BRICS RS-02-37 Ager et al.: A Symmetric Approach to Compilation and Decompilation

BRICS

Basic Research in Computer Science

A Symmetric Approach to Compilation and Decompilation

Mads Sig Ager
Olivier Danvy
Mayer Goldberg

BRICS Report Series **RS-02-37**

ISSN 0909-0878 **August 2002**

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/02/37/`

A Symmetric Approach to Compilation and Decompilation

Mads Sig Ager and Olivier Danvy — Mayer Goldberg

BRICS *

Dept. of Computer Science, University of Aarhus †

Dept. of Computer Science, Ben Gurion University ‡

August 2002

Abstract

Just as an interpreter for a source language can be turned into a compiler from the source language to a target language, we observe that an interpreter for a target language can be turned into a compiler from the target language to a source language. In both cases, the key issue is the choice of whether to perform an evaluation or to emit code that represents this evaluation.

We substantiate this observation with two source interpreters and two target interpreters. We first consider a source language of arithmetic expressions and a target language for a stack machine, and then the λ-calculus and the SECD-machine language. In each case, we prove that the target-to-source compiler is a left inverse of the source-to-target compiler, i.e., that it is a decompiler.

In the context of partial evaluation, the binding-time shift of going from a source interpreter to a compiler is classically referred to as a Futamura projection. By symmetry, it seems logical to refer to the binding-time shift of going from a target interpreter to a compiler as a Futamura embedding.

To Neil Jones, for his 60th birthday.

*Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: {mads,danvy}@brics.dk

‡Be'er Sheva 84105, Israel.
E-mail: gmayer@cs.bgu.ac.il

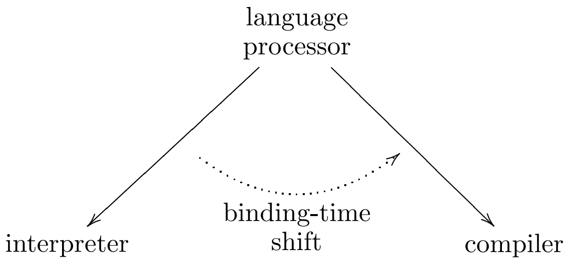
Contents

1 Introduction

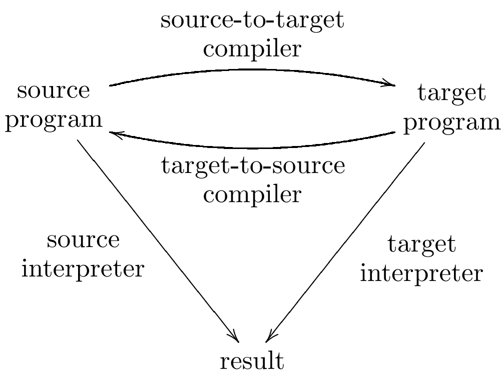
Intuitions run strong when it comes to connecting interpreters and compilers, be it by calculation [51], by derivation [47,61] or by partial evaluation [7,11,18–20,31,38,39,42–44]. These intuitions have provided a fertile ground for two-level languages [56] and code generation [8,59,60,62]. Common to these approaches is the idea, in two-level programs, of shifting from a semantic model to a syntactic model in order to generate code: Rather than performing an evaluation in a source-language interpreter, one emits target-language code that represents this evaluation, as in a compiler.

We observe that this binding-time shift directly applies to decompiling: Rather than performing an evaluation in a target-language interpreter, one can emit source-language code that represents this evaluation, as in a decompiler.

In the rest of this article, we illustrate both instances of this binding-time shift with a source language of arithmetic expressions and a target language for a stack machine (Section 2), and then with the λ-calculus and the SECD-machine language (Section 3). We stage each language processor as a core semantics, represented as an ML functor, and as interpretations, represented as ML structures. This staging corresponds to the factorized semantics of Jones and Nielson [41]. We show how instantiating each functor with elementary evaluation functions yields an interpreter and how instantiating it with elementary code-generating functions yields a compiler:



In one case, the compiler maps a source program to a target program, and in the other, it maps a target program to a source program:



In each of Sections 2 and 3, we formally prove that the target-to-source compiler is a left inverse of the source-to-target compiler, i.e., that it is a decompiler.

2 Arithmetic expressions and a stack machine

We consider a simplified source language of arithmetic expressions and a simplified target language for a stack machine. It is straightforward to extend both languages with more arithmetic operators.

2.1 Staged specification of the source language

The source language is as follows.

```
structure Source
= struct
    datatype exp = LIT of int
                 | PLUS of exp * exp
    type program = exp
  end
```

2.1.1 The core semantics

Recursive traversal of programs in the source language can be expressed generically as follows, using an ML functor. In this functor, the function `process` implements the fold function associated with the data type of the source language [6,16]. This fold function is parameterized by a structure of type `INTEGER`, which packages two types and a collection of operators corresponding to each constructor of the data type.

```
signature INTEGER
= sig
    type integer
    type result
    val lit : int -> integer
    val plus : integer * integer -> integer
    val compute : integer -> result
  end;

signature SOURCE_PROCESSOR
= sig
    type result
    val process : Source.program -> result
  end
```

```
functor Make_source_processor (structure I : INTEGER)
: SOURCE_PROCESSOR
= struct
    type result = I.result

    fun process p
        = let fun walk (Source.LIT n)
                    = I.lit n
                | walk (Source.PLUS (e1, e2))
                    = I.plus (walk e1, walk e2)
          in I.compute (walk p)
          end
  end
```

2.1.2 A code-generation instantiation: source identity

As an example of the use of `Make_source_processor`, this functor can be instantiated to obtain the identity transformation over source programs. To this end, we specify a structure of type `INTEGER` containing a syntactic representation of integers. In this structure, `integer` is defined as the type of source expressions, `result` as the type of source programs, and the operators as the corresponding code-generating functions:

```
structure Integer_source_syntax : INTEGER
= struct
    type integer = Source.exp
    type result = Source.program

    fun lit n
        = Source.LIT n
    fun plus (e1, e2)
        = Source.PLUS (e1, e2)
    fun compute e
        = e
  end

structure Source_identity
= Make_source_processor (structure I = Integer_source_syntax)
```

2.2 Staged specification of the target language

A target program is a list of instructions for a stack machine:

```
structure Target
= struct
    datatype instr = PUSH of int
                   | ADD
    type program = instr list
  end
```

2.2.1 The core semantics

Recursive traversal of programs in this language can be expressed generically as follows, again using an ML functor. In this functor, the function `process` implements the fold function associated with the target data type. This fold function is parameterized by a structure of type `TARGET_PARAMETERS`, which packages two types and a collection of operators corresponding to each constructor of the data type.

```
signature TARGET_PARAMETERS
= sig
    type computation
    type result
    val terminate : computation
    val push : int * computation -> computation
    val add : computation -> computation
    val compute : computation -> result
  end

signature TARGET_PROCESSOR
= sig
    type result
    val process : Target.program -> result
  end

functor Make_target_processor (structure P : TARGET_PARAMETERS)
: TARGET_PROCESSOR
= struct
    type result = P.result

    fun process p
        = let fun walk nil
                    = P.terminate
                | walk ((Target.PUSH n) :: is)
                    = P.push (n, walk is)
                | walk (Target.ADD :: is)
                    = P.add (walk is)
          in P.compute (walk p)
          end
  end
```

2.2.2 A code-generation instantiation: target identity

As in Section 2.1.2, `Make_target_processor` can be instantiated to obtain the identity transformation over target programs by defining `computation` as the type of lists of target instructions, `result` as the type of target programs, and the operators as the corresponding code-generating functions:

```
structure Target_parameters_identity : TARGET_PARAMETERS
= struct
    type computation = Target.instr list
    type result = Target.program

    val terminate = nil
    fun push (n, is)
        = (Target.PUSH n) :: is
    fun add is
        = Target.ADD :: is
    fun compute is
        = is
  end

structure Target_identity
= Make_target_processor (structure P = Target_parameters_identity)
```

2.3 Interpretation and compilation for the source language

We instantiate `Make_source_processor` into an interpreter for the source language and into a compiler from the source language to the target language.

2.3.1 An evaluation instantiation: source interpretation

It is straightforward to instantiate the functor of Section 2.1.1 to obtain an interpreter. To this end, we define a structure of type `INTEGER` containing a semantic representation of integers. In this structure, both `integer` and `result` are defined as the type `int`, and the operators as the standard arithmetic operators:

```
structure Integer_semantics : INTEGER
= struct
    type integer = int
    type result = int

    fun lit n
        = n
    fun plus (n1, n2)
        = n1 + n2
    fun compute n
        = n
  end
```

We can now instantiate the functor of Section 2.1.1 to obtain an interpreter for the source language:

```
structure Source_int
= Make_source_processor (structure I = Integer_semantics)
```

For example, applying `Source_int.process` to the source program

```
PLUS (PLUS (LIT 10, LIT 20), PLUS (LIT 30, LIT 40))
```

yields the integer `100`.

Compared to the identity instantiation of Section 2.1.2, rather than choosing a syntactic model and emitting source-language code, we choose a semantic model and carry out evaluation. The two instantiations illustrate a simple binding-time shift.

2.3.2 A code-generation instantiation: source compilation (version 1)

It is also straightforward to instantiate `Make_source_processor` to obtain a compiler to the target language. To this end, we implement a binding-time shift of going from an interpreter to a compiler with another structure of type `INTEGER` containing a syntactic representation of integers. In this structure, `integer` is defined as the type of lists of target instructions, `result` as the type of target programs, and operators as first-order code-generating functions:

```
structure Integer_target_syntax1 : INTEGER
= struct
    type integer = Target.instr list
    type result = Target.program

    fun lit n
        = [Target.PUSH n]
    fun plus (is1, is2)
        = is1 @ is2 @ [Target.ADD]
    fun compute is
        = is
  end

structure Source_cmp1
= Make_source_processor (structure I = Integer_target_syntax1)
```

For example, applying `Source_cmp1.process` to the source program

```
PLUS (PLUS (LIT 10, LIT 20), PLUS (LIT 30, LIT 40))
```

yields the following target program:

```
[PUSH 10, PUSH 20, ADD, PUSH 30, PUSH 40, ADD, ADD]
```

2.3.3 A code-generation instantiation: source compilation (version 2)

We can also instantiate `Make_source_processor` to obtain a less trivial but equivalent compiler that uses an accumulator instead of concatenating intermediate lists of instructions. To this end, we define yet another structure of type `INTEGER` containing a syntactic representation of integers. In this structure, `integer` is defined as a transformer of lists of target instructions, `result` as the type of target programs, and the operators as second-order code-generating functions:

```
structure Integer_target_syntax2 : INTEGER
= struct
    type integer = Target.instr list -> Target.instr list
    type result = Target.program

    fun lit n
        = (fn is => (Target.PUSH n) :: is)
    fun plus (c1, c2)
        = (fn is => c1 (c2 (Target.ADD :: is)))
    fun compute c
        = c nil
  end
```

In passing, let us stress the relation between Version 1 and Version 2 of the compiler with the following equivalent definition of Version 2, using a curried version of list construction and function composition instead of list construction and list concatenation, respectively:

```
structure Integer_target_syntax2' : INTEGER
= struct
    type integer = Target.instr list -> Target.instr list
    type result = Target.program

    fun cons x
        = (fn xs => x :: xs)

    fun lit n
        = cons (Target.PUSH n)
    fun plus (c1, c2)
        = c1 o c2 o (cons Target.ADD)
    fun compute c
        = c nil
  end
```

Either of `Integer_target_syntax2` or `Integer_target_syntax2'` can be used to obtain a compiler from the source language to the target language:

```
structure Source_cmp2
= Make_source_processor (structure I = Integer_target_syntax2)

structure Source_cmp2'
= Make_source_processor (structure I = Integer_target_syntax2')
```

2.4 Interpretation and compilation for the target language

A target program is processed using a stack. This process is partial in that it expects the stack to be well-formed. We make it total in ML using an `option` type:

```
datatype 'a option = NONE
                   | SOME of 'a
```

When interpreting programs, the stack contains integers. According to the binding-time shift of going from an interpreter to a compiler, when compiling programs, the stack should contain representations of integers. We thus further parameterize the parameters of the target-language processor by a structure of type `INTEGER`:

```
functor Make_target_parameters (structure I : INTEGER)
: TARGET_PARAMETERS
= struct
    type computation = I.integer list -> I.integer list option
    type result = I.result option

    val terminate = (fn s => SOME s)
    fun push (n, c)
         = (fn s => c ((I.lit n) :: s))
    fun add c
        = (fn (x2 :: x1 :: xs) => c ((I.plus (x1, x2)) :: xs)
            | _ => NONE)
    fun compute c
        = (case c nil
             of (SOME (x :: nil)) => SOME (I.compute x)
              | _ => NONE)
  end
```

2.4.1 An evaluation instantiation: target interpretation

It is straightforward to instantiate `Make_target_parameters` to obtain the target parameters for an interpreter. To this end, we use the semantic representation of the integers specified in Section 2.3.1:

```
structure Target_parameters_semantics
= Make_target_parameters (structure I = Integer_semantics)
```

We can now instantiate the functor of Section 2.2.1 to obtain an interpreter for the target language:

```
structure Target_int
= Make_target_processor (structure P = Target_parameters_semantics)
```

For example, applying `Target_int.process` to the target program

```
[PUSH 10, PUSH 20, ADD, PUSH 30, PUSH 40, ADD, ADD]
```

yields the optional integer `SOME 100`.

2.4.2 A code-generation instantiation: target compilation

It is also straightforward to instantiate `Make_target_parameters` to obtain the target parameters for a compiler to the source language. To this end, we use the syntactic representation of the integers specified in Section 2.1.2:

```
structure Target_parameters_source_syntax
= Make_target_parameters (structure I = Integer_source_syntax)
```

We can now instantiate the functor of Section 2.2.1 to obtain a compiler from the target language to the source language:

```
structure Target_cmp
= Make_target_processor (structure P = Target_parameters_source_syntax)
```

For example, applying `Target_cmp.process` to the target program

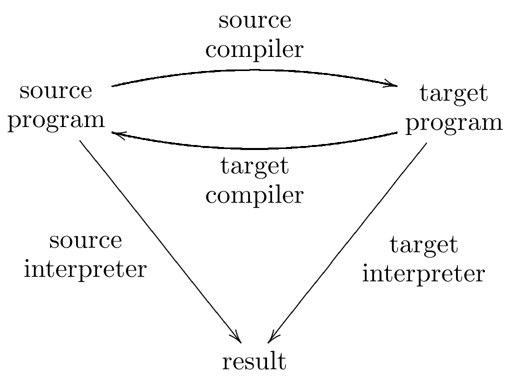
```
[PUSH 10, PUSH 20, ADD, PUSH 30, PUSH 40, ADD, ADD]
```

yields the following optional source program:

```
SOME (PLUS (PLUS (LIT 10, LIT 20), PLUS (LIT 30, LIT 40)))
```

2.5 Properties

We successively consider the total correctness of the source compiler with respect to the source interpreter and the target interpreter, the partial correctness of the target compiler with respect to the target interpreter and the source interpreter, and the left inverseness of the source compiler and of the target compiler:



In particular, we prove that the target compiler is a left inverse of the source compiler and therefore that it is a decompiler.

Terminology and notation:

- `Source_int.process` is the `process` function of the functor `Make_source_processor` of Section 2.1.1 instantiated with the structure `Integer_semantics` of Section 2.3.1. We refer to the corresponding `walk` function as `w_s_int` (for "walk function of the source interpreter").

- `Source_cmp2.process` is the `process` function of the functor `Make_source_processor` of Section 2.1.1 instantiated with the structure `Integer_target_syntax2` of Section 2.3.3. We refer to the corresponding `walk` function as `w_s_cmp` (for "walk function of the source compiler").
- `Target_int.process` is the `process` function of the functor `Make_target_processor` of Section 2.2.1 instantiated with the structure `Target_parameters_semantics` of Section 2.4.1. We refer to the corresponding `walk` function as `w_t_int` (for "walk function of the target interpreter").
- `Target_cmp.process` is the `process` function of the functor `Make_target_processor` of Section 2.2.1 instantiated with the structure `Target_parameters_source_syntax` of Section 2.4.2. We refer to the corresponding `walk` function as `w_t_cmp` (for "walk function of the target compiler").

All the functions above are pure and total, i.e., they are side-effect free and they always terminate, since they only use primitive recursion (the `process` functions in the functors `Make_source_processor` and `Make_target_processor` are fold functions).

We reason equationally on the ML syntax of the interpreters and compilers, using observational equivalence. We say that two expressions `e1` and `e2` are observationally equivalent, which we write as

`e1` ≅ `e2`

whenever evaluating `e1` and `e2` in the same context yield the same result. Our equational reasoning involves unfolding function calls, which is sound for pure and total functions.

2.5.1 Total correctness of the source compiler

The compiler is correct if composing `Target_int.process` and `Source_cmp.process` yields the same function as `Source_int.process`. We use the following lemma as a stepping stone for proving this correctness.

Lemma 1 *For all ML values* `e : Source.exp`*,* `is : Target.instr list`*, and* `s : int list`*, the following observational equivalence holds:*

```
w_t_int (w_s_cmp e is) s  ≅  w_t_int is ((w_s_int e) :: s).
```

Proof: The proof is by structural induction on the source syntax.

Base case: `Source.LIT n`

For all ML values `is : Target.instr list` and `s : int list`, we want to show the following observational equivalence:

```
w_t_int (w_s_cmp (Source.LIT n) is) s
≅  w_t_int is ((w_s_int (Source.LIT n)) :: s)
```

We proceed by unfolding function calls:

```
w_t_int (w_s_cmp (Source.LIT n) is) s
≅ (unfolding w_s_cmp)
w_t_int (Integer_target_syntax2.lit n is) s
≅ (unfolding Integer_target_syntax2.lit)
w_t_int ((fn is => ((Target.PUSH n) :: is)) is) s
≅ (function application)
w_t_int ((Target.PUSH n) :: is) s
≅ (unfolding w_t_int)
Target_parameters_semantics.push (n, w_t_int is) s
≅ (unfolding Target_parameters_semantics.push)
(fn s => w_t_int is ((Integer_semantics.lit n) :: s)) s
≅ (function application)
w_t_int is ((Integer_semantics.lit n) :: s)
≅ (unfolding Integer_semantics.lit)
w_t_int is (n :: s)
```

Conversely,

```
w_t_int is ((w_s_int (Source.LIT n)) :: s)
≅ (unfolding w_s_int)
w_t_int is ((Integer_semantics.lit n) :: s)
≅ (unfolding Integer_semantics.lit n)
w_t_int is (n :: s)
```

Induction case: `Source.PLUS (e1, e2)`

For all ML values `is : Target.instr list`, `s : int list`, and for ML values `e1 : Source.exp` and `e2 : Source.exp` satisfying the induction hypothesis, we want to show the following observational equivalence:

```
w_t_int (w_s_cmp (Source.PLUS (e1, e2)) is) s
≅  w_t_int is ((w_s_int (Source.PLUS (e1, e2))) :: s)
```

Again, we proceed by unfolding function calls:

```
w_t_int (w_s_cmp (Source.PLUS (e1, e2)) is) s
≅ (unfolding w_s_cmp)
w_t_int (Integer_target_syntax2.plus (w_s_cmp e1, w_s_cmp e2) is) s
≅ (unfolding Integer_target_syntax2.plus)
w_t_int ((fn is => w_s_cmp e1 (w_s_cmp e2 (Target.ADD :: is))) is) s
≅ (function application)
w_t_int (w_s_cmp e1 (w_s_cmp e2 (Target.ADD :: is))) s
≅ (induction hypothesis on e1)
w_t_int (w_s_cmp e2 (Target.ADD :: is)) ((w_s_int e1) :: s)
≅ (induction hypothesis on e2)
w_t_int (Target.ADD :: is) ((w_s_int e2) :: (w_s_int e1) :: s)
≅ (unfolding w_t_int)
```

```
Target_parameters_semantics.add (w_t_int is)
                              ((w_s_int e2) :: (w_s_int e1) :: s)
```
≅ (unfolding `Target_parameters_semantics.add`)
```
(fn (x2 :: x1 :: xs)
    => w_t_int is (Integer_semantics.plus (x1, x2) :: xs)
  | _
     => NONE)
((w_s_int e2) :: (w_s_int e1) :: s)
```
≅ (function application)
```
w_t_int is (Integer_semantics.plus ((w_s_int e1), (w_s_int e2)) :: s)
```
≅ (unfolding `Integer_semantics.plus`)
```
w_t_int is (((w_s_int e1) + (w_s_int e2)) :: s)
```

Conversely,

```
w_t_int is ((w_s_int (Source.PLUS (e1, e2))) :: s)
```
≅ (unfolding `w_s_int`)
```
w_t_int is ((Integer_semantics.plus (w_s_int e1, w_s_int e2)) :: s)
```
≅ (unfolding `Integer_semantics.plus`)
```
w_t_int is (((w_s_int e1) + (w_s_int e2)) :: s)
```
□

Theorem 1 *For ML values* `sp : Source.program`*, the following observational equivalence holds:*

`Target_int.process (Source_cmp2.process sp)` ≅ `SOME (Source_int.process sp)`.

Proof: For all ML values `sp : Source.program` and `tp : Target.program`, the following observational equivalences holds:

```
Source_int.process sp
```
≅ (unfolding `Source_int.process`)
```
Integer_semantics.compute (w_s_int sp)
```
≅ (unfolding `Integer_semantics.compute`)
```
w_s_int sp
```

```
Source_cmp2.process sp
```
≅ (unfolding `Source_cmp2.process`)
```
Integer_target_syntax2.compute (w_s_cmp sp)
```
≅ (unfolding `Integer_target_syntax2.compute`)
```
w_s_cmp sp nil
```

```
Target_int.process tp
```
≅ (unfolding `Target_int.process`)
```
Target_parameters_semantics.compute (w_t_int tp)
```
≅ (unfolding `Target_parameters_semantics.compute`)
```
case w_t_int tp nil
  of (SOME (x :: nil)) => SOME (Integer_semantics.compute x)
   | _ => NONE
```

For all ML values `sp : Source.program` we therefore have to prove that the following observational equivalence holds:

```
/case w_t_int (w_s_cmp sp nil) nil                \
|  of (SOME (x :: nil))                           |
|     => SOME (Integer_semantics.compute x)       |  ≅  SOME (w_s_int sp)
|   | _                                           |
\     => NONE                                     /
```

This observational equivalence, however, follows from Lemma 1. Indeed, for all ML values `e : Source.exp`, `nil : Target.instr list`, and `s : int list`, the observational equivalence of Lemma 1 reads as

```
w_t_int (w_s_cmp e nil) nil  ≅  w_t_int nil ((w_s_int e) :: nil)
```

In particular,

```
w_t_int nil ((w_s_int e) :: nil)
≅ (unfolding w_t_int)
Target_parameters_semantics.terminate ((w_s_int e) :: nil)
≅ (unfolding Target_parameters_semantics.terminate)
(fn s => SOME s) ((w_s_int e) :: nil)
≅ (function application)
SOME ((w_s_int e) :: nil)
```

Since source programs are expressions and target programs are lists of instructions,

```
case w_t_int (w_s_cmp sp nil) nil
  of (SOME (x :: nil)) => SOME (Integer_semantics.compute x)
   | _ => NONE
≅ (using the observational equivalence just above in context)
case SOME ((w_s_int sp) :: nil)
  of (SOME (x :: nil)) => SOME (Integer_semantics.compute x)
   | _ => NONE
≅ (reducing the case expression)
SOME (Integer_semantics.compute (w_s_int sp))
≅ (unfolding Integer_semantics.compute)
SOME (w_s_int sp)
```

which concludes the proof. □

2.5.2 Partial correctness of the target compiler

As in Section 2.5.1, the compiler is correct if composing `Source_int.process` and `Target_cmp.process` yields the same function as `Target_int.process`. The issue, however, is more murky here because not all values of type `Target.program` are well-formed programs, as indicated by the `option` type in Section 2.4. Such ill-formed target programs are the reason why `Target_int.process` and `Target_cmp.`

process may yield NONE. On the other hand, it is a corollary of Theorem 1 that compiling a source expression yields a well-formed target program and that interpreting a well-formed target program yields SOME n, for some integer n.

We leave the issue of partial correctness aside, and instead we turn to proving that the target compiler is a left inverse of the source compiler.

2.5.3 Left inverseness

We use the following lemma as a stepping stone for proving that Target_cmp.process is a left inverse of Source_cmp2.process for all source expressions.

Lemma 2 *For all ML values* e : Source.exp*,* is : Target.instr list*, and* s : Source.exp list*, the following observational equivalence holds:*

```
w_t_cmp (w_s_cmp e is) s  ≅  w_t_cmp is (e :: s).
```

Proof: The proof is by structural induction on the source syntax.

Base case: Source.LIT n

For all ML values is : Target.instr list and s : Source.exp list, we want to show the following observational equivalence:

```
w_t_cmp (w_s_cmp (Source.LIT n) is) s
≅  w_t_cmp is ((Source.LIT n) :: s)
```

We proceed by unfolding function calls:

```
w_t_cmp (w_s_cmp (Source.LIT n) is) s
≅  (unfolding w_s_cmp)
w_t_cmp (Integer_target_syntax2.int n is) s
≅  (unfolding Integer_target_syntax2.int)
w_t_cmp ((fn is => ((Target.PUSH n) :: is)) is) s
≅  (function application)
w_t_cmp ((Target.PUSH n) :: is) s
≅  (unfolding w_t_cmp)
Target_parameters_source_syntax.push (n, w_t_cmp is) s
≅  (unfolding Target_parameters_source_syntax.push)
(fn s => w_t_cmp is ((Integer_source_syntax.lit n) :: s)) s
≅  (function application)
w_t_cmp is ((Integer_source_syntax.lit n) :: s)
≅  (unfolding Integer_source_syntax.lit)
w_t_cmp is ((Source.LIT n) :: s)
```

Induction case: Source.PLUS (e1, e2)

For all ML values is : Target.instr list, s : Source.exp list, and for all ML values e1 : Source.exp and e2 : Source.exp satisfying the induction hypothesis, we want to show the following observational equivalence:

```
w_t_cmp (w_s_cmp (Source.PLUS (e1, e2)) is) s
≅  w_t_cmp is ((Source.PLUS (e1, e2)) :: s)
```

Again, we proceed by unfolding function calls:

```
w_t_cmp (w_s_cmp (Source.PLUS (e1, e2)) is) s
≅ (unfolding w_s_cmp)
w_t_cmp (Integer_target_syntax2.plus (w_s_cmp e1, w_s_cmp e2) is) s
≅ (unfolding Integer_target_syntax2.plus)
w_t_cmp ((fn is => w_s_cmp e1 (w_s_cmp e2 (Target.ADD :: is))) is) s
≅ (function application)
w_t_cmp (w_s_cmp e1 (w_s_cmp e2 (Target.ADD :: is))) s
≅ (induction hypothesis on e1)
w_t_cmp (w_s_cmp e2 (Target.ADD :: is)) (e1 :: s)
≅ (induction hypothesis on e2)
w_t_cmp (Target.ADD :: is) (e2 :: e1 :: s)
≅ (unfolding w_t_cmp)
Target_parameters_source_syntax.add (w_t_cmp is) (e2 :: e1 :: s)
≅ (unfolding Target_parameters_source_syntax.add)
(fn (x2 :: x1 :: xs)
    => w_t_cmp is ((Integer_source_syntax.plus (x1, x2)) :: xs)
  | _
    => NONE)
(e2 :: e1 :: s)
≅ (function application)
w_t_cmp is ((Integer_source_syntax.plus (e1, e2)) :: s)
≅ (unfolding Integer_source_syntax.plus)
w_t_cmp is ((Source.PLUS (e1, e2)) :: s)
```

□

Theorem 2 *For all ML values* `sp : Source.program`*, the following observational equivalence holds:*

`Target_cmp.process (Source_cmp2.process sp)` ≅ `SOME sp`.

Proof: For all ML values `sp : Source.program` and `tp : Target.program`, the following observational equivalences holds:

```
Source_cmp2.process sp
≅ (unfolding Source_cmp2.process)
Integer_target_syntax2.compute (w_s_cmp sp)
≅ (unfolding Integer_target_syntax2.compute)
w_s_cmp sp nil
```

```
Target_cmp.process tp
≅ (unfolding Target_cmp.process)
Target_parameters_source_syntax.compute (w_t_cmp tp)
≅ (unfolding Target_parameters_source_syntax.compute)
case w_t_cmp tp nil
  of (SOME (x :: nil)) => SOME (Integer_source_syntax.compute x)
   | _ => NONE
```

For all ML values `sp : Source.program`, we therefore have to prove that the following observational equivalence holds:

```
( case w_t_cmp (w_s_cmp sp nil) nil            )
(   of (SOME (x :: nil))                       )
(      => SOME (Integer_source_syntax.compute x) )  ≅  SOME sp
(    | _                                       )
(      => NONE                                 )
```

This observational equivalence, however, follows from Lemma 2. Indeed, for all ML values `e : Source.exp`, `nil : Target.instr list`, and `nil : Source.exp list`, the observational equivalence of Lemma 2 reads as

```
w_t_cmp (w_s_cmp e nil) nil  ≅  w_t_cmp nil (e :: nil)
```

In particular,

```
w_t_cmp nil (e :: nil)
≅ (unfolding w_t_cmp)
Target_parameters_source_syntax.terminate (e :: nil)
≅ (unfolding Target_parameters_source_syntax.terminate)
(fn s => SOME s) (e :: nil)
≅ (function application)
SOME (e :: nil)
```

Since source programs are expressions and target programs are lists of instructions,

```
case w_t_cmp (w_s_cmp sp nil) nil
  of (SOME (x :: nil)) => SOME (Integer_source_syntax.compute x)
   | _ => NONE
≅ (using the observational equivalence just above in context)
case SOME (sp :: nil)
  of (SOME (x :: nil)) => SOME (Integer_source_syntax.compute x)
   | _ => NONE
≅ (reducing the case expression)
SOME (Integer_source_syntax.compute sp)
≅ (unfolding Integer_source_syntax.compute)
SOME sp
```

which concludes the proof. □

2.6 Summary

We have systematically parameterized a source-language processor and a target-language processor and instantiated them into identity transformations, interpreters, and compilers. We also have shown that the target compiler is a left-inverse of the source compiler, and thus a decompiler.

Most of our instantiations hinge on a particular representation of integers—syntactic or semantic. The exception is the identity transformation over target programs, in Section 2.2.2, which hinges on a syntactic instantiation of target parameters. We can, however, instantiate `Make_target_parameters` with either of the syntactic interpretations of the integers in Sections 2.3.2 or 2.3.3:

```
structure Target_parameters_target_syntax1
= Make_target_parameters (structure I = Integer_target_syntax1)

structure Target_parameters_target_syntax2
= Make_target_parameters (structure I = Integer_target_syntax2)
```

We can now instantiate the functor of Section 2.2.1:

```
structure Target_identity1
= Make_target_processor (structure P = Target_parameters_target_syntax1)

structure Target_identity2
= Make_target_processor (structure P = Target_parameters_target_syntax2)
```

In this instantiation, the target program is processed with a stack and each component is mapped to a representation of an integer in either `Integer_target_syntax1` or `Integer_target_syntax2`, i.e., to target code. The instantiation yields a `process` function of type `Target.program -> Target.program option`. This `process` function reflects the partial correctness mentioned in Section 2.5.2 in that it maps any well-formed target program `p` into `SOME p` and all the other target programs into `NONE`.

Overall, we have shown that just as specializing a source-language processor can achieve compilation to a target language, specializing a target-language processor can achieve decompilation to a source language. This observation is very simple, but the authors have not seen it stated elsewhere. For example, specific efforts have been dedicated to decompiling compiled arithmetic expressions, independently of their interpretation, compilation, and execution [13, 14, 49].

3 Lambda-terms and the SECD machine

In this section we show that the symmetric approach to compilation and decompilation scales to an expression language with binding, namely the λ-calculus. We consider Henderson's version of the SECD machine [35, 46, 53].

3.1 Staged specification of the source language

The source language is the untyped λ-calculus with integers and a plus operator. A program is a closed term.

```
structure Source
= struct
    type ide = string
```

```
    datatype term = LIT of int
                  | PLUS of term * term
                  | VAR of ide
                  | LAM of ide * term
                  | APP of term * term
    type program = term
  end
```

3.1.1 The core semantics

Recursive traversal of programs in this language can be expressed generically using an ML functor as in Section 2.1.

```
signature SOURCE_PARAMETERS
= sig
    type computation
    type result
    val lit : int -> computation
    val plus : computation * computation -> computation
    val var : int -> computation
    val lam : computation -> computation
    val app : computation * computation -> computation
    val compute : computation -> result
  end

signature SOURCE_PROCESSOR
= sig
    type result
    val process : Source.program -> result
  end

functor Make_source_processor (structure P : SOURCE_PARAMETERS)
: SOURCE_PROCESSOR
= struct
    type result = P.result

    fun process p
        = let fun walk (Source.LIT n) xs
                  = P.lit n
                | walk (Source.PLUS (t1, t2)) xs
                  = P.plus (walk t1 xs, walk t2 xs)
                | walk (Source.VAR x) xs
                  = P.var (Index.establish (x, xs))
                | walk (Source.LAM (x, t)) xs
                  = P.lam (walk t (x :: xs))
                | walk (Source.APP (t0, t1)) env
                  = P.app (walk t0 env, walk t1 env)
          in P.compute (walk p nil)
          end
  end
```

In order to account for bindings, the `walk` function threads a lexical environment `xs`. This environment is extended for each λ-abstraction and consulted for each occurrence of a variable. The lexical offset of each occurrence of a variable is established using `Index.establish`. (Given two ML values `x : Source.ide` and `xs : Source.ide list` where `x` occurs, applying `Index.establish` to `x` and `xs` yields the index of the first occurrence of `x` in `xs`.)

3.1.2 A code-generation instantiation: source identity modulo renaming

As an example of the use of `Make_source_processor`, this functor can be instantiated as follows to obtain the identity transformation over source programs, modulo renaming. To this end, we define a structure of type `SOURCE_PARAMETERS` where `computation` is a mapping from a list of identifiers to a source term, `result` is the type of source programs, and the operators are the corresponding code-generating functions:

```
structure Source_parameters_identity : SOURCE_PARAMETERS
= struct
    type computation = Source.ide list -> Source.term
    type result = Source.program

    fun lit n
        = (fn xs => Source.LIT n)
    fun plus (c1, c2)
        = (fn xs => Source.PLUS (c1 xs, c2 xs))
    fun var i
        = (fn xs => Source.VAR (Index.fetch (xs, i)))
    fun lam c
        = (fn xs => let val x = "x" ^ Int.toString (length xs)
                    in Source.LAM (x, c (x :: xs))
                    end)
    fun app (c0, c1)
        = (fn xs => Source.APP (c0 xs, c1 xs))
    fun compute c
        = c nil
  end

structure Source_identity
= Make_source_processor (structure P = Source_parameters_identity)
```

Fresh identifiers are needed to construct source λ-abstractions. We obtain them from the current de Bruijn level. These fresh identifiers are grouped in a list `xs` in the reverse order of their declaration. For each λ-abstraction, the list is extended, and for each occurrence of a variable, the corresponding fresh identifier is fetched using `Index.fetch`. (Given two values `i : int` and `xs : Source.ide list`, applying `Index.fetch` to `xs` and `i` fetches the corresponding identifier in `xs`.) A computation is a mapping from lists of fresh identifiers to source terms.

For example, the source term

```
LAM ("a", LAM ("b", APP (APP (LAM ("x", VAR "x"),
                              LAM ("y", VAR "y")),
                         APP (VAR "a", VAR "b"))))
```

is mapped into the following source term:

```
LAM ("x0", LAM ("x1", APP (APP (LAM ("x2",VAR "x2"),
                                LAM ("x2",VAR "x2")),
                           APP (VAR "x0",VAR "x1"))))
```

3.2 Staged specification of the target language

A target program is a list of instructions for the SECD machine [35]:

```
structure Target
= struct
    datatype instr = PUSH of int
                   | ADD
                   | ACCESS of int
                   | CLOSE of instr list
                   | CALL
   type program = instr list
  end
```

Unlike the other interpreters considered in this article, the SECD machine is not directly defined by induction over the structure of target programs. For the sake of familiarity, we follow the canonical definition to write the target-language interpreter in Section 3.4.1. (Therefore, we stay away from the gymnastics of using a functor implementing a recursive descent, as in Appendix A.)

3.3 Interpretation and compilation for the source language

We instantiate `Make_source_processor` into an interpreter for the source language and into a compiler from the source language to the target language.

3.3.1 An evaluation instantiation: source interpretation

In order to instantiate the functor of Section 3.1.1 to obtain a call-by-value interpreter for the source language, we define a data type of values containing integers and functions from values to values. The computation type is then defined to be a mapping from environments, represented by lists of values, to values. The result type is defined to be values.

```
structure Source_parameters_std : SOURCE_PARAMETERS
= struct
    datatype value = INT of int
                   | FUN of value -> value option
```

```
    type computation = value list -> value option
    type result = value option

    fun lit n
        = (fn vs => SOME (INT n))
    fun plus (c1, c2)
        = (fn vs => (case (c1 vs, c2 vs)
                       of (SOME (INT n1), SOME (INT n2))
                          => SOME (INT (n1 + n2))
                        | (_, _)
                          => NONE))
    fun var i
        = (fn vs => SOME (Index.fetch (vs, i)))
    fun lam c
        = (fn vs => SOME (FUN (fn v => c (v :: vs))))
    fun app (c0, c1)
        = (fn vs => (case (c0 vs, c1 vs)
                       of (SOME (FUN f), SOME v)
                          => f v
                        | _
                          => NONE))
    fun compute c
        = c nil
  end

structure Source_int
= Make_source_processor (structure P = Source_parameters_std)
```

For example, applying `Source_int.process` to the source program

```
APP (APP (LAM ("a", LAM ("b", VAR "a")), LIT 10), LIT 20)
```

yields the optional value `SOME (INT 10)`.

3.3.2 A code-generation instantiation: source compilation (version 1)

It is also straightforward to instantiate `Make_source_processor` to obtain a compiler for the source language, by defining both `computation` and `result` as lists of instructions, and by defining the operators as first-order code-generating functions, as in Section 2.3.2:

```
structure Source_parameters_cogen1 : SOURCE_PARAMETERS
= struct
    type computation = Target.instr list
    type result = Target.program

    fun lit n
        = [Target.PUSH n]
    fun plus (is1, is2)
        = is1 @ is2 @ [Target.ADD]
```

```
    fun var n
        = [Target.ACCESS n]
    fun lam is
        = [Target.CLOSE is]
    fun app (is0, is1)
        = is0 @ is1 @ [Target.CALL]
    fun compute is
        = is
  end

structure Source_cmp1
= Make_source_processor (structure P = Source_parameters_cogen1)
```

The resulting compiler is a subset of Henderson's compiler for the SECD machine [35].

3.3.3 A code-generation instantiation: source compilation (version 2)

As in Section 2.3.3, we can also instantiate `Make_source_processor` to obtain a less trivial but equivalent compiler that uses an accumulator instead of concatenating intermediate lists of instructions. To this end, we define `computation` as a transformer of lists of instructions, `result` as a program, and the operators as second-order code-generating functions:

```
structure Source_parameters_cogen2 : SOURCE_PARAMETERS
= struct
    type computation = Target.instr list -> Target.instr list
    type result = Target.program

    fun lit n
        = (fn is => (Target.PUSH n) :: is)
    fun plus (f1, f2)
        = (fn is => f1 (f2 (Target.ADD :: is)))
    fun var n
        = (fn is => (Target.ACCESS n) :: is)
    fun lam f
        = (fn is => (Target.CLOSE (f nil)) :: is)
    fun app (f1, f2)
        = (fn is => f1 (f2 (Target.CALL :: is)))
    fun compute f
        = f nil
  end

structure Source_cmp2
= Make_source_processor (structure P = Source_parameters_cogen2)
```

For example, applying `Source_cmp2.process` to the source program

```
APP (APP (LAM ("a", LAM ("b", VAR "a")), LIT 10), LIT 20)
```

yields the following target program

```
[CLOSE [CLOSE [ACCESS 1]], PUSH 10, CALL, PUSH 20, CALL]
```

3.4 Interpretation and compilation for the target language

We now turn to defining an interpreter and a compiler for SECD machine code. As already mentioned, for clarity, we refrain from factoring the two definitions through an ML functor. Instead, we present each of them on its own.

3.4.1 An evaluation instantiation: target interpretation

The interpreter for the target language is a scaled-down version of Henderson's interpreter [35], which is itself an implementation of the SECD machine [46,53]. As before, given two ML values `e : 'a list` and `i : int`, applying `Index.fetch` to `e` and `i` fetches the corresponding entry in `e`.

```
structure Target_int
= struct
    datatype value = INT of int
                   | CLOSURE of Target.instr list * value list

    (*  process : Target.program -> value option  *)
    fun process p
        = let fun walk (v :: nil, e, nil, nil)
                  = SOME v
                | walk (v :: nil, e, nil, (s', e', c') :: d)
                  = walk (v :: s', e', c', d)
                | walk (s, e, (Target.PUSH n) :: c, d)
                  = walk ((INT n) :: s, e, c, d)
                | walk ((INT n2) :: (INT n1) :: s, e, Target.ADD :: c, d)
                  = walk ((INT (n1 + n2)) :: s, e, c, d)
                | walk (s, e, (Target.ACCESS i) :: c, d)
                  = walk ((Index.fetch (e, i)) :: s, e, c, d)
                | walk (s, e, (Target.CLOSE c') :: c, d)
                  = walk ((CLOSURE (c', e)) :: s, e, c, d)
                | walk (a :: (CLOSURE (c', e')) :: s, e, Target.CALL :: c, d)
                  = walk (nil, a :: e', c', (s, e, c) :: d)
                | walk (_, _, _, _)
                  = NONE
          in walk (nil, nil, p, nil)
          end
  end
```

For example, applying `Target_int.process` to the target program

```
[CLOSE [CLOSE [ACCESS 1]], PUSH 10, CALL, PUSH 20, CALL]
```

yields the optional value `SOME (INT 10)`.

3.4.2 A code-generation instantiation: target compilation

We obtain a compiler for the target language by instrumenting the SECD machine to build source terms (on the stack) instead of calculating values. Fresh identifiers are needed to construct residual λ-abstractions, and we obtain them by threading an integer.

```
structure Target_cmp
= struct
    type value = Source.term

    (*  process : Target.program -> value option  *)
    fun process p
        = let fun walk (v :: nil, e, nil, nil, g)
                    = SOME v
                  | walk (t :: nil, x :: e, nil, (s', e', c') :: d, g)
                    = walk (Source.LAM (x, t) :: s', e', c', d, g)
                  | walk (s, e, (Target.PUSH n) :: c, d, g)
                    = walk ((Source.LIT n) :: s, e, c, d, g)
                  | walk (t2 :: t1 :: s, e, Target.ADD :: c, d, g)
                    = walk ((Source.PLUS (t1, t2)) :: s, e, c, d, g)
                  | walk (s, e, (Target.ACCESS i) :: c, d, g)
                    = walk ((Source.VAR (Index.fetch (e, i))) :: s, e, c, d, g)
                  | walk (s, e, (Target.CLOSE c') :: c, d, g)
                    = let val x = "x" ^ Int.toString g
                      in walk (nil, x :: e, c', (s, e, c) :: d, g+1)
                      end
                  | walk (t1 :: t0 :: s, e, Target.CALL :: c, d, g)
                    = walk ((Source.APP (t0, t1)) :: s, e, c, d, g)
                  | walk (_, _, _, _, _)
                    = NONE
          in walk (nil, nil, p, nil, 0)
          end
  end
```

- `PUSH n` and `ADD`: Pushing a number and adding two numbers implement the binding-time shift between an interpreter and a compiler: instead of treating the integers numerically, we treat them symbolically.

- `CALL`: Both the function and the argument occur on the stack; we construct the corresponding residual application and we store it on the stack.

- `CLOSE c'`: We residualize `c'` into the body of a λ-abstraction in an environment with a fresh identifier `x`. When residualization completes (second clause in the definition of `walk`), `x` is available in the environment to manufacture the complete λ-abstraction, which we store on the stack.

For example, applying `Target_cmp.process` to the target program

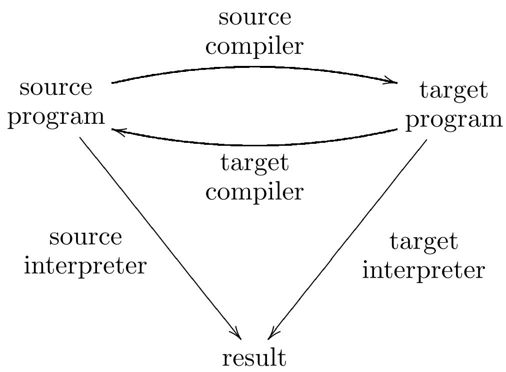
```
[CLOSE [CLOSE [ACCESS 1]], PUSH 10, CALL, PUSH 20, CALL]
```

yields the following optional source program:

```
SOME (APP (APP (LAM ("x0", LAM ("x1", VAR "x0")), LIT 10), LIT 20))
```

3.5 Properties

As in Section 2.5, we successively consider the total correctness of the source compiler with respect to the source interpreter and the target interpreter, the partial correctness of the target compiler with respect to the target interpreter and the source interpreter, and the left inverseness of the source compiler and of the target compiler:



In particular, we prove that the target compiler is a left inverse of the source compiler and therefore that it is a decompiler.

Terminology and notation:

- `Source_int.process` is the `process` function of the functor `Make_source_processor` of Section 3.1.1 instantiated with the structure `Source_parameters_std` of Section 3.3.1. We refer to the corresponding `walk` function as `w_s_int` (for "walk function of the source interpreter").

- `Source_cmp2.process` is the `process` function of the functor `Make_source_processor` of Section 3.1.1 instantiated with the structure `Source_parameters_cogen2` of Section 3.3.3. We refer to the corresponding `walk` function as `w_s_cmp` (for "walk function of the source compiler").

- `Target_int.process` is the `process` function of the structure `Target_int` of Section 3.4.1. We refer to the corresponding `walk` function as `w_t_int` (for "walk function of the target interpreter").

- `Target_cmp.process` is the `process` function of the structure `Target_cmp` of Section 3.4.2. We refer to the corresponding `walk` function as `w_t_cmp` (for "walk function of the target compiler").

Among the functions above, `Source_cmp2.process` (and thus `w_s_cmp`) and `Target_cmp.process` (and thus `w_t_cmp`) are pure and total. They are pure because they have no side effects, and they terminate because they recursively traverse finite source and target programs.

As in Section 2.5, we reason equationally on the ML syntax of the interpreters and compilers, using observational equivalence.

3.5.1 Total correctness of the source compiler

Theorem 3 *For all ML values* `sp : Source.program`*, the following observational equivalence holds:*

```
Target_int.process (Source_cmp2.process sp)
≅  SOME (Source_int.process sp).
```

The proof of this theorem (i.e., of the correctness of Henderson's compiler for the SECD machine) is more involved than the proof of Theorem 1 and is beyond the scope of the present article. Therefore we omit it.

3.5.2 Partial correctness of the target compiler

The situation is the same as in Section 2.5.2, i.e., not all values of type `Target.program` are well-formed programs, as indicated by the option type in Section 3.4.1 and Section 3.4.2. As in Section 2.5.2, we leave the issue of partial correctness aside, and instead we turn to proving that the target compiler is a left inverse of the source compiler.

3.5.3 Left inverseness

In this section we prove that `Target_cmp.process` is a left inverse of `Source_cmp2.process` modulo α-renaming. Our proof uses structural induction on source terms, and therefore we need to treat open terms together with their environment:

- the environment of a term, in the source compiler, is a list of identifiers;
- the environment of a term, in the target compiler, is a list of identifiers, all distinct.

We therefore relate the terms together with their environments as follows.

Definition 1 (Left equivalence) *For all ML values* `t : Source.term`, `xs : Source.ide list` *containing the identifiers free in* `t` *in reverse order of their declaration,* `t' : Source.term`*, and* `e : Source.ide list` *with the same length as* `xs` *and containing distinct identifiers, we say that* `t` *and* `t'` *are* left-equivalent with respect to `xs` and `e` *whenever the relation*

$$\langle \texttt{t}, \texttt{xs} \rangle \approx \langle \texttt{t'}, \texttt{e} \rangle$$

is satisfied. This relation is defined inductively as follows:

$$\frac{\texttt{n} \cong \texttt{n'}}{\langle \texttt{LIT n}, \texttt{xs} \rangle \approx \langle \texttt{LIT n'}, \texttt{e} \rangle}$$

$$\frac{\langle \texttt{t1}, \texttt{xs} \rangle \approx \langle \texttt{t1'}, \texttt{e} \rangle \quad \langle \texttt{t2}, \texttt{xs} \rangle \approx \langle \texttt{t2'}, \texttt{e} \rangle}{\langle \texttt{PLUS (t1, t2)}, \texttt{xs} \rangle \approx \langle \texttt{PLUS (t1', t2')}, \texttt{e} \rangle}$$

$$\frac{\texttt{Index.fetch (e, Index.establish (x, xs))} \cong \texttt{x'}}{\langle \texttt{VAR x}, \texttt{xs} \rangle \approx \langle \texttt{VAR x'}, \texttt{e} \rangle}$$

$$\frac{\langle \texttt{t}, \texttt{x :: xs} \rangle \approx \langle \texttt{t'}, \texttt{x' :: e} \rangle}{\langle \texttt{LAM (x, t)}, \texttt{xs} \rangle \approx \langle \texttt{LAM (x', t')}, \texttt{e} \rangle}$$

$$\frac{\langle \texttt{t0}, \texttt{xs} \rangle \approx \langle \texttt{t0'}, \texttt{e} \rangle \quad \langle \texttt{t1}, \texttt{xs} \rangle \approx \langle \texttt{t1'}, \texttt{e} \rangle}{\langle \texttt{APP (t0, t1)}, \texttt{xs} \rangle \approx \langle \texttt{APP (t0', t1')}, \texttt{e} \rangle}$$

For closed terms that contain no λ-abstractions, left equivalence reduces to structural equality. For all closed terms, left equivalence implies α-equivalence.

In Lemma 3 and Theorem 4, we use left equivalence to establish left inverseness.

Lemma 3 *For all ML values* `t : Source.term`*,* `xs : Source.ide list` *containing all the identifiers free in* `t`*,* `e : Source.ide list` *with the same length as* `xs` *and containing fresh (and all distinct) identifiers,* `s : Source.term list`*,* `c : Target.instr list`*,* `d : (Source.term list * Source.ide list * Target.instr list) list`*, and* `g : int`*, there exist two ML values* `t' : Source.term` *and* `g' : int` *such that the following conjunction holds:*

$$\langle \texttt{t}, \texttt{xs} \rangle \approx \langle \texttt{t'}, \texttt{e} \rangle \ \wedge\ \texttt{w\_t\_cmp (s, e, w\_s\_cmp t xs c, d, g)}$$
$$\cong \texttt{w\_t\_cmp (t' :: s, e, c, d, g')}.$$

Proof: The proof is by structural induction on the source syntax.

Base case: `LIT n`

For all ML values `xs : Source.ide list`, `e : Source.ide list` with the same length as `xs` and containing fresh (and all distinct) identifiers, `s : Source.term list`, `c : Target.instr list`, `d : (Source.term list * Source.ide list * Target.instr list) list`, and `g : int`, we want to show that the following conjunction holds:

$$\langle \texttt{LIT n}, \texttt{xs} \rangle \approx \langle \texttt{LIT n}, \texttt{e} \rangle \ \wedge\ \texttt{w\_t\_cmp (s, e, w\_s\_cmp (LIT n) xs c, d, g)}$$
$$\cong \texttt{w\_t\_cmp ((LIT n) :: s, e, c, d, g')}$$

for some ML value `g' : int`.

The left conjunct holds by definition of $\approx$. As for the right conjunct,

`w_t_cmp (s, e, w_s_cmp (LIT n) xs c, d, g)`
$\cong$ (unfolding `w_s_cmp`)
`w_t_cmp (s, e, Source_parameters_cogen2.lit n c, d, g)`
$\cong$ (unfolding `Source_parameters_cogen2.lit`)
`w_t_cmp (s, e, (fn is => (PUSH n) :: is) c, d, g)`
$\cong$ (function application)
`w_t_cmp (s, e, (PUSH n) :: c, d, g)`
$\cong$ (unfolding `w_t_cmp`)
`w_t_cmp ((LIT n) :: s, e, c, d, g)`

Induction case: `PLUS (t1, t2)`

For all ML values `xs : Source.ide list` containing all the identifiers free in `t1` and `t2`, `e : Source.ide list` with the same length as `xs` and containing fresh (and all distinct) identifiers, `s : Source.term list`, `c : Target.instr list`, `d : (Source.term list * Source.ide list * Target.instr list) list`, and `g : int`, we want to show that the following conjunction holds:

```
             ⟨PLUS (t1, t2), xs⟩ ≈ ⟨PLUS (t1', t2'), e⟩
                                  ∧
w_t_cmp (s, e, w_s_cmp (PLUS (t1, t2)) xs c, d, g)
≅ w_t_cmp ((PLUS (t1', t2')) :: s, e, c, d, g'')
```

for some ML value `g'' : int` and for all ML values `t1 : Source.term` and `t1' : Source.term` satisfying the induction hypothesis and for all ML values `t2 : Source.term` and `t2' : Source.term` satisfying the induction hypothesis.

The left conjunct holds because of the induction hypotheses and by definition of $\approx$. As for the right conjunct,

```
w_t_cmp (s, e, w_s_cmp (PLUS (t1, t2)) xs c, d, g)
≅ (unfolding w_s_cmp)
w_t_cmp (s, e, w_s_cmp (Source_parameters_cogen.plus
                        (w_s_cmp t1 xs, w_s_cmp t2 xs) c, d, g)
≅ (unfolding Source_parameters_cogen2.plus)
w_t_cmp (s, e, (fn is =>
                w_s_cmp t1 xs (w_s_cmp t2 xs (ADD :: is))) c, d, g)
≅ (function application)
w_t_cmp (s, e, w_s_cmp t1 xs (w_s_cmp t2 xs (ADD :: c)), d, g)
≅ (induction hypothesis on t1, for some ML value g' : int)
w_t_cmp (t1' :: s, e, w_s_cmp t2 xs (ADD :: c), d, g')
≅ (induction hypothesis on t2, for some ML value g'' : int)
w_t_cmp (t2' :: t1' :: s, e, ADD :: c, d, g'')
≅ (unfolding w_t_cmp)
w_t_cmp ((PLUS (t1', t2')) :: s, e, c, d, g'')
```

Base case: `VAR x`

For all ML values `xs : Source.ide list` containing `x`, `e : Source.ide list` with the same length as `xs` and containing fresh (and all distinct) identifiers, `s : Source.term list`, `c : Target.instr list`, `d : (Source.term list * Source.ide list * Target.instr list) list`, and `g : int` we want to show that the following conjunction holds:

```
⟨VAR x, xs⟩ ≈ ⟨VAR x', e⟩ ∧ w_t_cmp (s, e, w_s_cmp (VAR x) xs c, d, g)
                         ≅ w_t_cmp ((VAR x') :: s, e, c, d, g')
```

for some ML values `x' : Source.ide` and `g' : int`.

We reason equationally:

```
w_t_cmp (s, e, w_s_cmp (VAR x) xs c, d, g)
≅ (unfolding w_s_cmp)
w_t_cmp (s, e, Source_parameters_cogen.var
                 (Index.establish (x, xs)) c, d, g)
≅ (unfolding Source_parameters_cogen2.var)
w_t_cmp (s, e, (fn is =>
                (ACCESS (Index.establish (x, xs))) :: is) c, d, g)
≅ (function application)
w_t_cmp (s, e, ((ACCESS (Index.establish (x, xs))) :: c), d, g)
≅ (unfolding w_t_cmp)
w_t_cmp ((VAR (Index.fetch (e, Index.establish (x, xs)))) :: s, e, c,
d, g)
```

There are no unbound identifiers in source programs and by assumption all identifiers are accounted for in `xs`. Since `xs` and `e` have the same length, there exists an ML value `x' : Source.ide` in `e` satisfying

$$\texttt{Index.fetch (e, Index.establish (x, xs))} \cong \texttt{x'}$$

Given this `x'`, by definition of $\approx$,

$$\langle \texttt{VAR x}, \texttt{xs} \rangle \approx \langle \texttt{VAR x'}, \texttt{e} \rangle$$

holds and furthermore the following observational equality holds:

```
w_t_cmp ((VAR (Index.fetch (e, Index.establish (x, xs)))) :: s, e,
c, d, g)
≅ w_t_cmp ((VAR x') :: s, e, c, d, g)
```

Induction case: `LAM (x, t)`

For all ML values `xs : Source.ide list` containing all the identifiers free in `t`, `e : Source.ide list` with the same length as `xs` and containing fresh (and all distinct) identifiers, `s : Source.term list`, `c : Target.instr list`, `d : (Source.term list * Source.ide list * Target.instr list) list`, and `g : int` we want to show that the following conjunction holds:

$$\langle \texttt{LAM (x, t)}, \texttt{xs} \rangle \approx \langle \texttt{LAM (x', t')}, \texttt{e} \rangle$$
$$\wedge$$

```
w_t_cmp (s, e, w_s_cmp (LAM (x, t)) xs c, d, g)
≅ w_t_cmp (LAM (x', t') :: s, e, c, d, g')
```

for some ML value `g' : int` and for all ML values `t : Source.term` and `t' : Source.term` satisfying the induction hypothesis.

We reason equationally:

```
w_t_cmp (s, e, w_s_cmp (LAM (x, t)) xs c, d, g)
≅ (unfolding w_s_cmp)
w_t_cmp (s, e, Source_parameters_cogen2.lam (w_s_cmp t (x :: xs)) c,
d, g)
≅ (unfolding Source_parameters_cogen2.lam)
w_t_cmp (s, e, (fn is =>
                (CLOSE (w_s_cmp t (x :: xs) nil)) :: is) c, d, g)
≅ (function application)
w_t_cmp (s, e, (CLOSE t (w_s_cmp (x :: xs) nil)) :: c, d, g)
≅ (unfolding w_t_cmp)
w_t_cmp (nil, x' :: e, w_s_cmp t (x :: xs) nil, (s, e, c) :: d, g+1)
where x' = "x" ^ Int.toString g and is a fresh identifier.
≅ (induction hypothesis on t since xs' and e' have the same length,
   for some ML value t' : Source.term satisfying
   ⟨t, x :: xs⟩ ≈ ⟨t', x' :: e⟩ for some ML value g' : int)
w_t_cmp (t' :: nil, x' :: e, nil, (s, e, c) :: d, g')
≅ (unfolding w_t_cmp)
w_t_cmp (LAM (x', t') :: s, e, c, d, g')
```

By induction hypothesis on `t`, ⟨`t`, `x :: xs`⟩ $\approx$ ⟨`t'`, `x' :: e`⟩ holds, and therefore ⟨`LAM (x, t)`, `x :: xs`⟩ $\approx$ ⟨`LAM (x', t')`, `x' :: e`⟩ also holds, by definition of $\approx$.

Induction case: `APP (t0, t1)`

This case is similar to the `PLUS` case above.

□

Theorem 4 *For each ML value* `sp : Source.program`*, there exists an ML value* `sp' : Source.program` *that is α-equivalent to* `sp` *and that satisfies the following observational equivalence:*

```
Target_cmp.process (Source_cmp2.process sp)  ≅  SOME sp'.
```

Proof: For all ML values `sp : Source.program` and `tp : Target.program`, the following observational equivalences hold:

```
Source_cmp2.process sp
≅ (unfolding Source_cmp2.process)
Source_cmp2.compute (w_s_cmp sp nil)
≅ (unfolding Source_cmp2.compute)
w_s_cmp sp nil nil
```

```
Target_cmp.process tp
≅ (unfolding Target_cmp.process)
w_t_cmp (nil, nil, tp, nil, 0)
```

For all ML values `sp : Source.program`, we therefore have to prove the following observational equivalence:

```
w_t_cmp (nil, nil, w_s_cmp sp nil nil, nil, 0)  ≅  SOME sp'
```

for a program `sp'` that is α-equivalent to `sp`. This observational equivalence, however, follows from Lemma 3. Indeed, for all ML values `t : Source.term` that are closed `nil : Source.ide list`, `nil : Source.term list`, `nil : Target.instr list`, `nil : (Source.term list * Source.ide list * Target.instr list) list`, and `0 : int`, Lemma 3 reads as

```
⟨t,nil⟩ ≈ ⟨t',nil⟩ ∧ w_t_cmp (nil, nil, w_s_cmp t nil nil, nil, 0)
                    ≅ w_t_cmp (t' :: nil, nil, nil, nil, g')
```

for some ML values `t' : Source.term` and `g' : int`. Therefore `t` and `t'` are left-equivalent. Since `t` is a closed term, `t'` is a closed term too, i.e., a program. Since they are left-equivalent, they are also α-equivalent.

Finally,

```
w_t_cmp (t' :: nil, nil, nil, nil, g')
≅ (unfolding of w_t_cmp)
SOME t'
```

and the result follows. □

3.6 Summary

We have shown that the symmetric approach to compilation and decompilation scales to the λ-calculus and the SECD-machine language. We have not seen this approach to decompilation described elsewhere. For example, specific efforts have been dedicated to decompiling terms for abstract machines in the literature, independently of interpreting them and of compiling them [32, 34].

4 Related work

This section situates our symmetric approach to compilation and decompilation with respect to compilation, decompilation, partial evaluation, and parsing.

4.1 Compilation and decompilation

We consider in turn the construction, correctness, and derivation of compilers and decompilers.

4.1.1 Construction

Compilation and decompilation technologies have been around for over five decades. While many authors note that compilation is an inverse of decompilation [17, 33], in practice these technologies have evolved independently.

The area of compilation is well established and well mapped today, with a number of subdivisions—e.g., syntactic analysis, semantic analysis, and code generation. In contrast, the area of decompilation is not in the main stream and it is less well defined and less well-mapped. The general problem of decompilation is known to be unsolvable [17,28,36,37], or requiring "human", i.e., "manual" intervention.

In general, writing a real decompiler is an engineering challenge which is documented comprehensively at ⟨`http://www.program-transformation.org/twiki/bin/view/Transform/DeCompilation`⟩.

In an imperative setting, the strategy is to establish control-flow and data-flow graphs to build high-level constructs [17]. For an example closer to our work here, Proebsting and Watterson decompile Java expressions by symbolically executing JVM instructions [57].

In a logical setting, decompiling by executing compiled programs is a standard technique that directly builds on a relational specification such as the one in Section 4.1.2 [12, 15].

4.1.2 Correctness

In their work on the lambda-sigma calculus [34], Hardin, Maranget, and Pagano consider a compiler to Cardelli's functional abstract machine and the corresponding compiler, and they prove an inverseness property. Similarly, in their work on strong reduction [32], Grégoire and Leroy also consider a compiler and the corresponding decompiler. We are not aware of any other work addressing inverseness properties for a compiler and a decompiler. Also, we are aware of only few semantic approaches to decompilation, including Mycroft's type-based strategy and Katsumata and Ohori's proof-directed strategy [45, 54, 55].

Since McCarthy and Painter's first correctness proof of a compiler [50], correctness proofs for compilers typically use structural induction on the source syntax. Alternatively to defining two functions `Source_cmp.process` and `Target_cmp.process`, however, one can define a relation $\sim$ between source and target programs. For example, for the arithmetic expressions of Section 2, one can define the following relation between source expressions and lists of target instructions:

$$\frac{}{\texttt{Source.LIT n} \sim \texttt{[Target.PUSH n]}}$$

$$\frac{\texttt{e1} \sim \texttt{is1} \qquad \texttt{e2} \sim \texttt{is2}}{\texttt{Source.PLUS (e1, e2)} \sim \texttt{is1 @ is2 @ [Target.ADD]}}$$

This specification is the relational counterpart of the compiler of Section 2.3.2 and proving properties about it is done relationally.

In general, compilation and decompilation form yet another example of Galois connections in computer science, as outlined by Melton, Schmidt, and Strecker [52]. Indeed in general the image of each transformation is a sublanguage over which the composition of the two transformations acts as the identity, whereas it acts as a normalizer for programs in the annulus. The two examples presented here do not illustrate this normalization, but adding let expressions to the arithmetic expressions of Section 2 is enough to make it appear: target programs are decompiled into source programs where all the let expressions have been lifted out. More concretely, if the source language is

```
datatype exp = LIT of int
             | PLUS of exp * exp
             | LET of ide * exp * exp
             | VAR of ide
```

then the source sublanguage of normal forms is

```
datatype operation_nf = LIT_nf of int
                      | VAR_nf of ide
                      | PLUS_nf of operation_nf * operation_nf

datatype exp_nf = LET_nf of ide * operation_nf * exp_nf
                | BODY_nf of operation_nf
```

Another way to illustrate normalization by compilation and decompilation is to consider an optimizing compiler—e.g., one that includes constant propagation, constant folding, and common sub-expression elimination. In principle, the decompiler yields a correspondingly optimized source program, if one is expressible in the source language. The issue then is that of completeness. The phenomenon could be referred to as *normalization by staged evaluation*, in reference to normalization by evaluation [5, 9, 21–24, 30].

4.1.3 Derivation

Much literature has been devoted to deriving a compiler from an interpreter, up to and including undergraduate textbooks [1,25]. We single out Morris's 700 follow-up paper for its observation that massaging a λ-interpreter can yield a compiler for the SECD machine [53] and Wand's article *Deriving target code as a representation of continuation semantics* for its compelling title that precisely characterizes the binding-time shift of going from evaluation to code generation [60].

In principle decompilation could be achieved by program inversion over a compiler [2, 29]. Abramov, Glück, and Klimov have recently reported ongoing efforts in this direction [3].

Our work is a study of a simultaneous derivation of a compiler and decompiler. The two are related by left-inverseness properties (Theorems 2 and 4). The relations between the compiler, the decompiler, and the two interpreters for the source and target languages are given by the the standard commuting diagram displayed in Section 1.

4.2 Partial evaluation

In some sense, we are doing offline partial evaluation by hand. In particular, the factorizations into functors and structures of our language processors manifest a binding-time separation between the static (compile-time) and the dynamic (run-time) components of the language—what Lee refers to as macro- and micro-semantics [47] and as identified by Jones and Muchnick [40]. Given an unfactorized language processor, the binding-time analysis of an offline partial evaluator could achieve this binding-time division provided the language processor is well-written [38]. Specialization then corresponds to the instantiation of a functor with a code-generating structure.

Specializing interpreters is a popular application of partial evaluation, one that was discovered by Futamura in the early 1970s [26, 27].

4.2.1 The first Futamura projection for compiling

Given an interpreter for a defined language written in a defining language and given a program written in the defined language, specializing the interpreter with respect to the program gives a residual program written in the defining language. In conjunction with a self-applicable partial evaluator, the first Futamura projection has been a major source of inspiration in the area of partial evaluation [39, 43].

In practice, specializing an interpreter with respect to a program yields a residual program that includes all the idiosyncrasies of the interpreter. For example, the residual program shown in Futamura's original article reveals that his interpreter represents environments as association lists [26, page 390]. Against this backdrop, the notion of Jones-optimality has been developed [48, 58].

4.2.2 The first Futamura projection for decompiling

In principle, the first Futamura projection directly applies for decompiling, given an interpreter for a target language written in a source language and given a program written in the target language. In practice, specializing this interpreter with respect to this program does give a residual program written in the source language but this residual program in general includes all the idiosyncrasies of the interpreter. In that sense, decompiling using the first Futamura projection is far from Jones-optimal.

In contrast, doing partial evaluation by hand as we do here gives us some extra flexibility regarding the target language in which to express residual programs, up to the point of left inverseness. For symmetry, it seems logical to refer to our methodology as a *Futamura embedding*.

4.3 Parsing

In some sense, and as agreed upon in the decompilation community, decompiling arithmetic expressions in reverse Polish form is akin to parsing [10]. More generally, a parser generator such as Yacc makes it possible to generate a compiler as well as an interpreter. A Yacc user parameterizes the core parsing engine by semantic actions, and these semantic actions can either carry out computations and construct intermediate results or they can build abstract-syntax trees. In that sense, we could use Yacc to decompile and to interpret arithmetic expressions in reverse Polish notation and also to decompile and to interpret programs for the SECD machine.

5 Conclusion

At the heart of turning an interpreter into a (front-end) compiler, there is a binding-time shift: Where the interpreter performs an evaluation, the compiler emits code representing this evaluation. In this article, we have shown how this binding-time shift can be used not only to construct a compiler from a source language to a target language but also to construct a compiler from a target language to a source language. We have treated two examples and we have proven that in each case the target compiler is a left inverse of the source compiler—i.e., formally, that the target compiler is a decompiler.

The source languages we have considered are a canonical language of expressions and its functional extension, the λ-calculus. Independently, we have also considered several other languages of expressions:

- a source language of boolean expressions, a target language of conditional expressions, and a compiler that implements short-cut boolean evaluation;
- a language of expressions with block structure and the language of a register-stack machine, as in Section 4.1.2; and
- another abstract machine for the λ-calculus.

In each case, we were able to apply the methodology of specifying each language processor with a functor implementing the corresponding fold function and instantiating this functor into an interpreter (with elementary evaluation functions) or a compiler (with elementary code-generation functions). To this end, we took advantage of the correspondence between source expressions and expressible values in functional languages. For imperative languages, however, it seems unavoidable to use some form of control-flow graph, as in traditional decompilation. At any rate, the methodology is not an end in itself; we see it as a systematic means to explore the binding-time shift between an interpreter and a (de)compiler.

Acknowledgments: We are grateful to Neil Jones for his generous inspiration, both direct and indirect, which includes coining the term 'binding-time shift'. This article has also benefited from comments by anonymous reviewers, by the editors of the Festschrift, and by Kenichi Asai, Robert Glück, Julia L. Lawall, Karoline Malmkjær, Lasse R. Nielsen, Henning Korsholm Rohde, and Ulrik P. Schultz. Thanks are also due to Jon Erickson, from Dr. Dobb's Journal, for a timely communication with the second author in March 2002.

This work was partly carried out while the third author was visiting BRICS.

A Factorized version of the processor for the SECD-machine language

```
signature TARGET_PROCESSOR
= sig
    type value

    val push : int -> value
    val add : value * value -> value
    val lookup : value list * int -> value
    val close : (value * (value list * value list * Target.instr list) list
                 -> value) -> value
    val call : value * value * (value -> value) -> value
  end

functor Make_target_processor (structure S : TARGET_PROCESSOR)
= struct local open Target in
    exception CORE_DUMPED

    fun process p
        = let fun exec (v :: nil) e nil nil
                  = v
                | exec (v :: nil) e nil ((s', e', c') :: d)
                  = exec (v :: s') e' c' d
                | exec s e ((PUSH n) :: c) d
                  = exec ((S.push n) :: s) e c d
                | exec (n1 :: n2 :: s) e (ADD :: c) d
                  = exec ((S.add (n1, n2)) :: s) e c d
                | exec s e ((ACCESS i) :: c) d
                  = exec ((S.lookup (e, i)) :: s) e c d
                | exec s e ((CLOSE c') :: c) d
                  = exec ((S.close (fn (a, d)
                                  => (exec nil (a :: e) c' d))) :: s) e c d
                | exec (a :: f :: s) e (CALL :: c) d
                  = S.call (f, a, fn r => exec (r :: s) e c d)
                | exec _ _ _ _
                  = raise CORE_DUMPED
          in exec nil nil p nil
          end
  end end
```

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 1985.

[2] Sergei M. Abramov and Robert Glück. The universal resolving algorithm: inverse computation in a functional language. In Roland Backhouse and José N. Oliveira, editors, *Mathematics of Program Construction. Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 187–212. Springer-Verlag, 2000.

[3] Sergei M. Abramov, Robert Glück, and Yury Klimov. Faster answers and improved termination in inverse computation of non-flat languages. Technical report, Program Systems Institut, Russian Academy of Sciences, Pereslavl-Zalessky, March 2002 2002.

[4] Mads Sig Ager, Olivier Danvy, and Mayer Goldberg. A symmetric approach to compilation and decompilation (extended version). Technical Report BRICS RS-02-37, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, August 2002.

[5] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip Scott. Normalization by evaluation for typed lambda calculus with coproducts. In Joseph Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 203–210, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[6] Roland Carl Backhouse, Patrik Jansson, Johan Jeuring, and Lambert G. L. T. Meertens. Generic programming: an introduction. In S. Doaitse Swierstra, Pedro Rangel Henriques, and José N. Oliveira, editors, *Advanced functional programming, Third International School*, number 1608 in Lecture Notes in Computer Science, pages 28–115, Braga, Portugal, September 1998. Springer-Verlag.

[7] Guntis Barzdins. Mixed computation and compiler basis. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 15–26. North-Holland, 1988.

[8] Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 4–12, San Antonio, Texas, January 1999. Available online at `http://www.brics.dk/~pepm99/programme.html`.

[9] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.

[10] M. N. Bert and L. Petrone. Decompiling context-free languages from their Polish-like representations. *Calcolo*, XIX(1):35–57, March 1982.

[11] Anders Bondorf. Compiling laziness by partial evaluation. In Simon L. Peyton Jones, Guy Hutton, and Carsten K. Holst, editors, *Functional Programming, Glasgow 1990*, Workshops in Computing, pages 9–22, Glasgow, Scotland, 1990. Springer-Verlag.

[12] Jonathan Bowen. From programs to object code and back again using logic programming: Compilation and decompilation. *Journal of Software Maintenance: Research and Practice*, 5(4):205–234, 1994.

[13] Peter J. Brown. Re-creation of source code from reverse Polish. *Software—Practice and Experience*, 2:275–278, 1972.

[14] Peter J. Brown. More on the re-creation of source code from reverse Polish. *Software—Practice and Experience*, 7(8):545–551, 1977.

[15] Kevin A. Buettner. Fast decompilation of compiled Prolog clauses. In Ehud Y. Shapiro, editor, *Proceedings of the third international conference on logic programming*, number 225 in Lecture Notes in Computer Science, pages 663–670, London, United Kingdom, July 1986. Springer-Verlag.

[16] Rod M. Burstall and Peter J. Landin. Programs and their proofs: An algebraic approach. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 17–43. Edinburgh University Press, 1969.

[17] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, July 1994.

[18] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[19] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[20] Charles Consel and Siau-Cheng Khoo. Semantics-directed generation of a Prolog compiler. *Science of Computer Programming*, 21:263–291, 1993.

[21] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[22] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[23] Olivier Danvy, Morten Rhiger, and Kristoffer Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.

[24] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

[25] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill, 1991.

[26] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[27] Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

[28] R. Stockton Gaines. On the translation of machine language programs. *Communications of the ACM*, 8(12):736–741, 1965.

[29] Robert Glück and Andrei V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R. Trappl, editor, *Cybernetics and Systems Research '94*, volume 2, pages 1563–1570, Singapore, 1994. World Scientific.

[30] Mayer Goldberg. Gödelization in the λ-calculus. *Information Processing Letters*, 75(1-2):13–16, 2000.

[31] Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

[32] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, Pittsburgh, Pennsylvania, September 2002. ACM Press. To appear.

[33] Maurice H. Halstead. *Machine Independent Computer Programming.* Spartan Books, Washington, D.C., 1962.

[34] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[35] Peter Henderson. *Functional Programming – Application and Implementation*. Prentice-Hall International, 1980.

[36] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[37] Barron C. Housel and Maurice H. Halstead. A methodology for machine language decompilation. In *Proceedings of the 27th ACM Annual Conference*, pages 254–260, 1974.

[38] Neil D. Jones. What *not* to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 216–237, Dagstuhl, Germany, February 1996. Springer-Verlag.

[39] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.

[40] Neil D. Jones and Steven S. Muchnick. Some thoughts towards the design of an ideal language. In Susan L. Graham, editor, *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 77–94. ACM Press, January 1976.

[41] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *The Handbook of Logic in Computer Science*. North-Holland, 1992.

[42] Neil D. Jones and David A. Schmidt. Compiler generation from denotational semantics. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, pages 70–93, Aarhus, Denmark, 1980. Springer-Verlag.

[43] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[44] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 258–268, Albuquerque, New Mexico, January 1992. ACM Press.

[45] Shin-ya Katsumata and Atsushi Ohori. Proof-directed de-compilation of low-level code. In David Sands, editor, *Proceedings of the Tenth European Symposium on Programming*, number 2028 in Lecture Notes in Computer Science, pages 352–366, Genova, Italy, April 2001. Springer-Verlag.

[46] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[47] Peter Lee. *Realistic Compiler Generation*. The MIT Press, 1989.

[48] Henning Makholm. On Jones-optimal specialization for strongly typed languages. In Walid Taha, editor, *Proceedings of the First Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, number 1924 in Lecture Notes in Computer Science, pages 129–148, Montréal, Canada, September 2000. Springer-Verlag.

[49] William May. A simple decompiler – recreating source code without token resistance. *Dr. Dobb's Journal*, 50:50–52, June 1988.

[50] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings of the 1967 Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, Providence, Rhode Island, 1967.

[51] Erik Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, Nijmegen, The Netherlands, 1992.

[52] Austin Melton, David A. Schmidt, and George Strecker. Galois connections and computer science applications. In David H. Pitt et al., editors, *Category Theory and Computer Programming*, number 240 in Lecture Notes in Computer Science, pages 299–312, Guildford, UK, September 1986. Springer-Verlag.

[53] Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993.

[54] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 208–223, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[55] Alan Mycroft, Shin-ya Katsumata, and Atsushi Ohori. Comparing type-based and proof-directed decompilation. In Peter Aiken and Elizabeth Burd, editors, *Proceedings of the Working Conference on Reverse Engineering*, Stuttgart, Germany, 2001. `http://reengineer.org/wcre2001/`.

[56] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[57] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in Java (does bytecode reveal source?). In Steve Vinoski, editor, *Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 185–197, Portland, Oregon, June 1997. The USENIX Association.

[58] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects, Second Symposium, PADO 2001*, number 2053 in Lecture Notes in Computer Science, pages 257–275, Aarhus, Denmark, May 2001. Springer-Verlag.

[59] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[60] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

[61] Mitchell Wand. From interpreter to compiler: a representational derivation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 306–324, Copenhagen, Denmark, October 1985. Springer-Verlag.

[62] Zhe Yang. *Language Support for Program Generation: Reasoning, Implementation, and Applications*. PhD thesis, Computer Science Department, New York University, New York, New York, August 2001.

Recent BRICS Report Series Publications

RS-02-37 **Mads Sig Ager, Olivier Danvy, and Mayer Goldberg.** ***A Symmetric Approach to Compilation and Decompilation*****. August 2002. 45 pp. Appears in Mogensen, Schmidt and Sudborough, editors,** ***The Essence of Computation: Complexity, Analysis*****, Transformation, LNCS 2556, 2002, pages 296–331.**

RS-02-36 **Daniel Damian and Olivier Danvy.** ***CPS Transformation of Flow Information, Part II: Administrative Reductions*****. August 2002. 9 pp. To appear in the** ***Journal of Functional Programming*****. This report supersedes the earlier BRICS report RS-01-40.**

RS-02-35 **Patricia Bouyer.** ***Timed Automata May Cause Some Troubles*****. August 2002. 44 pp.**

RS-02-34 **Morten Rhiger.** ***A Foundation for Embedded Languages*****. August 2002. 29 pp.**

RS-02-33 **Vincent Balat and Olivier Danvy.** ***Memoization in Type-Directed Partial Evaluation*****. July 2002. 18 pp. To appear in Batory and Consel, editors,** ***ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*****, GPCE '02 Proceedings, LNCS, 2002.**

RS-02-32 **Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde.** ***On Obtaining Knuth, Morris, and Pratt's String Matcher by Partial Evaluation*****. July 2002. 43 pp. To appear in Chin, editor,** ***ACM SIGPLAN ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*****, ASIA-PEPM '02 Proceedings, 2002.**

RS-02-31 **Ulrich Kohlenbach and Paulo B. Oliva.** ***Proof Mining: A Systematic Way of Analysing Proofs in Mathematics*****. June 2002. 47 pp.**

RS-02-30 **Olivier Danvy and Ulrik P. Schultz.** ***Lambda-Lifting in Quadratic Time*****. June 2002.**

RS-02-29 **Christian N. S. Pedersen and Tejs Scharling.** ***Comparative Methods for Gene Structure Prediction in Homologous Sequences*****. June 2002. 20 pp.**