho$$

$$\mathcal{C}[\![\mathtt{const}^{\beta_1}\ k :: cs]\!]\,\sigma\,\rho \;=\; \mathcal{C}[\![cs]\!]\,(\beta_1 :: \sigma)\,\rho$$

$$\mathcal{C}[\![\mathtt{dup}_{\beta}^{\beta_1,\beta_2,\beta_3} :: cs]\!]\,(\beta_1 :: \sigma)\,\rho \;=\; \{\beta_1 \lhd\rhd \beta, \beta_1 \leadsto \beta_2, \beta_1 \leadsto \beta_3\} \,\cup\, \mathcal{C}[\![cs]\!]\,(\beta_2 :: \beta_3 :: \sigma)\,\rho$$

$$\begin{aligned}&\mathcal{C}[\![\mathtt{goto}^{\sigma,\sigma_1,\rho,\rho_1}\ l :: \mathtt{label}\ l' :: cs]\!]\,\sigma\,\rho \;=\; \{\sigma \leadsto \sigma_1, \rho \leadsto \rho_1\} \,\cup\, \mathcal{C}[\![cs]\!]\,\sigma_2\,\rho_2\\ &\qquad \text{where } (\sigma_1, \rho_1) = \lambda(l) \wedge (\sigma_2, \rho_2) = \lambda(l')\end{aligned}$$

$$\begin{aligned}&\mathcal{C}[\![\mathtt{goto}^{\sigma,\sigma_1,\rho,\rho_1}\ l :: nil]\!]\,\sigma\,\rho \;=\; \{\sigma \leadsto \sigma_1, \rho \leadsto \rho_1\}\\ &\qquad \text{where } (\sigma_1, \rho_1) = \lambda(l)\end{aligned}$$

$$\begin{aligned}&\mathcal{C}[\![\mathtt{if}_{\beta}^{\beta_1,\beta_2,\beta_3,\beta_4,\sigma,\sigma_1,\sigma_2,\rho,\rho_1,\rho_2}\ r\ l :: cs]\!]\,(\beta_1 :: \beta_2 :: \sigma)\,\rho \;=\\ &\qquad \{\beta_1 \leadsto \beta_3, \beta_2 \leadsto \beta_4, \beta_3 \lhd\rhd \beta, \beta_4 \lhd\rhd \beta, \sigma \leadsto \sigma_1, \sigma \leadsto \sigma_2, \rho \leadsto \rho_1, \rho \leadsto \rho_2\} \,\cup\, \mathcal{C}[\![cs]\!]\,\sigma_2\,\rho_2\\ &\qquad \text{where } (\sigma_1, \rho_1) = \lambda(l)\end{aligned}$$

[21] In practice, the label environment λ may be built by a preprocessing phase before constraint generation (see section 3.8).

$$\mathcal{C}[\![\texttt{label}^{\sigma,\sigma_1,\rho,\rho_1}\ l :: cs]\!]\ \sigma\ \rho\ =\ \{\sigma \rightsquigarrow \sigma_1, \rho \rightsquigarrow \rho_1\}\ \cup\ \mathcal{C}[\![cs]\!]\ \sigma_1\ \rho_1$$
$$\text{where } (\sigma_1, \rho_1) = \lambda(l)$$

$$\mathcal{C}[\![\texttt{load}_{\beta}^{\beta_1,\beta_2}\ j :: cs]\!]\ \sigma\ \rho\ =\ \{\beta_1 \lhd\!\rhd \beta, \beta_1 \rightsquigarrow \beta_2\}\ \cup\ \mathcal{C}[\![cs]\!]\ (\beta_2 :: \sigma)\ \rho$$
$$\text{where } \beta_1 = \rho(j)$$

$$\mathcal{C}[\![\texttt{newarray}_{\beta}^{\beta_1,\beta_4}\ d :: cs]\!]\ (\beta_1 :: \sigma)\ \rho\ =\ \{\beta_1 \rightsquigarrow \beta_4, \beta_4 \lhd\!\rhd \beta, \beta_3[\,] \leq \beta\}\ \cup\ \mathcal{C}[\![cs]\!]\ (\beta :: \sigma)\ \rho$$

$$\mathcal{C}[\![\texttt{pop}_{\beta} :: cs]\!]\ (\beta_1 :: \sigma)\ \rho\ =\ \{\beta_1 \lhd\!\rhd \beta\}\ \cup\ \mathcal{C}[\![cs]\!]\ \sigma\ \rho$$

$$\mathcal{C}[\![\texttt{return}^{\beta_1} :: \texttt{label}\ l :: cs]\!]\ (\beta_1 :: \sigma)\ \rho\ =\ \{\beta_1 \rightsquigarrow \mathsf{D}\}\ \cup\ \mathcal{C}[\![cs]\!]\ \sigma_1\ \rho_1$$
$$\text{where } (\sigma_1, \rho_1) = \lambda(l)$$

$$\mathcal{C}[\![\texttt{return}^{\beta_1} :: nil]\!]\ (\beta_1 :: \sigma)\ \rho\ =\ \{\beta_1 \rightsquigarrow \mathsf{D}\}$$

$$\mathcal{C}[\![\texttt{store}_{\beta}\ j :: cs]\!]\ (\beta_1 :: \sigma)\ \rho\ =\ \{\beta_1 \lhd\!\rhd \beta\}\ \cup\ \mathcal{C}[\![cs]\!]\ \sigma\ \rho[j \mapsto \beta_1]$$

$$\mathcal{C}[\![\texttt{swap}_{\beta}^{\beta_1,\beta_2,\beta_3,\beta_4} :: cs]\!]\ (\beta_1 :: \beta_2 :: \sigma)\ \rho\ =$$
$$\{(\beta_1, \beta_2) \lhd\!\rhd \beta, \beta_1 \rightsquigarrow \beta_3, \beta_2 \rightsquigarrow \beta_4\}\ \cup\ \mathcal{C}[\![cs]\!]\ (\beta_4 :: \beta_3 :: \sigma)\ \rho$$

$$\mathcal{C}[\![nil]\!]\ \sigma\ \rho\ =\ \emptyset$$

In each of the 'cases' defined above, a fresh variable is implicitly used every time a new variable is mentioned—except for variables appearing in the binding-time variable stack or binding-time variable environment parameters to the $\mathcal{C}[\![\cdot]\!]$ function. For example, the binding-time variables β and β_2 (but not β_1, σ and ρ) appearing in the definition of $\mathcal{C}[\![\cdot]\!]$ for an `arraylength` instruction are fresh variables.

An equality constraint of the form $\beta_1 = \beta_2$, which is generated for an `arrayload` or `arraystore` instruction, asserts that the binding-time variables β_1 and β_2 must have the same value for any valid solution S, i.e., that $S(\beta_1) = S(\beta_2)$ must hold.

Note that the binding-time constraints generated for some instructions, e.g. `arraylength`, include lift constraints instead of the equality constraints appearing in the corresponding inference rules defined in the previous section. We use lift constraints to express that lift instructions may have to be inserted in the resulting annotated core language program.

For example, the lift constraint generated for an `arraylength` instruction indicates that the result pushed onto the operand stack by the instruction may have to be lifted to a more dynamic context: in case $S(\beta) = \mathsf{S}$ and $S(\beta_2) = \mathsf{D}$ a `liftTop` instruction must be inserted after the annotated `arraylength` instruction (see section 3.7).

For most core language instructions, lift constraints are generated for the operand(s) of the instructions as well as the resulting value(s) pushed onto the operand stack (if any). The intention is that lifting may occur before and/or after most instructions, in order to yield the most static annotation of the resulting annotated core language program (i.e., having as many instructions as possible annotated as non-dynamic).

As an exception to this rule, lift constraints are not generated for operands (or results) known to be array references. Array references cannot be lifted to a more dynamic context (cf. section 3.3), so we avoid generating superfluous lift constraints for operands known to be array references.

Furthermore, no lift constraints are generated for a `store` j instruction. This is because we are free to lift the value of local variable number j directly at any point where it is 'merged' with a dynamic value. In other words, we will never need to lift the top-most stack operand before storing it into a local variable.

For `goto`, `if` and `label` instructions, stack and local variable environment lift constraints are generated to express that the operand stack and the local variables may have to be lifted in connection with these intructions. In the next section we show how to 'unfold' stack and local variable lift constraints to sets of simple lift and equality constraints on binding-time variables.

In the definition of the $\mathcal{C}[\![\cdot]\!]$ function it is assumed that a `goto` or `return` instruction will either be followed by a `label` instruction or be the last instruction of the program (cf. section 2.4). When a `goto` or `return` instruction is followed by a `label` l instruction, the binding-time variable stack σ_1 and binding-time variable environment ρ_1, where $(\sigma_1, \rho_1) = \lambda(l)$, is used (recursively) in the constraint generation for the succeeding instructions cs. When a `goto` or `return` instruction appears as the last instruction of the program, recursion stops.

For a given core language program p, the set C of constraints that must hold for any valid solution S may be calculated as follows:

$$C = \mathcal{C}[\![p']\!]\, nil\, \rho$$

where p' corresponds to p, except that the relevant binding-time variables have been associated with each instruction, and ρ is a binding-time variable environment holding the binding-time variables representing the (unknown) binding-time types of the program parameters.

3.6 Constraint solving

The set of constraints generated by the $\mathcal{C}[\![\cdot]\!]$ function defined in the previous section may be solved by first translating our constraints to the binding-time constraints defined by Birkedal and Welinder [8] and then using their constraint solver to find an optimal solution to the resulting constraint set.

Our binding-time domain BT corresponds to a subset of Birkedal and Welinder's binding-time domain. Informally, the binding-time types D and S are the same in our binding-time domain BT as in Birkedal and Welinder's binding-time domain, whereas a binding-time type of the form $b[\,]$ in our domain corresponds to a 'structured' type $[b]$ with only one component in theirs.

The relation may be formalized as follows:

$$\begin{array}{lcll}
\mathcal{BW}_{BT}[\![\mathsf{D}]\!] & = & \mathsf{D} & \\
\mathcal{BW}_{BT}[\![\mathsf{S}]\!] & = & \mathsf{S} & \\
\mathcal{BW}_{BT}[\![b[\,]]\!] & = & [b'] & \quad \text{where } b' = \mathcal{BW}_{BT}[\![b]\!] \\
\mathcal{BW}_{BT}[\![\bot]\!] & = & \bot &
\end{array}$$

The function $\mathcal{BW}_{BT}[\![\cdot]\!]$ is an injective, monotonic increasing function, mapping a binding-time type from our BT domain to the corresponding type in Birkedal and Welinder's binding-time domain.

The function $\mathcal{BW}_{C}[\![\cdot]\!]$ may be used for translating a constraint generated by our $\mathcal{C}[\![\cdot]\!]$ function to a set of Birkedal and Welinder constraints. We define $\mathcal{BW}_C[\![\cdot]\!]$ as follows:

$$
\begin{array}{lcl}
\mathcal{BW}_C[\![\beta_1 \lhd\!\rhd \beta_2]\!] & = & \{\beta_1 \rhd \beta_2, \beta_1 \rhd \beta_2\} \\
\mathcal{BW}_C[\![(\beta_1,\beta_2) \lhd\!\rhd \beta_3]\!] & = & \{(\beta_1,\beta_2) \rhd \beta_3, \beta_3 \rhd \beta_1, \beta_3 \rhd \beta_2\} \\
\mathcal{BW}_C[\![\beta_1[\,] \leq \beta_2]\!] & = & \{[\beta_1] \leq \beta_2\} \\
\mathcal{BW}_C[\![\beta_1 \leadsto \beta_2]\!] & = & \{\beta_1 \leadsto \beta_2\} \\
\mathcal{BW}_C[\![\sigma_1 \leadsto \sigma_2]\!] & = & \left\{ \begin{array}{cl} \emptyset & \text{if } \sigma_1 = \sigma_2 = nil \\ \{\beta_1 \leadsto \beta_2\} & \text{if } \sigma_1 = \beta_1 :: nil \wedge \sigma_2 = \beta_2 :: nil \\ \{\beta_1 \leadsto \beta_2, \beta_1' \leadsto \beta_2'\} \cup Eq(\sigma_1', \sigma_2') & \text{if } \sigma_1 = \beta_1 :: \beta_1' :: \sigma_1' \wedge \\ & \quad \sigma_2 = \beta_2 :: \beta_2' :: \sigma_2' \end{array} \right. \\
\mathcal{BW}_C[\![\rho_1 \leadsto \rho_2]\!] & = & \bigcup\limits_{j \in dom(\rho_2)} \{\rho_1(j) \leadsto \rho_2(j)\}
\end{array}
$$

where

$$
Eq(\sigma_1, \sigma_2) = \left\{ \begin{array}{cl} \{\beta_1 = \beta_2\} \cup Eq(\sigma_1', \sigma_2') & \text{if } \sigma_1 = \beta_1 :: \sigma_1' \wedge \sigma_2 = \beta_2 :: \sigma_2' \\ \emptyset & \text{if } \sigma_1 = \sigma_2 = nil \end{array} \right.
$$

The definition of $\mathcal{BW}_C[\![\cdot]\!]$ is derived straightforwardly from the definition of our constraint operators in section 3.3.

We extend the definition of the $\mathcal{BW}_C[\![\cdot]\!]$ function for translating a set C of constraints generated by our $\mathcal{C}[\![\cdot]\!]$ function to a set of corresponding Birkedal and Welinder constraints as follows:

$$
\mathcal{BW}_C[\![C]\!] = \bigcup_{c \in C} \mathcal{BW}_C[\![c]\!]
$$

After translating the generated constraints into Birkedal and Welinder's constraint system by means of the $\mathcal{BW}_C[\![\cdot]\!]$ function, their constraint solver may be used to find a solution to the resulting set of constraints.

Given a set of binding-time constraints and an 'initial solution' S_0, specifying the minimum binding-time type for each program parameter, the constraint solver finds a solution S_{BW} mapping each binding-time variable β, β_1, β_2, etc., to a binding-time type from Birkedal and Welinder's binding-time domain. For each binding-time type $S_0(\beta)$ specified by the initial solution, the binding-time type specified by the solution S_{BW} will be at least as dynamic as that specified by the initial solution. In other words, $S_0(\beta) \sqsubseteq S_{BW}(\beta)$ holds for all β bound in the initial solution S_0.

An important feature of the constraint set generated by our $\mathcal{C}[\![\cdot]\!]$ function for a given core language program is that a valid solution S_{BW} will include only binding-time types from the subset of Birkedal and Welinder's binding-time domain that corresponds to our BT domain. Hence, the inverse function to $\mathcal{BW}_{BT}[\![\cdot]\!]$ may be used to translate the solution S_{BW} in Birkedal and Welinder's binding-time domain 'back' into our BT domain, although $\mathcal{BW}_{BT}^{-1}$ is only a partial function from Birkedal and Welinder's binding-time domain to our BT domain.

The solution S_{BW} found by Birkedal and Welinder's constraint solver will be optimal in the sense that as many binding-time variables as possible will be non-dynamic. Any binding-time variable that is not otherwise constrained by one of the generated constraints will be resolved to S. Furthermore, since the function $\mathcal{BW}_{BT}[\![\cdot]\!]$ defined above is monotonic increasing, the inverse function $\mathcal{BW}_{BT}^{-1}[\![\cdot]\!]$ will also be monotonic increasing, and the resulting solution S, defined as

$$
S = \mathcal{BW}_{BT}^{-1} \circ S_{BW}
$$

will be optimal in our BT domain as well. This implies that as much evaluation as possible may be performed by a specializer which is guided by the resulting program annotations.

We refer the interested reader to Birkedal and Welinder [8] for a more thorough explanation of their constraint system and constraint solver.

3.7 Translation to annotated code

In this section we describe how to translate a core language program to a well-annotated core language program, that is, where to insert lift instructions and how to annotate each instruction of the original program with a valid binding-time type (cf. section 3.4).

Given a valid solution S to the binding-time constraints that have been generated for the program, each of the instructions in the original core language program translates to one or more instructions from the annotated core language instruction set as follows:

Core language instruction	*Resulting annotated code*
$\texttt{arraylength}_{\beta}^{\beta_2}$	$\texttt{arraylength}_{B}$ $\mathit{liftTop}(B, B_2)$
$\texttt{arrayload}_{\beta}^{\beta_1,\beta_3,\beta_4,\beta_5}$	$\mathit{liftTop}(B_1, B_4)$ $\texttt{arrayload}_{B}$ $\mathit{liftTop}(B_3, B_5)$
$\texttt{arraystore}_{\beta}^{\beta_1,\beta_2,\beta_4,\beta_5}$	$\mathit{lift2nd}(B_2, B_5)$ $\mathit{liftTop}(B_1, B_4)$ $\texttt{arraystore}_{B}$
$\texttt{binop}_{\beta}^{\beta_1,\beta_2,\beta_3}\ \odot$	$\mathit{lift2nd}(B_2, B)$ $\mathit{liftTop}(B_1, B)$ $\texttt{binop}_{B}\ \odot$ $\mathit{liftTop}(B, B_3)$
$\texttt{const}^{\beta_1}\ k$	$\texttt{const}\ k$ $\mathit{liftTop}(\texttt{S}, B_1)$
$\texttt{dup}_{\beta}^{\beta_1,\beta_2,\beta_3}$	$\texttt{dup}_{B}$ $\mathit{lift2nd}(B_1, B_3)$ $\mathit{liftTop}(B_1, B_2)$
$\texttt{goto}^{\sigma,\sigma_1,\rho,\rho_1}\ l$	$\mathit{liftStack}(\sigma, \sigma_1)$ $\mathit{liftVars}(\rho, \rho_1)$ $\texttt{goto}\ l$
$\texttt{if}_{\beta}^{\beta_1,\beta_2,\beta_3,\beta_4,\sigma,\sigma_1,\sigma_2,\rho,\rho_1,\rho_2}\ r\ l$	$\mathit{lift2nd}(B_2, B_4)$ $\mathit{liftTop}(B_1, B_3)$ $\texttt{if}_{B}\ r\ l'$ $\mathit{liftStack}(\sigma, \sigma_2)$ $\mathit{liftVars}(\rho, \rho_2)$ $\texttt{goto}\ l''$ $\texttt{label}\ l'$ $\mathit{liftStack}(\sigma, \sigma_1)$ $\mathit{liftVars}(\rho, \rho_1)$ $\texttt{goto}\ l$ $\texttt{label}\ l''$
$\texttt{label}^{\sigma,\sigma_1,\rho,\rho_1}\ l$	$\mathit{liftStack}(\sigma, \sigma_1)$ $\mathit{liftVars}(\rho, \rho_1)$ $\texttt{label}\ l$
$\texttt{load}_{\beta}^{\beta_1,\beta_2}\ j$	$\texttt{load}_{B}\ j$ $\mathit{liftTop}(B_1, B_2)$
$\texttt{newarray}_{\beta}^{\beta_1,\beta_4}\ d$	$\mathit{liftTop}(B_1, B_4)$ $\texttt{newarray}_{B}\ d$
$\texttt{pop}_{\beta}$	$\texttt{pop}_{B}$

$\texttt{return}^{\beta_1}$	$\mathit{liftTop}(B_1, \mathsf{D})$ $\texttt{return}$
$\texttt{store}_\beta\ j$	$\texttt{store}_B\ j$
$\texttt{swap}_\beta^{\beta_1,\beta_2,\beta_3,\beta_4}$	$\texttt{swap}_B$ $\mathit{lift2nd}(B_1, B_3)$ $\mathit{liftTop}(B_2, B_4)$

For brevity we use B and B_i, where $i = 1, 2, \ldots$, as shorthands for $S(\beta)$ and $S(\beta_i)$. We also use a more relaxed and informal notation for the resulting sequences of annotated core language instructions.

For example, the core language instruction sequence

$$\texttt{arraylength}_\beta^{\beta_2} :: cs$$

translates to the following annotated core language instruction sequence in case $S(\beta) = \mathsf{S}$ and $S(\beta_2) = \mathsf{D}$:

$$\texttt{arraylength}_\mathsf{S} :: \texttt{liftTop} :: cs'$$

where cs' is the translation of cs.

Otherwise, the instruction sequence simply translates to:

$$\texttt{arraylength}_B :: cs'$$

where $B = S(\beta)$.

We define the function *liftTop*, which is used in the above translation, as follows:

$$\mathit{liftTop}(B_1, B_2) = \begin{cases} \texttt{liftTop} & \text{if } B_1 = \mathsf{S} \wedge B_2 = \mathsf{D} \\ \textit{(no-op)} & \text{otherwise} \end{cases}$$

The notation *(no-op)* is used to indicate that no instruction is inserted in case a lift instruction is not required. In principle, a no-op instruction could be inserted—provided the `nop` instruction from the actual JVM instruction set was included in our core language—but we simply avoid generating any unnecessary lift instruction.

Similarly, we define the function *lift2nd* as follows:

$$\mathit{lift2nd}(B_1, B_2) = \begin{cases} \texttt{lift2nd} & \text{if } B_1 = \mathsf{S} \wedge B_2 = \mathsf{D} \\ \textit{(no-op)} & \text{otherwise} \end{cases}$$

The function *liftStack* is used to insert zero, one or two stack lift instructions as required to 'lift the stack':

$$\mathit{liftStack}(\sigma_1, \sigma_2) = \begin{cases} \begin{array}{l} \mathit{lift2nd}(S(\beta_1'), S(\beta_2')) \\ \mathit{liftTop}(S(\beta_1), S(\beta_2)) \end{array} & \text{if } \sigma_1 = \beta_1 :: \beta_1' :: \sigma_1' \wedge \sigma_2 = \beta_2 :: \beta_2' :: \sigma_2' \\ \mathit{liftTop}(S(\beta_1), S(\beta_2)) & \text{if } \sigma_1 = \beta_1 :: \mathit{nil} \wedge \sigma_2 = \beta_2 :: \mathit{nil} \\ \textit{(no-op)} & \text{otherwise} \end{cases}$$

If the binding-time variable stacks σ_1 and σ_2 are of depth two (or greater) then at most two lift instructions are inserted: a `lift2nd` instruction in case $S(\beta_1') = \mathsf{S} \wedge S(\beta_2') = \mathsf{D}$ and a `liftTop` instruction in case $S(\beta_1) = \mathsf{S} \wedge S(\beta_2) = \mathsf{D}$. If both stacks hold only one binding-time variable then a `liftTop` instruction is inserted in case $S(\beta_1) = \mathsf{S}$ and $S(\beta_2) = \mathsf{D}$. If both σ_1 and σ_2 are empty then no lift instructions are inserted.

The function *liftVars* is used to insert one or more local variable lift instructions if required. The result of $\mathit{liftVars}(\rho_1, \rho_2)$ is the following sequence:

$$\begin{array}{c} \mathit{lift}(S(\rho_1(j_1)), S(\rho_2(j_1)))\ j_1 \\ \mathit{lift}(S(\rho_1(j_2)), S(\rho_2(j_2)))\ j_2 \\ \vdots \\ \mathit{lift}(S(\rho_1(j_n)), S(\rho_2(j_n)))\ j_n \end{array}$$

where $j_i \in dom(\rho_2)$ for $i = 1, 2, \ldots, n$; $j_x = j_y \Rightarrow x = y$ and n is the cardinality of $dom(\rho_2)$. We define the *lift* function as follows:

$$lift(B_1, B_2)\ j = \begin{cases} \texttt{lift}\ j & \text{if } B_1 = \mathsf{S} \wedge B_2 = \mathsf{D} \\ \textit{(no-op)} & \text{otherwise} \end{cases}$$

In other words, the result of $liftVars(\rho_1, \rho_2)$ is a (possibly empty) sequence of `lift` j instructions, depending on the binding-time types bound to the binding-time variables in ρ_1 and ρ_2.

For most of the core language instructions, translation to one ore more instructions of the annotated core language is fairly straightforward, as shown above. However, the translation of an `if` instruction warrants some explanation.

At most two stack lift instructions, `lift2nd` and `liftTop`, are inserted before the resulting annotated `if` instruction to lift its operands if so required. The resulting `if` instruction tests for the same relation r as the original core language instruction, but branches to a fresh label l' when the test succeeds. The reason for this is that we may have to insert stack and local variable environment lift instructions for each of the two branches of the conditional[22].

The resulting `if` instruction is immediately followed by lift instructions for the 'false' branch and a `goto` l'' instruction, where l'' is another fresh label, corresponding to the original 'false' branch. After this comes the label l', followed by lift instructions for the 'true' branch, a `goto` l instruction, where l is the target of the original `if` instruction, and the label l'' itself.

Given a core language program and a binding-time type for each program parameter, a corresponding annotated program may now be generated as follows:

- generate a set of binding-time constraints from the program (cf. section 3.5);
- translate the constraints to Birkedal and Welinder constraints and use their constraint solver to find a solution to the generated constraints (cf. section 3.6);
- use the solution found in the previous step to translate each core language instruction into one or more annotated core language instructions, as described above.

The net result will be a well-annotated core language program (cf. section 3.4) that may be specialized by an off-line specializer, given concrete values for the known program parameters; this is discussed in section 5.

3.8 Our implementation

The binding-time analysis presented in the preceeding sections has been implemented using the functional programming language *Standard ML* (SML) [14]. SML was chosen as implementation language for a number of reasons:

- SML supports user-defined abstract datatypes and pattern matching; this is very well-suited for analysing and manipulating program fragments.
- Our implementation is partly based on the *SML-JVM toolkit* [3], a toolkit for generating and manipulating Java bytecode and Java class files, which is implemented in SML[23].
- Birkedal and Welinder's implementation of their constraint solver, which we use for solving the generated constraints, is written in SML [7].
- Good SML compilers exist for most popular computing platforms, e.g. *Moscow ML* [15] which has been used in the development of our implementation.
- We already have much experience with programming in SML.

[22] In case lifting is necessary for only one of the branches, the translation of a core language `if` instruction to an annotated `if` instruction may be simplified. In general, the translation of a core language `if` instruction to an annotated `if` instruction may be simplified by using the negated test ($\neg r$) in case lifting is necessary for the 'true' branch of the conditional.

[23] At the time of writing, the SML-JVM toolkit only worked with Moscow ML [15].

3.8.1 Structure of the implementation

Our implementation of the binding-time analysis comprises the following SML modules:

Module	Implements
`BWbttype`	Birkedal and Welinder's binding-time types
`BWsolving`	Birkedal and Welinder's constraint solver
`Btanalysis`	The 'main module' of our binding-time analysis
`Btvar`	Definition of abstract binding-time variables
`Btvarmap`	Definition of maps with binding-time variables as keys
`Constrgen`	Constraint generation
`Constrsolving`	Constraint solving (using Birkedal and Welinder's solver)
`Corelang`	Core language and annotated core language
`Stackdepth`	Calculation of operand stack depths

The modules `BWbttype` and `BWsolving` are based on Birkedal and Welinder's implementation of the *SML-mix* partial evaluator for SML programs [7]. Basically, these modules are slightly modified extracts of the `BType` and `BindingTimeAnalysis` modules from the SML-mix distribution.

The `Corelang` module implements translation between the abstract JVM instruction defined in the SML-JVM toolkit [3] and the core language considered here (cf. section 2.3). This may be used to experiment with binding-time analysis of Java bytecode methods by first translating into our core language (when possible). Potentially, this could also be used to translate a specialized core language program (back) into Java bytecode for execution by a JVM.

The `Corelang` module also implements various utility functions, e.g. for pretty-printing core language instructions. A number of modules from the SML-JVM toolkit are used in the representation of core language instructions, e.g. `Label` and `Localvar` which define abstract labels and abstract local variable indices.

3.8.2 Phases

During binding-time analysis of a core language program, our implementation performs the following steps:

1. finding the maximum local variable index being used;
2. calculating the stack depth at each label;
3. building the label environment λ (cf. section 3.5);
4. generating constraints and associating variables with each instruction (cf. section 3.5);
5. converting generated constraints to Birkedal and Welinder constraints (cf. section 3.6);
6. solving the constraints by means of Birkedal and Welinder's constraint solver; and
7. generating annotated code, based on the solution to the generated constraints (cf. section 3.7).

The stack depth d_l at each label and the maximum local variable index j_{max} is used in building the label environment λ: for each label l, a stack of fresh binding-time variables of depth d_l is created and a binding-time variable environment is built. The binding-time variable environment maps each local variable index j, where $j_{min} \leq j \leq j_{max}$, to a fresh binding-time variable.

Along with the generation of binding-time constraints, an intermediate version of the core language program is built in which each instruction is associated with the proper number of binding-time variables. This information is exploited in the last step, where each instruction of the core language is to be annotated with a binding-time type and lift instructions must be inserted where required.

3.8.3 Using our implementation

The `Btanalysis` module implements the `bta` function, which is the 'main function' of our implementation of the binding-time analysis. It takes two parameters: an initial local variable binding-time environment ρ_0 and a list cs of core language instructions constituting a core language program. The environment ρ_0 should map the local variable index of each program parameter to the *minimum* binding-time type of that parameter.

The `bta` function returns a pair (ρ, cs'), where ρ is another binding-time environment ρ and cs' is the resulting annotated core language program generated from cs. The resulting annotated program will be well-annotated with respect to the initial local variable binding-time environment ρ, where ρ is (point-wise) larger than or equal to ρ_0:

$$\forall j \in dom(\rho_0).(\rho_0(j) \sqsubseteq \rho(j))$$

The idea is that the binding-time analysis may determine that a program parameter has to be 'more dynamic' than specified in the minimum initial environment ρ_0.

For example, this would be the case if the core language program being analysed takes an array reference as parameter and returns the same array reference as the result of the program. The minimum environment ρ_0 could indicate e.g. that the reference must at least have the binding-time type $\mathsf{D}[\,]$, whereas the environment ρ generated by our implementation of the analysis would state that the array reference parameter must have binding-time type D (since it is to be returned by the residual program).

A user of the binding-time analysis could compare the environment ρ with ρ_0 to decide how the residual program generated by a specializer should be treated. For example, if $\rho_0(j) \neq \rho_0(j)$ for any $j \in dom(\rho_0)$ then this might be considered an error, since a well-annotated program could not be generated with respect to the binding-time types specified by ρ_0.

Alternatively, the user may exploit differences between binding-time types specified by ρ_0 and ρ to decide when to insert a 'wrapper' routine to handle the necessary lifting (if possible).

3.8.4 Availability

The SML source files for our implementation of the binding-time analysis will be available at the following address:

`http://www.dina.kvl.dk/~pmb`

The implementation comprises less than 2,000 lines of SML source code, excluding module signatures and modules from the SML-JVM toolkit [3].

4 Examples

In the following sections we demonstrate our implementation of the binding-time analysis through some selected example programs. For each program we show:

- a Java method corresponding to the sample core language program (to help the reader understand the core language program),
- the instruction sequence of the core language program itself and
- the resulting annotated code generated by our implementation of the binding-time analysis, given a specific initial binding-time environment ρ_0 (cf. section 3.8).

Note that our implementation does *not* work at the level of Java source programs. Moreover, a translation from Java source to core language programs has not been implemented; the core language programs shown in the following examples were constructed by hand.

4.1 Adding two numbers

A very simple routine to add two numbers x and y might be written like this in Java:

```
static int add (int x, int y) {
  return x+y;
}
```

In our core language (cf. section 2) this may be implemented as follows[24]:

load j_0
load j_1
binop add
return

This program expects the two parameters (x and y) to be passed in local variables j_0 and j_1; it loads the parameters, adds them and returns the sum.

Given the initial binding-time environment $\rho_0 = \{j_0 \mapsto \mathsf{D}, j_1 \mapsto \mathsf{D}\}$, in which both program parameters are forced to be *dynamic*, our implementation of the binding-time analysis generates the following annotated code from the above core language program[25]:

```
load_D j_0
load_D j_1
binop_D add
return
```

This is the most simple (and trivial) annotation of the program: all four instructions of the original program are annotated as dynamic, meaning that the entire program will have to be residualized since the specializer cannot evaluate any of the instructions[26].

Using instead the initial binding-time environment $\rho_0 = \{j_0 \mapsto \mathsf{S}, j_1 \mapsto \mathsf{S}\}$, in which both program parameters are indicated to be *static*, our implementation generates the following annotated code from the original core language program shown above:

[24]We use a relaxed and brief notation in examples of core language programs, rather than the more precise and elaborate syntax of the form $c :: cs$ (as in the preceeding sections). For each sample program we simply list the instructions of the program, usually one instruction per line.

[25]In examples of output from our implementation of the binding-time analysis, an instruction mnemonic ending in `_b`, where `b` $\in BT$, means that the instruction is annotated as having binding-time type `b`. Furthermore, local variable indices are shown as `j_i`, where `i` $\in \mathcal{N}_0$. This is not a LaTeX error.

[26]A specializer has not yet been implemented; we describe a potential specializer in more detail in section 5.

```
load_S j_0
load_S j_1
binop_S add
liftTop
return
```

This is the most advantageous annotation of the program: as many instructions as possible may be evaluated by the specializer. Note that a `liftTop` instruction has been inserted before the `return` instruction in order to lift the (static) result of the `binop` instruction to a dynamic context[27].

Based on this annotation, a specializer could perform the addition at specialization time, e.g. producing the following residual (core language) code:

`const` k
`return`

where k would be the sum of the two (known) parameters to the original program. The residual program would not take any parameters.

4.2 Calculating x^n

In this section we demonstrate our binding-time analysis on a slightly larger and rather traditional example from literature on partial evaluation: the *power* function, $power(x, n) = x^n$.

In Java, a routine to calculate x to the n'th power (for $n \geq 0$) might be implemented as follows:

```
static int power (int x, int n) {
  int p = 1;
  while (n != 0) {
    n = n - 1;
    p = p * x;
  }
  return p;
}
```

In our core language, the *power* function may be implemented as follows:

`const` 1
`store` j_2
`goto` l_1
`label` l_0
`load` j_1
`const` 1
`binop` sub
`store` j_1
`load` j_2
`load` j_0
`binop` mul
`store` j_2
`label` l_1
`load` j_1
`const` 0
`if` ne l_0
`load` j_2
`return`

[27]The `return` instruction requires a dynamic operand because a value must be returned from the residual program (cf. section 3.4).

This core language program expects the two parameters (x and n) to be passed in local variables j_0 and j_1, and stores the intermediate result (p) in local variable j_2.

In order to specialize the program with respect to a known value of n, the initial binding-time environment $\rho_0 = \{j_0 \mapsto \mathsf{D}, j_1 \mapsto \mathsf{S}\}$, in which the first parameter (x) is forced to be dynamic and the second parameter (n) is allowed to be static, should be used. Given this, our implementation of the binding-time analysis generates the following annotated code from the above core language program[28]:

```
const 1
store_S j_2
lift j_2
goto l_1
label l_0
load_S j_1
const 1
binop_S sub
store_S j_1
load_D j_2
load_D j_0
binop_D mul
store_D j_2
label l_1
load_S j_1
const 0
if_S ne l_0
load_D j_2
return
```

Note that the `load` and `store` instructions accessing local variable j_1 (n) are annotated as static; moreover, the `if` instruction, which implements the test for $n \neq 0$, is annotated as static. This implies that the loop may be unfolded by a specializer for a given value of n (as expected). We illustrate this by an example in section 5.3.1.

Consider instead the case where we wish to specialize the *power* function with respect to a known value of x. To this end, the initial binding-time environment $\rho_0 = \{j_0 \mapsto \mathsf{S}, j_1 \mapsto \mathsf{D}\}$, in which the first parameter (x) is allowed to be static and the second parameter (n) is forced to be dynamic, should be used. Given this, our implementation of the binding-time analysis generates the following annotated code from the *power* core language program shown above:

```
const 1
store_S j_2
goto l_1
label l_0
load_D j_1
const 1
liftTop
binop_D sub
store_D j_1
load_S j_2
load_S j_0
binop_S mul
store_S j_2
label l_1
load_D j_1
```

[28]In the output from our implementation of the binding-time analysis, labels are shown as `l_i`, where `i` $\in \mathcal{N}_0$. This is not a LaTeX error.

```
const 0
liftTop
if_D ne l_0
load_S j_2
liftTop
return
```

Note that the `if` instruction, which implements the test for $n \neq 0$, is now annotated as dynamic. Similarly, the `load` and `store` instructions accessing local variable j_1 (n) are annotated as dynamic.

However, the `load` and `store` instructions accessing local variable j_0 (x) and j_2 (p), as well as the `binop` instruction, are annotated as static. This may lead to infinite specialization: a specializer that naïvely follows the annotations produced by our binding-time analysis may attempt to evaluate these static instructions under dynamic control over and over again *ad infinitum*. We describe this problem in more detail in section 5.4.

4.3 Euclid's *gcd* algorithm

One of the harder problems in partial evaluation is achieving good specialization of Euclid's algorithm to calculate the greatest common divisor (*gcd*) of two natural numbers.

Euclid's *gcd* algorithm may be implemented as follows in Java:

```
static int gcd (int a, int b) {
  while (b != 0) {
    int tmp = a;
    a = b;
    b = tmp % b;
  }
  return a;
}
```

In our core language, the *gcd* algorithm may be implemented as follows:

goto l_1
label l_0
load j_1
load j_0
swap
store j_0
load j_1
binop rem
store j_1
label l_1
load j_1
const 0
if eq l_0
load j_0
return

This program expects the two parameters (a and b) to be passed in local variables j_0 and j_1. However, unlike the Java method `gcd` shown above, it does not use an extra temporary variable in the body of the loop; instead, the operand stack is exploited to hold the 'old' value of local variable j_0 (a) while storing the value of local variable j_1 (b) into it. This usage of the operand stack cannot be expressed directly in Java.

Given the initial binding-time environment $\rho_0 = \{j_0 \mapsto \mathsf{S}, j_1 \mapsto \mathsf{D}\}$, in which the first parameter (a) is allowed to be static and the second parameter (b) is forced to be dynamic, our implementation of the binding-time analysis generates the following annotated code from the above core language program:

```
lift j_0
goto l_1
label l_0
load_D j_1
load_D j_0
swap_D
store_D j_0
load_D j_1
binop_D rem
store_D j_1
label l_1
load_D j_1
const 0
liftTop
if_D eq l_0
load_D j_0
return
```

None of the instructions are annotated as static and the static value of local variable j_0 (a) is lifted at the very beginning of the resulting annotated program. This implies that a specializer cannot perform any interesting evaluation of the program: all instructions of the original program must be residualized.

The reason that all instructions from the original *gcd* program become dynamic in the annotated program is that the dynamic value of local variable j_1 (b) is stored into local variable j_0 (a). Hence, local variable j_0 is forced to be dynamic after the `store` j_0 instruction; and since the flow of control may return to label l_0 in case the loop test succeeds, local variable j_0 also has to be dynamic from label l_0 and onwards. Finally, since the flow of control goes directly to label l_1 after the program starts executing, and since local variable j_0 is initially static, local variable j_0 must be lifted before branching to label l_1.

A very similar annotated program is generated from the above core language program if the initial binding-time environment $\rho_0 = \{j_0 \mapsto \mathsf{D}, j_1 \mapsto \mathsf{S}\}$ is used instead:

```
lift j_1
goto l_1
label l_0
load_D j_1
load_D j_0
swap_D
store_D j_0
load_D j_1
binop_D rem
store_D j_1
label l_1
load_D j_1
const 0
liftTop
if_D eq l_0
load_D j_0
return
```

The only difference from the first annotated version of the *gcd* program shown above is that the static value of local variable j_1 is lifted at the beginning of the program.

As before, the reason for all original instructions being annotated as dynamic is that the dynamic value of local variable j_0 (a) is used in a calculation, the result of which is stored into local variable j_1 (b).

All in all, our binding-time analysis cannot help in achieving any non-trivial specialization of Euclid's *gcd* algorithm. The above example mainly goes to show that our implementation of the binding-time analysis behaves as expected.

4.4 A *NORMA2* interpreter

In this section we present an even larger and more complex example: an interpreter for an abstract machine.

The abstract machine *NORMA2* is a simple register machine [6] with only two registers, X and Y. The instruction set of NORMA2 may be encoded as follows:

Opcode	Meaning
0	$X := X + 1$
1	$Y := Y + 1$
2	$X := X - 1$
3	$Y := Y - 1$
4	if $X = 0$ goto l
5	if $Y = 0$ goto l
6	goto l

where the branch target l of instruction 4, 5 and 6 is encoded as an immediate operand. In other words, a NORMA2 program may be represented as a vector of integers, where the value immediately following an instruction with opcode 4, 5 or 6 is the index of the branch target.

A simple interpreter for the NORMA2 instruction set shown above may implemented as follows in Java:

```
static int norma2 (int[] P, int X) {
  int Y = 0;
  int pc = 0;   // program counter
  while (pc >= 0 && pc < P.length) {
    switch (P[pc++]) {
    case 0: X = X + 1;  continue;
    case 1: Y = Y + 1;  continue;
    case 2: X = X - 1;  continue;
    case 3: Y = Y - 1;  continue;
    case 4: if (X == 0) pc = P[pc];
            else pc++;   // skip past immediate argument (branch target)
            continue;
    case 5: if (Y == 0) pc = P[pc];
            else pc++;   // skip past immediate argument (branch target)
            continue;
    default: pc = P[pc];
    }
  }
  return Y;
}
```

The `norma2` method takes two parameters: the NORMA2 program P to be executed and the initial value of register X. Register Y always has the inital value 0. The interpreter stops when an attempt is made to branch to an index outside the program P; it then returns the resulting value of register Y.

For the sake of simplicity, it is assumed that the program P is a valid NORMA2 program. Specifically, the interpreter does not check that the current opcode is valid—all opcodes not in the range $[0;5]$ are treated as a NORMA2 goto instruction—and it is not checked whether a branch target within the bounds of P actually refers to a valid instruction index.

In our core language, the NORMA2 interpreter may be implemented as follows[29]:

```
const 0          label l1         label l3         label l5
store j2         dup              dup              const 5
const 0          const 1          const 3          if ne l6
store j3         if ne l2         if ne l4         load j2
............     pop              pop              const 0
label l0         load j2          load j2          if eq l6
load j3          const 1          const 1          load j3
const 0          binop add        binop sub        const 1
if lt l7         store j2         store j2         binop add
load j3          goto l0          goto l0          store j3
load j0          ..........       ..........       goto l0
arraylength      label l2         label l4         ..........
if ge l7         dup              dup              label l6
............     const 2          const 4          load j0
load j0          if ne l3         if ne l5         load j3
load j3          pop              pop              arrayload
dup              load j1          load j1          store j3
const 1          const 1          const 0          goto l0
binop add        binop sub        if eq l6         ..........
store j3         store j1         load j3          label l7
arrayload        goto l0          const 1          load j2
............                      binop add        return
dup                               store j3
const 0                           goto l0
if ne l1
pop
load j1
const 1
binop add
store j1
goto l0
```

This core language program expects the two parameters (P and X) to be passed in local variable j_0 and j_1. The value of register Y is stored in local variable j_2 and the NORMA2 program counter (pc) is stored in local variable j_3.

The second block, starting at label l_0, corresponds to the beginning of the `while` statement in the `norma2` method above. The third block, starting with a `load` j_0 instruction, corresponds to the beginning of the `switch` statement. The fourth block, starting with a `dup` instruction, corresponds to the first `case` entry for opcode 0. Similarly, the blocks starting at label l_1, l_2, l_3, l_4 and l_5 correspond to the `case` entries for opcode 1, 2, 3, 4 and 5, respectively. The block starting at label l_6 corresponds to the `default` case[30] and the block starting at label l_7 corresponds to the `return` statement in the `norma2` method.

Note that the `switch` statement of the Java method `norma2` is represented as a series of 'cascaded' `if` instructions in the above core language program[31]. From one `if` instruction to the next, the opcode is held on the operand stack by (repeatedly) duplicating it.

Given the initial binding-time environment $\rho_0 = \{j_0 \mapsto \mathsf{S}[\,], j_1 \mapsto \mathsf{D}\}$, in which the first parameter (P) is allowed to be (a reference to) an array of static integers and the second parameter (X)

[29] The instruction sequence has been broken into four columns to make the program more legible. The dotted lines have been inserted only to indicate the structure of the program; they are not present in the program.

[30] The code in the block starting at label l_6 is 'shared' by the blocks starting at label l_4 and l_5: instead of duplicating the code for implementing `pc = P[pc]`, these blocks just branch to label l_6.

[31] The actual JVM instruction set [13] has special 'compound branch' instructions for implementing Java's `switch` statements: `lookupswitch` and `tableswitch`. Our core language does not include these instructions.

is forced to be dynamic, our implementation of the binding-time analysis generates the following annotated code from the above core language program[32]:

```
const 0
store_S j_2
const 0
store_S j_3
label l_0
load_S j_3
const 0
if_S lt l_7
load_S j_3
load_S j_0
arraylength_S
if_S ge l_7
load_S j_0
load_S j_3
dup_S
const 1
binop_S add
store_S j_3
arrayload_S[]
dup_S
const 0
if_S ne l_1
pop_S
load_D j_1
const 1
liftTop
binop_D add
store_D j_1
goto l_0
label l_1
dup_S
const 1
if_S ne l_2
pop_S
load_S j_2
const 1
binop_S add
store_S j_2
goto l_0
label l_2
dup_S
const 2
if_S ne l_3
pop_S
load_D j_1
const 1
liftTop
binop_D sub
store_D j_1
goto l_0
label l_3
dup_S
const 3
if_S ne l_4
pop_S
load_S j_2
const 1
binop_S sub
store_S j_2
goto l_0
label l_4
dup_S
const 4
if_S ne l_5
pop_S
load_D j_1
const 0
liftTop
if_D eq l_6
load_S j_3
const 1
binop_S add
store_S j_3
goto l_0
label l_5
const 5
if_S ne l_6
load_S j_2
const 0
if_S eq l_6
load_S j_3
const 1
binop_S add
store_S j_3
goto l_0
label l_6
load_S j_0
load_S j_3
arrayload_S[]
store_S j_3
goto l_0
label l_7
load_S j_2
liftTop
return
```

Note that most of the instructions are annotated as static. In particular, all `load` and `store` instructions accessing local variable j_2 (Y), all `load` and `store` instructions accessing local variable j_3 (pc) and all `if` instructions testing the value of the program counter are annotated as static. Only the `load` and `store` instructions accessing local variable j_1 (X), the `binop` instructions calculating a value based on the value of register X and the `if` instruction implementing the test for $X = 0$ are annotated as dynamic.

This implies that a specializer may partially evaluate the interpreter for a given NORMA2 program such that the layer of interpretation is eliminated in the residual program (as expected). We illustrate this by an example in section 5.3.3.

As with the *power* function (cf. section 4.2), there is the potential problem that the instructions following the dynamic `if` instruction (`load` j_3, `binop add` and `store` j_3) are annotated as static. A specializer could happen to loop infinitely, trying to evaluate these static instructions under dynamic control over and over again. We return to a discussion of this problem in section 5.4.

[32] As for the original core language program, the annotated instruction sequence has been broken into four columns to make the program more legible.

5 Specialization

In the following sections we describe the flavour of partial evaluation that our binding-time analysis is meant to support. A specializer for our core language has not yet been implemented. Instead, we outline a potential approach to off-line specialization of programs written in the annotated core language of section 3.

5.1 Approach

An annotated core language program is viewed as a sequence of 'basic blocks' comprising the initial block followed by a number of labelled blocks. Each labelled block starts with a `label` l instruction. The last instruction in each block is a `goto` or `return` instruction.

A partial evaluator for our annotated core language should perform *basic block specialization*[33], whilst emulating the operand stack, local variable environment and heap of the program being specialized. For each basic block of the annotated core language program, zero, one or more specialized versions of the block should be produced. Each block is to be specialized with respect to the *configuration* at the beginning of the block (see below). The result of program specialization is the concatenation of all specialized program blocks.

In order to keep track of static and dynamic values during specialization, the specializer should operate with an extended set of values Val_{spec}, defined as follows:

$$Val_{spec} = Val \cup \{\mathsf{dyn}\}$$

where dyn is a special value used to represent dynamic values during specialization.

The specializer should also use a corresponding operand stack s, local variable environment e and heap h:

$$\begin{aligned} s &: Stack_{spec} \\ e &: Locals_{spec} \\ h &: Heap_{spec} \end{aligned}$$

where

$$\begin{aligned} Stack_{spec} &= Val^*_{spec} \\ Locals_{spec} &= Locals \\ Heap_{spec} &= Ref \stackrel{\text{fin}}{\to} Comp^* \\ Comp &= Val \cup Index \end{aligned}$$

The type of $Locals_{spec}$of specialization-time local variable environments is the same as the type *Locals* used for ordinary local variable environments (cf. section 2.1). This is sufficient since we need not keep track of the value of dynamic local variables.

The specialization-time heap h, which is of type $Heap_{spec}$, maps an array reference r to an array (vector) of components. The array components, which are now of type *Comp*, are either integers, references or local variable indices.

Local variable indices are used as array components for 'unfolding' a partially static array into local variables in the residual program. A partially static array with components of binding-time D are represented at specialization-time as a vector of local variable indices. Each component of the vector holds the index of the local variable in the residual program that the dynamic value of the array component will be stored in.

As mentioned above, each basic block is to be specialized with respect to a specific *configuration* (s, e, h). The result of specializing a labelled basic block l with respect to the configuration (s, e, h) is thus a residual program block labelled $l(s, e, h)$, where $l(s, e, h)$ is the label l specialized with respect to (s, e, h).

During specialization of a core language program, the specializer should maintain a *pending set*, that is, a set of blocks awaiting specialization with respect to specific configuations.

[33]This corresponds to *program point specialization* as described in [12].

5.2 Specializing annotated instructions

In general, an instruction annoated as dynamic must be residualized[34], whereas an instruction annotated as non-dynamic must be (partially) evaluated by the specializer. In each of the following sections, we describe in greater detail how partial evaluation should work for specific 'families' of instructions.

5.2.1 Simple instructions

The following instructions of the annotated core language are considered 'simple' in connection with specialization:

$$\begin{array}{l}\texttt{binop}_b\ \odot\\ \texttt{const}\ k\\ \texttt{dup}_b\\ \texttt{load}_b\ j\\ \texttt{pop}_b\\ \texttt{store}_b\ j\\ \texttt{swap}_b\end{array}$$

A simple instruction annotated as b, where $b \neq \mathsf{D}$, is partially evaluated simply by emulating the instruction's effect on the specialization-time operand stack s, local variable environment e and heap h.

For example, the instruction $\texttt{binop}_\mathsf{S}$ `add` is specialized by popping the two topmost stack operands (assumed to be integers) and pushing their sum onto the operand stack. In other words, partial evaluation of the $\texttt{binop}_\mathsf{S}$ `add` instruction changes the specialization-time operand stack from $k_1 :: k_2 :: s$ to $k_3 :: s$, where $k_1 + k_2 = k_3$, but leaves the local variable environment e and heap h unchanged.

As another example, the instruction $\texttt{store}_\mathsf{S}\ j$ is specialized by popping the topmost stack operand and storing it into local variable j. In other words, partial evaluation of the $\texttt{store}_\mathsf{S}\ j$ instruction changes the specialization-time operand stack from $v :: s$ to s and changes the local variable environment from e to $e[j \mapsto v]$, but leaves the heap h unchanged.

A `const` k instruction is always evaluated at specialization-time; it is specialized by pushing the integer constant k onto the specialization-time operand stack.

The specializer should always residualize a simple instruction annotated as D; in connection with this the specialization-time operand stack changes as follows:

Instruction	*Stack before*	*Stack after*
$\texttt{binop}_\mathsf{D}\ \odot$	$\mathsf{dyn} :: \mathsf{dyn} :: s$	$\mathsf{dyn} :: s$
$\texttt{dup}_\mathsf{D}$	$\mathsf{dyn} :: s$	$\mathsf{dyn} :: \mathsf{dyn} :: s$
$\texttt{load}_\mathsf{D}\ j$	s	$\mathsf{dyn} :: s$
$\texttt{pop}_\mathsf{D}$	$\mathsf{dyn} :: s$	s
$\texttt{store}_\mathsf{D}\ j$	$\mathsf{dyn} :: s$	s
$\texttt{swap}_\mathsf{D}$	$\mathsf{dyn} :: \mathsf{dyn} :: s$	$\mathsf{dyn} :: \mathsf{dyn} :: s$

Specialization of a simple instruction annotated as D does not change the specialization-time local variable environment e or heap h.

After specialization of a simple instruction, the specializer continues with the following instruction.

[34]An (annotated) core language instruction is *residualized* by generating a corresponding instruction (without any binding-time annotation) in the residual program.

5.2.2 Array instructions

An array instruction annotated as b, where $b \neq \mathsf{D}$, should be specialized as follows:

- An $\texttt{arraylength}_b$ instruction, where $b \neq \mathsf{D}$, is specialized by popping the topmost stack operand, assumed to be a non-dynamic reference to an array $a = h(r)$ in the heap, and pushing the length of that array onto the stack.

- When specializing an $\texttt{arrayload}_b$ instruction, where $b = b'[\,]$, the two topmost stack operands i and r are first popped off the specialization-time stack. The operand r is assumed to be a reference to an array $a = h(r)$ in the heap and i is assumed to be a valid index into that array. Depending on the binding-time type b, a value representing the array component $v = a(i)$ at index i is then pushed onto the stack:

  - if $b = \mathsf{D}[\,]$ then v is assumed to be the index of the local variable in the residual program that is to be used for holding the value of the i'th array component; a residual instruction `load` v is generated and the special value `dyn` is pushed onto the specialization-time operand stack, representing the dynamic value that will be loaded by the residual instruction;

  - otherwise the value v itself, which is assumed to be an integer or a reference into the specialization-time heap, is pushed onto the stack.

- When specializing an $\texttt{arraystore}_b$ instruction, where $b = b'[\,]$, the three topmost stack operands v, i and r are first popped off the specialization-time operand stack. The operand r is assumed to be a reference to an array $a = h(r)$ in the heap and i is assumed to be a valid index into that array. Depending on the binding-time type b, either residual code is generated to store the dynamic value corresponding to v into a local variable or else the value v is stored into the array a in the specialization-time heap:

  - if $b = \mathsf{D}[\,]$ then a residual instruction `store` j is generated, where $j = a(i)$ is the value of the array component at index i, assumed to be the index of the local variable in the residual program that is to be used for holding the value of the i'th array component;

  - otherwise, the value v is stored into component number i in the array a; in other words, the specialization time heap is changed from h to $h[r \mapsto a']$, where $a' = a[i \mapsto v]$.

- A $\texttt{newarray}_b$ d instruction, where $b = b'[\,]$, is specialized by popping the topmost stack operand n, assumed to be a non-negative integer, and pushing a reference r to a new array a of length n onto the specialization-time stack. The initial values of the components of the newly created array depend on the binding-time type b:

  - if $b = \mathsf{D}[\,]$ then each component of the new array a will be the index j_i of an otherwise unused local variable in the residual program; for each index i of the new array, the following residual code is generated to initialize the corresponding local variable j_i in the residual program that is to be used for holding the value of the i'th array component:

    $$\texttt{const}\ 0$$
    $$\texttt{store}\ j_i$$

  - otherwise, each component of a will be initialized to 0 if $d = 0$ or to `null` if $d > 0$.

  The reference r will be a fresh reference not already in the domain of the specialization-time heap. The heap changes from h to $h[r \mapsto a]$.

An array instruction annotated as D should always be residualized; in connection with this the specialization-time stack changes as follows:

Instruction	*Stack before*	*Stack after*
$\texttt{arraylength}_b$	$\mathsf{dyn} :: s$	$\mathsf{dyn} :: s$
$\texttt{arrayload}_b$	$\mathsf{dyn} :: \mathsf{dyn} :: s$	$\mathsf{dyn} :: s$
$\texttt{arraystore}_b$	$\mathsf{dyn} :: \mathsf{dyn} :: \mathsf{dyn} :: s$	s
$\texttt{newarray}_b\ d$	$\mathsf{dyn} :: s$	$\mathsf{dyn} :: s$

Specialization of an array instruction annotated as D does not change the specialization-time local variable environment e or heap h.

After specialization of an array instruction, the specializer continues with the following instruction.

5.2.3 Control-flow instructions

The control-flow instructions of the annotated core language should be specialized as follows:

- Specialization of a `goto` l instruction depends on the specialization that has been performed so far: if the block labelled l has not yet been specialized with respect to the current configuration, then specialization continues with that block using the current configuration[35].

  Otherwise, a residual `goto` to the residual label $l(s, e, h)$ is generated, where (s, e, h) is the current configuration; if any blocks are pending specialization, the specializer continues with a pending block, using the corresponding configuration; otherwise specialization terminates.

- An $\texttt{if}_b\ r\ l$ instruction, where $b \neq \mathsf{D}$, is specialized by popping the two topmost stack operands and comparing them. If the relation r is satisfied (cf. section 2.2) then specialization proceeds as for a `goto` instruction with the same branch target (see above). Otherwise, specialization continues with the next instruction. This way, only one branch of a static conditional will be specialized.

- When specializing an $\texttt{if}_\mathsf{D}\ r\ l$ instruction, the two topmost stack operands are first popped off the specialization-time stack. A residual `if` $r\ l(s', e, h)$ instruction is then generated, where s' is the remaining stack, e is the current local variable environment and h is the current heap. If the block labelled l has not yet been specialized with respect to the configuration (s', e, h), then it is recorded that the block labelled l is pending specialization with respect to (s', e, h). Specialization continues with the instruction immediately following the `if` instruction. This way, both branches of a dynamic conditional will be specialized.

- A `label` l instruction is specialized by generating a residual `label` $l(s, e, h)$ instruction, where (s, e, h) is the current configuration, and recording that the block labelled l has been specialized with respect to the configuration (s, e, h). Specialization continues with the next instruction using the current configuration.

- A `return` instruction is specialized by generating a residual `return` instruction. If any blocks are pending specialization, the specializer continues with a pending block, using the corresponding configuration; otherwise specialization terminates.

Generating a residual `label` instruction every time a basic block in the annotated program is specialized with respect to a different configuration will generally lead to a number of superfluous `label` instructions in the residual program. These may be removed in a post-processing pass over the generated residual program.

After specialization is complete, generated labels may also be renamed to simple label names l'_0, l'_1, etc.

[35] This corresponds to *on-the-fly transition compression* [12]; the net effect is that we avoid generating a residual `goto` l instruction immediately followed by the instruction `label` l representing the label l.

5.2.4 Lift instructions

The lift instructions of the annotated core language should be specialized as follows:

- A `liftTop` instruction residualizes to a `const` k instruction, where k is the topmost stack operand (assumed to be an integer). Before continuing with specialization of the next instruction, the specialization-time operand stack is changed from $k :: s$ to $\mathsf{dyn} :: s$, where the special value dyn represents the newly lifted stack value.

- Specialization of a `lift2nd` instruction depends on the topmost value of the specialization-time operand stack: if the topmost operand is dyn then the residual code

  ```
  const k
  swap
  ```

  is generated, where k is the second topmost stack operand (assumed to be an integer); otherwise, the `lift2nd` instruction simply residualizes to a `const` k instruction. In the former case, the `swap` instruction ensures that the lifted value is inserted 'below' the topmost operand in the stack of the residual program. In the latter case, the second topmost operand of the specialization-time stack is in fact supposed to appear as the topmost operand in the stack of the residual program.

  Before continuing with specialization of the next instruction, the specialization-time operand stack is changed from $v :: k :: s$ to $v :: \mathsf{dyn} :: s$.

- A `lift` j instruction residualizes to the following code:

  ```
  const k
  store j
  ```

  where $k = e(j)$ is the value of local variable j (assumed to be an integer). Specialization continues with the next instruction, using the same configuration as before[36].

5.2.5 Errors during specialization

Since the specializer evaluates the non-dynamic instructions of an annotated core language program, an error that would otherwise occur during normal evaluation of the original core language program may occur during specialization. More specifically,

- specialization of an $\texttt{arraylength}_b$ instruction, where $b \neq \mathsf{D}$, fails in case the topmost stack operand is `null`;

- specialization of an $\texttt{arrayload}_b$ instruction, where $b \neq \mathsf{D}$, fails in case the second topmost stack operand (the array reference) is `null` or in case the topmost stack operand (the index) is not within the bounds of the referenced array;

- specialization of an $\texttt{arraystore}_b$ instruction, where $b \neq \mathsf{D}$, fails in case the third topmost stack operand (the array reference) is `null` or in case the second topmost stack operand (the index) is not within the bounds of the referenced array;

- specialization of a $\texttt{newarray}_b$ d instruction, where $b \neq \mathsf{D}$, fails in case the topmost stack operand (the length of the array to be created) is negative;

- specialization of a $\texttt{binop}_\mathsf{S}$ div or $\texttt{binop}_\mathsf{S}$ rem instruction fails if the topmost stack operand (the divisor) is zero.

When specialization of a non-dynamic instruction fails, specialization of the entire annotated core language program fails.

[36]We do not need to record that a local variable j is being lifted: the annotations determined by the binding-time analysis ensure that we do not attempt to load the (dynamic) value of local variable j later on.

5.3 Examples

In this section we show some examples of program specialization guided by the binding-time annotations determined by our binding-time analysis. The examples were specialized by hand.

5.3.1 Calculating x^3

The *power* program shown in section 4.2 may be specialized with respect to a known value of n based on the annotated program generated by our implementation of the binding-time analysis:

```
const 1
store_S j_2
lift j_2
goto l_1
label l_0
load_S j_1
const 1
binop_S sub
store_S j_1
load_D j_2
load_D j_0
binop_D mul
store_D j_2
label l_1
load_S j_1
const 0
if_S ne l_0
load_D j_2
return
```

The annotated *power* program shown here is the same as the first annotated version of the program shown in section 4.2, corresponding to the initial environment $\rho_0 = \{j_0 \mapsto \mathsf{D}, j_1 \mapsto \mathsf{S}\}$.

The program may be specialized for $n = 3$ as follows:

Block	*Stack*	*Environment*	*Residual code*
(initial)	*nil*	$\{j_1 \mapsto 3\}$	`const 1` `store` j_2
l_1	*nil*	$\{j_1 \mapsto 3\}$	`label` $l_1(nil, \{j_1 \mapsto 3\}, h)$
l_0	*nil*	$\{j_1 \mapsto 3\}$	`label` $l_0(nil, \{j_1 \mapsto 3\}, h)$ `load` j_2 `load` j_0 `binop mul` `store` j_2
l_1	*nil*	$\{j_1 \mapsto 2\}$	`label` $l_1(nil, \{j_1 \mapsto 2\}, h)$
l_0	*nil*	$\{j_1 \mapsto 2\}$	`label` $l_0(nil, \{j_1 \mapsto 2\}, h)$ `load` j_2 `load` j_0 `binop mul` `store` j_2
l_1	*nil*	$\{j_1 \mapsto 1\}$	`label` $l_1(nil, \{j_1 \mapsto 1\}, h)$
l_0	*nil*	$\{j_1 \mapsto 1\}$	`label` $l_0(nil, \{j_1 \mapsto 1\}, h)$ `load` j_2 `load` j_0 `binop mul` `store` j_2

l_1	nil	$\{j_1 \mapsto 0\}$	`label` $l_1(nil, \{j_1 \mapsto 0\}, h)$ `load` j_2 `return`

The first column indicates the block being specialized, starting with the initial block of the program (comprising the instructions up to the first `goto` instruction); the second column shows the initial contents of the emulated operand stack at the beginning of each block; the third column shows the static local variable environment at the beginning of each block and the last column shows the residual instructions generated for each block.

The residual code for the initial block is generated in connection with specialization of the `lift` j_2 instruction of the annotated program.

From the first `goto` instruction, specialization follows the control-flow to the block labelled l_1. The only code generated for this block is a `label` instruction. The static test for $n = 0$ succeeds and specialization follows the control-flow to the block labelled l_0.

The dynamic instructions in the block labelled l_0 are residualized and we fall through to the block labelled l_1, but this time with a different configuration since local variable j_1 (n) has been decreased by one. Again, the static test for $n = 0$ succeeds, specialization follows the control-flow to the block labelled l_0, and so forth (until $n = 0$).

When the value of local variable j_1 (n) has been decreased to 0, the static test for $n = 0$ fails; specialization continues with the instructions immediately following the `if` instruction and terminates when the `return` instruction has been residualized.

Note that a `label` $l_i(s, e, h)$ instruction is generated for each labelled block, corresponding to the original label l_i of the block specialized with respect to the stack s, environment e and heap h.

After removing superfluous labels we get the following residual code:

```
const 1
store j2
load j2
load j0
binop mul
store j2
load j2
load j0
binop mul
store j2
load j2
load j0
binop mul
store j2
load j2
return
```

This residual program actually calculates the third power of its parameter (being passed in local variable j_0). Note that the instruction sequence

```
  ⋮
load j2
load j0
binop mul
store j2
  ⋮
```

appears three times in the above residual code, corresponding to the body of the loop in the original program being performed three times (since $n = 3$).

A post-processing optimization phase could detect that storing into and immediately loading from local variable j_2 is superfluous; hence, the above residual program could be 'compressed' to the following program:

```
const 1
load j₀
binop mul
load j₀
binop mul
load j₀
binop mul
return
```

This program exploits the operand stack to hold the intermediate results, e.g. between multiplications.

5.3.2 Specializing an infinite static loop

A potential problem in connection with off-line specialization is a loop in the program to be specialized, depending only on static information. The following very short core language program implements an infinite static loop:

```
label l₀
goto l₀
```

The 'annotated' program generated by our binding-time analysis from the above core language program will be exactly identical to the original program, regardless of the initial environment ρ_0 being used, since neither of the two instructions are to be annotated with a binding-time type.

The annotated program may be specialized as follows:

Block	*Residual code*
l_0	`label` $l_0(nil, \emptyset, h)$ `goto` $l_0(nil, \emptyset, h)$

Specialization starts with generating the residual label $l_0(nil, \emptyset, h)$ and recording that the block labelled l_0 has been specialized with respect to the configuration $(nil, \emptyset, h)$. The `goto` instruction is then residualized, because the target block has already been specialized with respect to the current configuration, and specialization terminates.

Hence, the residual program will be very similar to the original program (after label renaming):

```
label l₀'
goto l₀'
```

This demonstrates that our approach to program specialization handles a static loop by generating residual code that implements the loop. In other words, the specializer itself will not loop infinitely for a static loop in the program being specialized.

5.3.3 Interpreting a NORMA2 program

The vector $[4, 4, 6, -1, 1]$ encodes the following NORMA2 program (cf. section 4.4):

Index	*Opcode*	*Operand*	*Meaning*
0	4	4	if $X = 0$ goto 4
2	6	-1	goto -1 (stop)
4	1		$Y := Y + 1$

This NORMA2 program returns 1 if the parameter (X) is 0 and returns 0 otherwise.

The NORMA2 interpreter shown in section 4.4 may be specialized with respect to the program $P = [4, 4, 6, -1, 1]$ as follows:

Block	*Residual code*	*Pending*
(initial)	(none)	(none)
l_0	`label` $l_0(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 0\}, h)$	do.
l_1	`label` $l_1(4 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$	do.
l_2	`label` $l_2(4 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$	do.
l_3	`label` $l_3(4 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$	do.
l_4	`label` $l_4(4 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$ `load` j_1 `const` 0 `if eq` $l_6(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$	$l_6(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$
l_0	`label` $l_0(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 2\}, h)$	do.
l_1	`label` $l_1(6 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 3\}, h)$	do.
l_2	`label` $l_2(6 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 3\}, h)$	do.
l_3	`label` $l_3(6 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 3\}, h)$	do.
l_4	`label` $l_4(6 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 3\}, h)$	do.
l_5	`label` $l_5(6 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 3\}, h)$	do.
l_6	`label` $l_6(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 3\}, h)$	do.
l_0	`label` $l_0(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto -1\}, h)$	do.
l_7	`label` $l_7(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto -1\}, h)$ `const` 0 `return`	do.
l_6	`label` $l_6(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$	(none)
l_0	`label` $l_0(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 4\}, h)$	do.
l_1	`label` $l_1(1 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 5\}, h)$	do.
l_0	`label` $l_0(nil, \{\ldots, j_2 \mapsto 1, j_3 \mapsto 5\}, h)$	do.
l_7	`label` $l_7(nil, \{\ldots, j_2 \mapsto 1, j_3 \mapsto 5\}, h)$ `const` 1 `return`	do.

We do not show the initial stack and local variable environment for each block in separate columns here; instead, these may be found in the residual labels for each block. The last column shows the pending blocks after each block has been specialized (along with the configuration that the pending block should be specialized with respect to).

During specialization the value of local variable j_0 (P) is unchanged; this is what is indicated by the ellipsis in each environment specified above. Furthermore, the heap holds the program array P, but this is not shown since it does not change during specialization of the interpreter.

When the annotated instruction $\texttt{if}_\texttt{D}$ `eq` l_6 in block l_4 is specialized, it is recorded that the block corresponding to label $l_6(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$ must also be specialized, i.e., that it is pending specialization; specialization then continues with the instructions following the dynamic `if` instruction.

After the `return` instruction in block l_7 has been residualized, specialization continues with the pending block $l_6(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$. When the `return` in block l_7 has been residualized for the second time, the set of pending blocks is empty and specialization stops.

As before the residual code includes a number of superfluous `label` instructions. After removing these and renaming the only label which is actually necessary, we get the following residual code:

```
load j_1
const 0
if eq l'_6
const 0
return
label l'_6
const 1
return
```

This demonstrates that specialization guided by the binding-time types determined by our binding-time analysis may remove the layer of interpretation in the NORMA2 interpreter for a given NORMA2 program. As we shall see in section 5.4.2, this does not hold for any valid NORMA2 program.

5.4 Infinite specialization

In certain situations, a program specializer that naïvely follows the annotations determined by our binding-time analysis will never terminate, as it attempts to perform infinite specialization of the program.

In this section we give some examples of such situations and discuss remedies to avoid infinite specialization.

5.4.1 Calculating 5^n

Our implementation of the binding-time analysis generates the following annotated program from the *power* program shown in section 4.2 using the initial environment $\rho_0 = \{j_0 \mapsto \mathsf{S}, j_1 \mapsto \mathsf{D}\}$[37]:

```
const 1
store_S j_2
goto l_1
label l_0
load_D j_1
const 1
liftTop
binop_D sub
store_D j_1
load_S j_2
load_S j_0
binop_S mul
store_S j_2
label l_1
load_D j_1
const 0
liftTop
if_D ne l_0
load_S j_2
liftTop
return
```

Specialization of the *power* program for $x = 5$ would proceed as follows:

[37]The annotated *power* program shown here is the same as the second annotated version of the program shown in section 4.2.

Block	*Residual code*	*Pending*
(initial)	(none)	(none)
l_1	`label` $l_1(nil, \{j_0 \mapsto 5, j_2 \mapsto 1\}, h)$ `load` j_1 `const` 0 `if ne` $l_0(nil, \{j_0 \mapsto 5, j_2 \mapsto 1\}, h)$ `const` 1 `return`	$l_0(nil, \{j_0 \mapsto 5, j_2 \mapsto 1\}, h)$
l_0	`label` $l_0(nil, \{j_0 \mapsto 5, j_2 \mapsto 1\}, h)$ `load` j_1 `const` 1 `binop sub` `store` j_1	(none)
l_1	`label` $l_1(nil, \{j_0 \mapsto 5, j_2 \mapsto 5\}, h)$ `load` j_1 `const` 0 `if ne` $l_0(nil, \{j_0 \mapsto 5, j_2 \mapsto 5\}, h)$ `const` 5 `return`	$l_0(nil, \{j_0 \mapsto 5, j_2 \mapsto 5\}, h)$
l_0	`label` $l_0(nil, \{j_0 \mapsto 5, j_2 \mapsto 5\}, h)$ `load` j_1 `const` 1 `binop sub` `store` j_1	(none)
l_1	`label` $l_1(nil, \{j_0 \mapsto 5, j_2 \mapsto 25\}, h)$ $\vdots$	$l_0(nil, \{j_0 \mapsto 5, j_2 \mapsto 25\}, h)$

Specialization would continue infinitely along the same pattern for larger and larger values of j_2, corresponding to 5^0, 5^1, 5^2, etc.

Every time the block l_0 is specialized with respect to a given configuration (s, e, h), the local variable environment e is updated by storing a new (static) value into j_2. Consequently, the following block labelled l_1 is specialized with respect to (s, e', h), where e' is the updated environment. In connection with specialization of the dynamic conditional in block l_1, the block l_0 is thus recorded as pending specialization with respect to the configuration (s, e', h). The pending block l_0 is then specialized with respect to (s, e', h), and so forth. This pattern is repeated over and over again for different values of local variable j_2.

The problem is that the `load` and `store` instructions accessing local variable j_2 in the block labelled l_0 are annotated as static. This leads to infinite specialization of static instructions under dynamic control, a well-known problem in partial evaluation [12].

The problem with infinite specialization of the *power* program might be 'solved' by reclassifying the program parameter j_1 (x) as *dynamic*[38]. Given the initial environment $\rho_0 = \{j_0 \mapsto \mathsf{D}, j_1 \mapsto \mathsf{D}\}$ our implementation of the binding-time analysis generates the following annotated core language program from the *power* program:

```
const 1
store_S j_2
lift j_2
goto l_1
label l_0
load_D j_1
const 1
liftTop
```

[38]This solution to the problem of infinite specialization is also known as *generalization* [12].

```
binop_D sub
store_D j_1
load_D j_2
load_D j_0
binop_D mul
store_D j_2
label l_1
load_D j_1
const 0
liftTop
if_D ne l_0
load_D j_2
return
```

Given this annotated program, the specializer will not attempt to calculate the value of local variable j_2 inside the loop at specialization-time. Hence, specialization of this program will indeed terminate.

Unfortunately, only trivial partial evaluation of the *power* program may be achieved based on the above annotated version of the program.

5.4.2 Interpreting another NORMA2 program

In section 5.3.3 we demonstrated how the NORMA2 interpreter from section 4.4 could be specialized with respect to a simple NORMA2 program.

Consider now the following seemingly innocent NORMA2 program, encoded by the vector $[4, -1, 2, 1, 6, 0]$:

Index	*Opcode*	*Operand*	*Meaning*
0	4	-1	if $X = 0$ goto -1 (stop)
2	2		$X := X - 1$
3	1		$Y := Y + 1$
4	6	0	goto 0

This NORMA2 program returns the same value as is passed in the parameter X (for $X \geq 0$).

Specialization of the NORMA2 interpreter with respect to the program $P = [4, -1, 2, 1, 6, 0]$ would proceed as follows[39]:

Block	*Residual code*	*Pending*
(initial)	(none)	(none)
l_0	`label` $l_0(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 0\}, h)$	do.
l_1	`label` $l_1(4 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$	do.
l_2	`label` $l_2(4 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$	do.
l_3	`label` $l_3(4 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$	do.
l_4	`label` $l_4(4 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$ `load` j_1 `const` 0 `if eq` $l_6(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$	$l_6(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$
l_0	`label` $l_0(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 2\}, h)$	do.
l_1	`label` $l_1(2 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 3\}, h)$	do.

[39]Again, the value of local variable j_0 (P) is unchanged; this is what is indicated by the ellipsis in each local variable environment.

l_2	`label` $l_2(2 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 3\}, h)$ `load` j_1 `const` 1 `binop` sub `store` j_1	$l_6(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 1\}, h)$
l_0	`label` $l_0(nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 3\}, h)$	do.
l_1	`label` $l_1(1 :: nil, \{\ldots, j_2 \mapsto 0, j_3 \mapsto 4\}, h)$	do.
l_0	`label` $l_0(nil, \{\ldots, j_2 \mapsto 1, j_3 \mapsto 4\}, h)$	do.
l_1	`label` $l_1(6 :: nil, \{\ldots, j_2 \mapsto 1, j_3 \mapsto 5\}, h)$	do.
l_2	`label` $l_2(6 :: nil, \{\ldots, j_2 \mapsto 1, j_3 \mapsto 5\}, h)$	do.
l_3	`label` $l_3(6 :: nil, \{\ldots, j_2 \mapsto 1, j_3 \mapsto 5\}, h)$	do.
l_4	`label` $l_4(6 :: nil, \{\ldots, j_2 \mapsto 1, j_3 \mapsto 5\}, h)$	do.
l_5	`label` $l_5(6 :: nil, \{\ldots, j_2 \mapsto 1, j_3 \mapsto 5\}, h)$	do.
l_6	`label` $l_6(nil, \{\ldots, j_2 \mapsto 1, j_3 \mapsto 5\}, h)$	do.
l_0	`label` $l_0(nil, \{\ldots, j_2 \mapsto 1, j_3 \mapsto 0\}, h)$	do.
$\vdots$	$\vdots$	$\vdots$

Specialization would continue infinitely along the same pattern for increasing values of j_2 (Y). As for the *power* example in the previous section, the problem is that a static local variable is updated in a loop controlled by a dynamic conditional.

Note that this problem occurs for the very same annotated core language that was successfully specialized in section 5.3.3 with respect to a different NORMA2 program. In other words, specialization of the NORMA2 interpreter with respect to a given NORMA2 program may or may not terminate, depending on the NORMA2 program. Unfortunately, the problem cannot be solved by reclassifying any of the program parameters here.

In order to keep the specializer from calculating (and changing) the value of local variable j_2 (Y) inside the loop, that local variable should be generalized. One way to achieve this is by modifying the original core language program implementing the NORMA2 interpreter.

The following implementation of the NORMA2 interpreter (shown in Java syntax for brevity) is very similar to the one presented in section 4.4:

```
static int norma2 (int[] P, int X) {
  int[] R = {X, 0};   // registers: R[0] = X, R[1] = Y
  int pc = 0;         // program counter
  while (pc >= 0 && pc < P.length) {
    switch (P[pc++]) {
    case 0: R[0] = R[0] + 1;  continue;
    case 1: R[1] = R[1] + 1;  continue;
    case 2: R[0] = R[0] - 1;  continue;
    case 3: R[1] = R[1] - 1;  continue;
    case 4: if (R[0] == 0) pc = P[pc];
            else pc++;   // skip past immediate argument (branch target)
            continue;
    case 5: if (R[1] == 0) pc = P[pc];
            else pc++;   // skip past immediate argument (branch target)
            continue;
    default: pc = P[pc];
    }
  }
  return R[1];
}
```

This program differs from the interpreter shown in section 4.4 only in that the values of register X and Y are stored as components in the array R.

The modified interpreter may also be implemented in our core language. We do not show the corresponding core language program here, but only note that the two components of the register array would have the same binding-time type in the annotated core language program generated by our binding-time analysis from the core language implementation of the modified NORMA2 interpreter.

Since the program parameter (X) is still considered to be dynamic, the register array (R) will have the binding-time type D[] and the value of register X as well as Y will be considered dynamic. This implies that the specializer will not attempt to calculate (and change) the value of register Y and specialization of the modified NORMA2 interpreter with respect to a given NORMA2 program will terminate for any NORMA2 program.

Unfortunately, somewhat less partial evaluation may be carried out when specializing the modified NORMA2 interpreter with respect to a given NORMA2 program. For example, specialization of the modified interpreter with respect to the NORMA2 program considered in section 5.3.3 would yield the following residual program (after label renaming and removal of superfluous labels):

`const` 0
`store` j_4
`const` 0
`store` j_5
`load` j_1
`store` j_4
`load` j_4
`const` 0
`if` eq l'_6
`load` j_5
`return`
`label` l'_6
`load` j_5
`const` 1
`binop` add
`store` j_5
`load` j_5
`return`

where local variable j_4 and j_5 hold the values of register X and Y, respectively. The residual program still expects the initial value of register X to be passed in local variable j_1.

A post-processing phase could then compress the above residual program to the following program:

`const` 0
`store` j_5
`load` j_1
`const` 0
`if` eq l'_6
`load` j_5
`return`
`label` l'_6
`load` j_5
`const` 1
`binop` add
`return`

This example demonstrates the unavoidable trade-off between optimal specialization and termination of the specializer.

5.4.3 Avoiding infinite specialization

As indicated in the previous examples of infinite specialization, the problem is caused by too many instructions being annotated as static. In some situations this may be avoided by changing the binding-time classification of the program parameters (cf. section 5.4.1) or by redesigning the program to be specialized, as discussed in the previous section. However, a fully automatic and general solution to the problem is to be preferred.

Ensuring finite specialization is a classical problem in connection with partial evaluation [12]. Much effort has been devoted to development of techniques for deciding a sufficient degree of generalization to guarantee termination of the specializer.

One such approach, aimed at off-line program specialization, involves *termination analysis* in connection with binding-time analysis. Once the binding-time analysis has determined a consistent annotation of the original program, the termination analysis calculates a safe approximation to the set of program variables that must be generalized in order to ensure termination.

Termination analyses to ensure finite specialization is still an area of active research.

6 Conclusion

We have presented a binding-time analysis for a simple core language, corresponding to a small subset of the Java Virtual Machine (JVM) instruction set [13]. Similarly to the JVM instruction set, the core language uses an explicit operand stack and a set of 'local variables', accessed by index. The core language operates only on integers and on references to heap-allocated (multi-dimensional) arrays.

The binding-time analysis is constraint-based: for a given core language program, a number of binding-time constraints are generated. Given a classification of the program parameters, a solution to the generated constraints may be found using Birkedal and Welinder's constraint solver [8]. Using the solution to the generated constraints, the original core language program may then be turned into a well-annotated core language program where each core language instruction is annotated with a binding-time type and lift instructions have been inserted as required.

The described binding-time analysis has been implemented using the functional programming language Standard ML [14]. The implementation uses Birkedal and Welinder's implementation of their constraint solver [7] and also uses some modules from the SML-JVM toolkit [3].

The implemented analysis was demonstrated by a number of examples. For each example, we showed the original core language program and the resulting annotated core language program that was generated by our implementation of the binding-time analysis.

A specializer for the annotated core language has not been implemented. We described, however, the flavour of partial evaluation that our binding-time analysis is meant to support. In connection with this, potential problems with infinite specialization were discussed.

It should be fairly straightforward to implement a simple specializer for the annotated core language, based on our description.

6.1 Further work

Our plans and ideas for further work include:

- implementing a specializer for our core language, following the approach described in section 5;
- extending the core language to include:
  - (static) methods,
  - (static) fields,
  - classes and subclasses, and
  - partially static objects;

  and extending the binding-time analysis presented here to support these features.

6.2 Related work

To the best of our knowledge, binding-time analysis and specialization of Java bytecode has not been attempted before. Moreover, our work appears to be the first attempt to do partial evaluation of a (simple) imperative language using an explicit operand stack.

Schultz et al describes an approach to specialization of Java programs, using the Java-to-C compiler *Harissa* and the C specializer *Tempo* [16]. This is probably the work that is most closely related to ours.

Partial evaluators for other imperative languages may also be considered related to our work:

- *C-mix* is a partial evaluator for the C programming language, originally developed by Andersen [1]. Secher et al has redesigned the implementation and improved the user interface [17].

- *Tempo* is also a specializer for C [10]. Tempo has been used to specialize various real-world software, e.g. operating systems code. Tempo also supports the fairly advanced technique of *run-time specialization*, i.e., using optimized binary code templates to do program specialization dynamically at run-time.
- *FSpec* is a partial evaluator for Fortran programs, developed by Baier et al [2]. The Fspec system has been used to specialize various numerical algorithms.

References

[1] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] R. Baier, R. Glück, and R. 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.

[3] P. Bertelsen. The SML-JVM toolkit: a toolkit, written in Standard ML, for generating and manipulating Java bytecode and Java class files. Available at `http://www.dina.kvl.dk/~pmb`.

[4] P. Bertelsen. Semantics of Java byte code. Technical report, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Copenhagen, Denmark, April 1997. Available at `http://www.dina.kvl.dk/~pmb`.

[5] P. Bertelsen. Compiling SML to Java bytecode. Master's thesis, Technical University of Denmark, January 1998. Available at `http://www.dina.kvl.dk/~pmb`.

[6] R. Bird. *Programs & Machines*. John Wiley & Sons, 1976. ISBN 0-471-01650-0.

[7] L. Birkedal and M. Welinder. SML-mix: a partial evaluator for core Standard ML. Available at `http://www.diku.dk/research-groups/topps/activities/sml-mix.html`.

[8] L. Birkedal and M. Welinder. Binding-Time Analysis for Standard ML. *Lisp and Symbolic Computation*, 8(3):191–208, September 1995.

[9] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[10] C. Consel et al. Tempo Specializer: a partial evaluator for C. Available at `http://www.irisa.fr/compose/tempo/`.

[11] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *FPCA*, pages 448–472. 5th ACM Conference, Cambridge, MA, USA, Springer-Verlag, August 1991. Lecture Notes in Computer Science, Vol. 523.

[12] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993. ISBN 0-13-020249-5.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996. ISBN 0-201-63452-X.

[14] R. Milner, M. Tofte, R. Harper, and D.B. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. ISBN 0-262-63181-4.

[15] S. Romanenko and P. Sestoft. Moscow ML: a compiler for a functor-free subset of Standard ML. Available at `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[16] U.P. Schultz, J.L. Lawall, C. Concel, and G. Muller. Towards automatic specialization of Java programs. Technical report, IRISA, 1998. Available at `http://www.irisa.fr/compose/`.

[17] J.P. Secher, H. Makholm, and A. Glenstrup. C-mix: an automatic partial evaluator for the C language. Available at `http://www.diku.dk/research-groups/topps/activities/cmix/`.

A Valid binding-time annotations

This section summarizes the inference rules defined in section 3.4.

$$\lambda, (b_1 :: \sigma), \rho \vdash \texttt{arraylength}_b \;:\; (b_2 :: \sigma), \rho \qquad \text{if } b_1 \lhd\rhd b \wedge b = b_2 \tag{1}$$

$$\lambda, (b_1 :: b_2 :: \sigma), \rho \vdash \texttt{arrayload}_b \;:\; (b_3 :: \sigma), \rho \qquad \text{if } b_1 \lhd\rhd b_2 \wedge b_2 = b \wedge b_3[\,] \leq b_2 \tag{2}$$

$$\lambda, (b_1 :: b_2 :: b_3 :: \sigma), \rho \vdash \texttt{arraystore}_b \;:\; \sigma, \rho \qquad \text{if } b_2 \lhd\rhd b_3 \wedge b_3 = b \wedge b_1[\,] \leq b_3 \tag{3}$$

$$\lambda, (b_1 :: b_2 :: \sigma), \rho \vdash \texttt{binop}_b\ \odot \;:\; (b_3 :: \sigma), \rho \qquad \text{if } b_1 = b_2 = b = b_3 \tag{4}$$

$$\lambda, \sigma, \rho \vdash \texttt{const}\ k \;:\; (\mathsf{S} :: \sigma), \rho \tag{5}$$

$$\lambda, (b_1 :: \sigma), \rho \vdash \texttt{dup}_b \;:\; (b_2 :: b_3 :: \sigma), \rho \qquad \text{if } b_1 \lhd\rhd b \wedge b_1 = b_2 = b_3 \tag{6}$$

$$\lambda, \sigma, \rho \vdash \texttt{goto}\ l \;:\; \sigma', \rho' \qquad \text{if } \lambda(l) = (\sigma, \rho) \tag{7}$$

$$\lambda, (b_1 :: b_2 :: \sigma), \rho \vdash \texttt{if}_b\ r\ l \;:\; \sigma, \rho \qquad \text{if } b_1 \lhd\rhd b \wedge b_2 \lhd\rhd b \wedge \lambda(l) = (\sigma, \rho) \tag{8}$$

$$\lambda, \sigma, \rho \vdash \texttt{label}\ l \;:\; \sigma, \rho \qquad \text{if } \lambda(l) = (\sigma, \rho) \tag{9}$$

$$\lambda, \sigma, \rho \vdash \texttt{load}_b\ j \;:\; (b_1 :: \sigma), \rho \qquad \text{if } \rho(j) = b_1 \wedge b_1 \lhd\rhd b \tag{10}$$

$$\lambda, (b_1 :: \sigma), \rho \vdash \texttt{newarray}_b\ d \;:\; (b_2 :: \sigma), \rho \qquad \text{if } b_1 \lhd\rhd b \wedge \exists b_3 . b_3[\,] \leq b \wedge b = b_2 \tag{11}$$

$$\lambda, (b_1 :: \sigma), \rho \vdash \texttt{pop}_b \;:\; \sigma, \rho \qquad \text{if } b_1 \lhd\rhd b \tag{12}$$

$$\lambda, (\mathsf{D} :: \sigma), \rho \vdash \texttt{return} \;:\; \sigma', \rho' \tag{13}$$

$$\lambda, (b_1 :: \sigma), \rho \vdash \texttt{store}_b\ j \;:\; \sigma, \rho[j \mapsto b_1] \qquad \text{if } b_1 \lhd\rhd b \tag{14}$$

$$\lambda, (b_1 :: b_2 :: \sigma), \rho \vdash \texttt{swap}_b \;:\; (b_4 :: b_3 :: \sigma), \rho \qquad \text{if } (b_1, b_2) \lhd\rhd b \wedge b_1 = b_3 \wedge b_2 = b_4 \tag{15}$$

$$\lambda, (\mathsf{S} :: \sigma), \rho \vdash \texttt{liftTop} \;:\; (\mathsf{D} :: \sigma), \rho \tag{16}$$

$$\lambda, (b_1 :: \mathsf{S} :: \sigma), \rho \vdash \texttt{lift2nd} \;:\; (b_2 :: \mathsf{D} :: \sigma), \rho \qquad \text{if } b_1 = b_2 \tag{17}$$

$$\lambda, \sigma, \rho[j \mapsto \mathsf{S}] \vdash \texttt{lift}\ j \;:\; \sigma, \rho[j \mapsto \mathsf{D}] \tag{18}$$

$$\frac{\lambda, \sigma, \rho \vdash c \;:\; \sigma_1, \rho_1 \qquad \lambda, \sigma_1, \rho_1 \vdash cs \;:\; \sigma_2, \rho_2}{\lambda, \sigma, \rho \vdash c :: cs \;:\; \sigma_2, \rho_2} \tag{19}$$

$$\lambda, \sigma, \rho \vdash \mathit{nil} \;:\; \sigma, \rho \tag{20}$$

An annotated core language program p is well-annotated with respect to a specific label environment λ and local variable binding-time environment ρ iff it holds that

$$\exists \lambda, \sigma_1, \rho_1 \,.\, (\lambda, \mathit{nil}, \rho \vdash \; p \; : \; \sigma_1, \rho_1)$$

where ρ is the initial local variable binding-time environment, holding the binding-time types of the program parameters.