A Language with Jones-Optimal Interpretation having Program-Independent Overhead

Lars Hartmann, lars@hartmannix.dk

September 14, 2007

Abstract

By considering a succession of programming languages, we give an analysis of the features necessary to obtain the seemingly incompatible properties "program independent interpretation overhead (PIIO)" and "Jones optimal interpretation (JOI)" for a single programming language.

Based on this, we devise a language (called $\mathcal{IF}$) with PIIO that is extremely close to being Jones optimal. we give a range of experiments to support this claim.

The problems with "Jones Optimal Interpretation" for $\mathcal{IF}$ is identified and a solution provided, the remaining work for obtaining "Jones Optimal Interpretation" is small. Futhermore the problems identified offers new insight into the general understanding of "Jones Optimal Interpretation".

The report contains a large introductory part serving as a general technical introduction to the concepts of partial evaluation, to "Jones Optimality" and to "program independent interpretation overhead".

Contents

List of Figures

List of Tables

Preface

This work was done as part of my pursuit of a Masters degree in Computer Science at DIKU in the summer of 2007.

At the onset of the project I had worked a little bit with the language $\mathcal{I}$, and I new a little about the "program independent interpretation overhead" property of $\mathcal{I}$.

I had only basic theoretical knownledge of specialization and partial evaluation. The definition of the "Jones Optimal Interpretation" property of specialization was new to me, even though the concept of removing the interpretation overhead by specialization was vaguely familiar.

During the work on this project I have learned a lot about implementing specializers, and I can relate to some of the problems that must have caused a lot of trouble when first implementing a self applicable partial evaluator.

A large part of the work done experimented with various language features. The base language chosen is important, as these experiments requires many and frequent changes to the timed interpreter used. If the base language of the implementation is not suited for quick changes to syntax or semantics, the experiments would be next to impossible.

Implementing specializers is tricky business. Every once in a while it seems that you are very close to completing the task, everything seems fine, and then you design and implement one more test, which causes everything to fall apart. This happend more than once for me.

Good specialization is the product of many small separate tasks. It is worth noting that some specializers combine several of these tasks into one making it more difficult to see what happens. My first shot at a specializer for $\mathcal{I}$ tried to do everything at once, and it failed miserably.

Introduction

When figuring out how to solve a problem, two approaches can be used. One where we solve the given problem exactly as it is stated and one where we try to solve the class of problems that the given problem is an instance of.

Specialization is the process of adapting the general solution of the class of problems to a specific instance of the given class of problems, it is bridging the gap between the general and the specific solution.

Since the early days of computers, programmers have written programs in languages of higher level than the machine code the computers understand. These programs was hand translated in the beginning until someone thought of writing a translater for the programming language.

When studying programming languages a general way to become familiar with a specific language is to write an interpreter for it. Interpreters solve the general problem of executing a program in a given language. The interpreter is typically slow. Specializing an interpreter with a specific program could produce a new program with better execution time than the original one, it specializes a general solution, the interpreter, to a specific problem, the program.

This work searches for a language which have certain properties with regard to specialization and interpretation.

Part one builds on the work done in specialization up until now, it highlights the theoretical background necessary to formulate the two properties that are central to this work. The basic notation is defined in part one, it is based on the notation used in the books "Computability and Complexity from a Programming Perspective" by Neil Jones (Jones, 1997) and the book "Partial Evaluation" by Neil Jones, Carsten Gomard and Peter Sestoft (Jones *et al.*, 1993). The reader is expected to have read enough of both books to understand the general idea of partial evaluation and be familiar with the notation used in both books.

Over the years a lot of best practices have emerged in the field of specialization for both specializers and interpreters. Part one presents the best practices needed in part two when interpreters and specializers are developed for new languages.

Part two search for a language with the two properties identified in part one. The search leads to two languages $\mathcal{I}$ and $\mathcal{IF}$. For both of these languages a self interpreter and a specializer is build using the advice from the best practices identified in part one.

The result of the search is supported both by theoretical and experimental work.

Chapter 1

Introduction to Specialization

This chapter is a short presentation of the themes from partial evaluation that are needed

The goal of introducing partial evaluation is to justify that it is interesting to work with specialization of a self interpreter for a language. This is interesting as specializing a self interpreter is a step on road to a self applicable partial evaluator (Jones *et al.*, 1993, Section 7.4).

This chapter contains:

- A historical overview of Specialization
- An introduction to the theoretical background of specialization.
- A discussion of aspects of interpretation relevant when specializing an interpreter.
- A discussion of the practical aspects of specialization.

1.1 Brief historical overview of specialization

The progress of specialization and partial evaluation can roughly be divided into two eras: The pre 1985 era and the post 1985 era.

The pre era is characterized by theoretical results and guesses on the existence of self applicable specializers.

1985 is the year the world saw the first self applicable partial evaluator.

After 1985 self applicable partial evaluators was developed for many language including C, Prolog and Scheme.

1.1.1 Pre 1985 era

The pre 1985 era included the following major events:

- 1952, Kleene asserts the feasibility of partial evaluation (Kleene, 1952).
- 1964, the term "partial evaluation" appears in the literature (Lombardi & Raphael, 1964).
- 1971, Futamura considers the self applicable partial evaluator, and links it to compilation (Futamura, 1971).
- 1975, First published account of compiler generator generation by double self application (Beckman *et al.*, 1976).
- 1985, The first self applicable partial evaluator is published by Jones, Gomard and Søndergaard.(Jones *et al.*, 1985)

1.1.2 Post 1985 era

After 1985 substantial work was done for all types of languages including imperative languages, functional languages, Prolog and object oriented languages. This lead to the publication of self applicable partial evaluators for many popular languages including C, Prolog and Lisp.

1.2 Partial Evaluation

In order to discuss the quality of partial evaluators a definition of what is good and what is bad is needed. This section defines all bits necessary to define the property needed in part two, when discussing the optimality of the developed partial evaluators.

The existence of program specializers is the result of what is known as Kleene's s-m-n theorem. This theorem states that for a programming language L there exists a specializer written in L specializing L-programs to L-programs. Later Futamura used a specializer and an interpreter to generate compilers and compiler genrators. This result is known as the Futamura Projections.

This section contains a recap of the basic definitions from the field of Partial evaluation that will be needed to define one of the two properties of interest in part two.

This introduction is a condensation of the material covered by Jones in the book "Computability and Complexity from a programming perspective" (Jones, 1997, chapter 6)

1.2.1 Definitions

To understand the property of specializers used in part two, the following definitions are needed. The definitions makes use of these names for programs and data.

- `source`: A program written in language S
- `target`: A program written in language T
- `data`: Program input in language S-data `data` = `(static.data)`
- `static`: Program data in language S-data that is static during program execution
- `dynamic`: Program data in language S-data, that can change during program execution.
- `output`: data in S-data, the result of executing a program.

Definition 1.2.1. A timed programming language.
A timed programming language is needed in order to discuss the quality of the output of a specializer. A timed programming language L consists of

1. Two sets, L-programs and L-data

2. The semantic function for L is a function $[\![\bullet]\!]^{\mathtt{L}}$: L-programs $\rightarrow$ L-data $\rightarrow_{\perp}$ L-data
3. The time function for L is a function $time^{\mathtt{L}}$: L-programs $\rightarrow$ L-data $\rightarrow_{\perp}$ $\mathbb{N}$ such that for any `source` $\in$ L-programs and `d` $\in$ L-data, $[\![\mathtt{source}]\!]^{\mathtt{L}}(\mathtt{d}) = \perp$ iff $time^{\mathtt{L}}_{\mathtt{source}}(\mathtt{d}) = \perp$

Definition 1.2.2. Semantic equality.
Two programs `p` and `q` are semantically equivalent, written $\mathtt{p} \sim \mathtt{q}$

$$\mathtt{p} \sim \mathtt{q} \Leftrightarrow [\![\mathtt{p}]\!](\mathtt{data}) =_{\perp} [\![\mathtt{q}]\!](\mathtt{data})$$

Definition 1.2.3. Program execution.
Executing a program `source` written in S, with input `data` that generates the output `output` is written as

$$\mathtt{output} = [\![\mathtt{source}]\!]^{\mathtt{S}}(\mathtt{data})$$

Applying the semantic function for `S` to the program `source` and the input `data` the result is `output`.

Definition 1.2.4. Interpretation.
L-program `int` is an *interpreter* of S written in L if $[\![\mathtt{int}]\!]^{\mathtt{L}}$ is an interpreting function of S.

$$\mathtt{output} = [\![\mathtt{int}]\!]^{\mathtt{L}}(\mathtt{source.data})$$

Applying the semantic function of L to the program `int` and the input `(source.data)`, the result `output` is the same as executing `source` with the input `input`.

Definition 1.2.5. Specializer.
L-program `spec` is a *specializer* from S to T if $[\![\mathtt{spec}]\!]^{\mathtt{L}}$ is a specializing function.

$$[\![\mathtt{source}]\!]^{\mathtt{S}}(\mathtt{data}) = [\![[\![\mathtt{spec}]\!]^{\mathtt{L}}(\mathtt{source.static})]\!]^{\mathtt{T}}(\mathtt{dynamic})$$

$[\![\mathtt{spec}]\!]^{\mathtt{L}}$`(source.static)` is a T-program. When applying the semantic function for T to the T-program and the dynamic part of the input `dynamic`, the result is the same as when executing `source` directly on the complete input `data`.

Definition 1.2.6. Compiler.
L-program `compiler` is a *compiler* from S to T if $[\![\mathtt{compiler}]\!]^{\mathtt{L}}$ is a compiling function.

$$\mathtt{target} = [\![\mathtt{compiler}]\!]^{\mathtt{L}}(\mathtt{source})$$

The connection between `source` and `target` is

$$[\![\texttt{target}]\!]^{\texttt{T}}(\texttt{data}) = [\![\texttt{source}]\!]^{\texttt{S}}(\texttt{data})$$

The result of executing the target is the same as that of executing source directly

Definition 1.2.7. Compiler generator.
L-program `cogen` is a *compiler generator* if

$$\texttt{compiler} = [\![\texttt{cogen}]\!]^{\texttt{L}}(\texttt{int})$$

A compiler generator is a program that when applied to an interpreter produces a compiler.

1.2.2 Specialization as partial evaluation

Kleene's s-m-n theorem proves the existence of specializers. The next obvious question is: Does the specialized program have properties that differ from the original program. The proof given by Jones (Jones, 1997, Section 5.2) gives the idea behind the trivial specialization i.e. construct the full input by combining the static and dynamic parts of the input data, and then run the original program on the combined data.

The trivial specialization runs slower than the original program. Partial evaluation is the subset of specializations where the output program runs at least as fast as the input program. In partial evaluation a general program is specialized with respect to partially available data and a residual program is generated. The goal of the partial evaluator is to do as many of the computations as possible, based in the partial input available.

This notion of doing as much as possible makes it possible to talk about specialization quality. This is discussed futher in the section on Jones Optimality in Section 1.2.4.

1.2.3 Futamura projections

The Futamura projections links interpretation, compilation and compiler generation with each other using two programs, an interpreter and a specializer.

`int` is an L-program that interprets programs written in S.

`spec` is an Imp-program that specializes programs written in S to programs written in T.

`source` is a program written in S

`target` is a program written in T

`input` = (`static.dynamic`) is the input, where `static` is the static part of the input, and `dynamic` is the dynamic.

First Futamura projection

The first Futamura projection describes compilation of a source program `source` written in language S to an equivalent program `target` written in T, using the programs `spec` and `int`.

Definition 1.2.8. Futamuta Projection 1

$$\texttt{target} = [\![\texttt{spec}]\!]^{\texttt{Imp}}(\texttt{int.source})$$

Verification is done starting with the equation for execution, and rewriting it using the definitions of Section 1.2.1.

$$\begin{aligned}
\texttt{output} &= [\![\texttt{source}]\!]^{\texttt{S}}(\texttt{static.dynamic}) && \text{Definition 1.2.3}\\
&= [\![\texttt{int}]\!]^{\texttt{L}}(\texttt{source.(static.dynamic)}) && \text{Definition 1.2.4}\\
&= [\![[\![\texttt{spec}]\!]^{\texttt{Imp}}(\texttt{int.source})]\!]^{\texttt{T}}(\texttt{static.dynamic}) && \text{Definition 1.2.5}\\
&= [\![\texttt{target}]\!]^{\texttt{T}}(\texttt{static.dynamic}) && \text{Definition 1.2.8}
\end{aligned}$$

Second Futamura projection

The second Futamura projection describes the generation of a S to T compiler. This is possible if the specializer is written in L, i.e. L = Imp.

Definition 1.2.9. Futamura Projection 2

$$\texttt{compiler} = [\![\texttt{spec}]\!]^{\texttt{L}}(\texttt{spec.int})$$

Verification

$$\begin{aligned}
\texttt{target} &= [\![\texttt{spec}]\!]^{\texttt{L}}(\texttt{int.source}) && \text{Definition 1.2.8}\\
&= [\![[\![\texttt{spec}]\!]^{\texttt{L}}(\texttt{spec.int})]\!]^{\texttt{T}}(\texttt{source}) && \text{Definition 1.2.5}\\
&= [\![\texttt{compiler}]\!]^{\texttt{T}}(\texttt{source}) && \text{Definition 1.2.9}
\end{aligned}$$

Third Futamura projection

The third Futamura projection describes the generation of compiler generators.

Definition 1.2.10. Futamura Projection 3

$$\texttt{cogen} = [\![\texttt{spec}]\!]^{\texttt{L}}(\texttt{spec.spec})$$

Verification

$$\begin{aligned}
\texttt{compiler} &= [\![\texttt{spec}]\!]^{\texttt{L}}(\texttt{spec.int}) && \text{Definition 1.2.9}\\
&= [\![[\![\texttt{spec}]\!]^{\texttt{L}}(\texttt{spec.spec})]\!]^{\texttt{T}}(\texttt{int}) && \text{Definition 1.2.5}\\
&= [\![\texttt{cogen}]\!]^{\texttt{T}}(\texttt{int}) && \text{Definition 1.2.10}
\end{aligned}$$

1.2.4 Desirable properties of specialization

In order to assert the quality of a partial evaluator, some properties are needed in order to quantify the quality. Other properties are used to specify when a partial evaluator works correctly. The most important property used in part two is Jones Optimality.

Totality

If possible $[\![\texttt{spec}]\!]$ should be a total function. This ensures that `spec` will terminate on when applied to any program `p` and input `s`.

Completeness

Computational completeness: All computations on the static input must be performed by the specializer.

A specializer which aims at computational completeness will have problems specializing a program `p` whose output depend solely on the static input, when $[\![\texttt{p}]\!]$ is a partial function. That is `p` does *not* terminate for some input `static`

Since the specializer must do all computations for `p` in order to be computational complete, it *will* fail to terminate when $[\![\texttt{p}]\!]\texttt{(static.dynamic)} = \perp$.

So either we restrict our selves to specializing total functions, or we do not aim at computational completeness. Computational completeness is not considered for any of the specializers presented in Chapter 2

Practical mix of Totality and Completeness

The competing notions of totality and completeness leads to this practically attainable requirement (Jones *et al.*, 1993, p. 104).

$$[\![\texttt{p}]\!](\texttt{s.d}) \neq \perp \implies [\![\texttt{spec(p.s)}]\!] \neq \perp$$

If executing `p` on `(s.d)` does terminate, then the specializer must terminate when given the input `(p.s)`. This property is part of the overall goal of the partial evaluators of part two.

Jones-optimal

Jones-optimality concerns a subset of specializations, precisely those that specialize an interpreter with respect to a program.

The existence of programs of this kind is easily proved:

Theorem 1.2.11. Existence of interpreter specializers.

Let `sint` be a self interpreter for some language, that takes as input `(prog.input)`, and let `p` be a program written in this language.

Then there exists a specializer `spec` with the following property

$$[\![[\![\texttt{spec}]\!](\texttt{sint.p})]\!](\texttt{input}) = [\![\texttt{sint}]\!](\texttt{p.input})$$

Proof of Theorem 1.2.11. Using pseudo code:

```
spec:
  output(
    sinput = (prog.input);
    sint( sinput )
  )
```

□

This is clearly not a very good specialization, as it just packs the self interpreter `sint` into the residual program. A better specializer would try to do as much of the computations of `sint` as possible, based on the input program `p`. If the specializer is able to remove all traces of the interpreter then the pair `(spec.sint)` is said to be Jones Optimal.

Definition 1.2.12. Jones-optimal.

A program specializer `spec` is Jones-optimal for a self-interpreter `sint` in case for every program `p` and data `d`, if $\texttt{sint}_{\texttt{p}} = [\![\texttt{spec}]\!](\texttt{sint.p})$ then

$$\mathit{time}_{\texttt{sint}_{\texttt{p}}}(\texttt{d}) \leq \mathit{time}_{\texttt{p}}(\texttt{d})$$

Self applicable partial evaluator

The last desirable property of a partial evaluator is the self applicability. If the partial evaluator is self applicable it will fill the role of the specializer in the Futamura projections from Section 1.2.3. A recipe for deriving a self applicable partial evaluator is given in Chapter 7 in the partial evaluation book (Jones *et al.*, 1993, pp. 157-159).

Before the actual recipe is given, a stepping stone is presented. Since a self applicable specializer must be able to interpret the static parts of the input program, it must contain a self interpreter. Thus specializing a self interpreter with good results is a simpler problem that will pave the way for the self applicable specializer.

1.3 Interpretation

The goal of part two is to analyse the effect of specializing interpreters with a certain property. This section introduces the property in question, and a series of considerations to remember when developing the interpreters in part two.

Interpretation is the act of executing a program. The interpreter is a program written in some language, called the implementation language. The implementation language can be the same language as the one interpreted.

Interpreters arise in two contexts:

- The interpreter is used to define a new language.
- The interpreter is the implementation of a language description, typically some sort of syntactic and semantic description of the language.

When comparing the time of directly executing a program with the time of interpreting the program, intuition says that interpretation must take extra time. This extra time comes from decoding the representation of the program used by the interpreter, and maintaining the program state. This is called the interpretation overhead.

1.3.1 Interpretation overhead

Interpretation overhead is the cost associated with interpreting a language S in a language L (S = L is possible but not required). The running time of the program `p` on either an S or L machine is linked in the following way:

$$\alpha_{\mathtt{p}} \cdot \mathit{time}^{\mathtt{S}}_{\mathtt{p}}(\mathtt{d}) \leq \mathit{time}^{\mathtt{L}}_{\mathtt{int}}(\mathtt{p.d})$$

That is, the interpretation of program `p` is at least a factor $\alpha_{\mathtt{p}}$ slower than executing it directly.

As indicated the factor depends on `p` in some way. The factor is build up of two parts. One that depends on `p` and one that does not.

$$\alpha_{\mathtt{p}} = c + f(\mathtt{p})$$

Where c covers the part of the interpreter that dispatch on syntax, and $f(\mathtt{p})$ covers the parts where the interpreter handles state information, like updating variables or looking up function bodies. $f(\mathtt{p})$ covers all the time taken by the interpreter to handle program dependent work.

1.3.2 Program independent interpretation overhead

Definition 1.3.1. Program independent interpretation overhead (PIIO).

An interpreter `int` is said to have program independent interpretation overhead if the following is true for all programs `p` and data `d`:

$$time^{\mathtt{L}}_{\mathtt{int}}(\mathtt{p.d}) \leq C \cdot time^{\mathtt{L}}_{\mathtt{p}}(\mathtt{d})$$

Where C is a factor that does *not* depend on `p`.

Program independent interpretation overhead binds the execution time of the interpreter from above by a constant factor that is independent of the program being interpreted.

In the article "Characterizing Computation Models with a Constant Factor Time Hierarchy" (Rose, 1996) Eva Rose list the requirements a language must meet in order for it to have the PIIO property.

These properties are:

1. Finite set of constructors. If a language supports user defined constructors, the interpreter needs to keep track of all the user defined constructors. Looking up a specific constructor will take time that depend on the number of constructors defined by a program.

2. Finite branching of data. If one or more constructors can take an unlimited number of arguments, the arguments must be stored in some kind of container, and lookup of a specific argument in this container will take time dependent on the program.

3. Finite number of variables. The interpreter must have a representation of the store. If the number of variables is unlimited, the lookup of a specific variable will take time that depends on the number of variables used in the interpreted program.

4. Finite number of functions. The interpreter must maintain some mapping between the function names and the function bodies. If the number of functions in a program is unlimited, the time to locate a function body will take time that depends on the number of functions in the interpreted code.

5. Finite branching of code. If the language supports branching with an unlimited number of branches in a single construct. The interpreter must lookup the correct branch in some way, and this lookup takes time that depends on the number of branches used in the interpreted program.

On face value it seems reasonable that these are certainly requirements to be met for a language if it is to have the PIIO property. If a language fails to have one of these five properties it seems plausible that the language might have program dependent overhead.

If one of the properties is missing, it is possible that the problem with the PIIO property can be handled by other means.

In Chapter 2 the five properties presented here is used to argue PIIO, and it is backed by experimental data.

1.3.3 Implementing interpreters

Like every where else in the software development there are certain practices that crystallize through years of work and experience. This is also the case for specialization. Neil Jones have collected some of these in the paper "What not to do when writing an interpreter for specialization" (Jones, 1996). The following is a condensation of the best practices collected over the years.

If the interpreter is meant for specialization there are a number of issues that the implementer must be aware of, as they will have impact on the feasibility and efficiency of the specialization.

Definitional interpreters

A definitional interpreter is used to define a given language. There are two flavors of definitional interpreters.

- Written in the language defined, that is a self interpreter.
- Written in a language different from the language defined. The language definition can then inherit features from the language the interpreter is written in.

Derived interpreters

Interpreters can be derived from definitions by other means, i.e. a language defined by some formal semantics. The choice of formal description will also have impact on the specializer.

Small step operational semantics: Small step operational semantics often gives rise to non-deterministic evaluation, which in turn gives rise to non-directionality (like the evaluation order of function arguments in C, it is undetermined). These two properties is generally a problem when specializing as the specializer must handle the two situations in some graceful way.

Big step operational semantics: Implementing a big step semantics is usually quite straight forward. The problems of the small step semantics are not present, which makes a big step semantics a good choice for describing languages later used for specialization.

Denotational semantics: A denotational semantics for a language will often be straight forward to implement, but it will most likely require both recursion and higher order domain definitions. This makes the interpreter quite cumbersome to work with when specializing it (Jones, 1996).

Interpreter properties

To get good specialization result when specializing interpreters it is important that the interpreter has one of two traits, either it is compositional or it is semi-compositional.

A compositional interpreter is an interpreter where all computations on the syntax of the interpreted program works on a substructure of the original program. That is if the interpreter calls a function to evaluate or interpret a subexpression, it must always work on a subset of the expression available in the calling context.

If an interpreter is compositional, the variables containing syntax will always be of bounded static variation (See Section 1.5.1 for definition).

An interpreter is said to be semi compositional if the syntactic parameters are substructures of the original program, but the requirements for the compositional interpreter that the “size” of the substructure must always decrease is relaxed.

In semi compositional interpreter’s the possible values of syntax variables will also be of bounded static variation.

1.4 Scheme0

Scheme0 is used as the illustration language for introducing the concepts and techniques of specialization. First Scheme0 syntax is presented, then the semantics of Scheme0 is presented informally and finally some example code that will be used when discussing the practical aspects of specialization in section 1.5

1.4.1 Syntax

Scheme0 is a first order functional language. It supports simple arithmetic expressions using known mathematical operators, and operations on binary trees. The syntax is given in Figure 1.1.

```
Program    ::=  Equation, ..., Equation
Equation   ::=  FuncName VarList Expr
VarList    ::=  ( Var, ..., Var )
Expr       ::=  Constant
             |  Var
             |  if Expr Expr Expr
             |  call FuncName Arglist
             |  op Arglist
Arglist    ::=  ( Expr, ..., Expr )
Constant   ::=  Numeral
             |  quote Value
           ::=  car | cdr | cons | = | + | ...
```

Figure 1.1: Scheme0 syntax

An implementation of the list append function in Scheme0 is given Figure 1.2. The lists are coded straight forwardly in a binary tree. For a complete description of the list encoding see Section 2.1.4.

1.4.2 Semantics

Scheme0 uses a call by value semantics in function calls. All arguments to functions are completely evaluated and the values are bound to the formal parameters of the called function.

A complete program consist of a number of function definitions, with the first one being the goal function. An example of a complete program is given in Figure 1.3

```
reverse( l, r ) =
  if( l, call reverse( tl l, cons( hd l, r ) ), r )

append_private( l1, l2 ) =
  if( l1, call append_private( tl l1, cons( hd l1, l2 ) ), l2 )

append( l1, l2 ) =
  append_reverse( reverse( l1, n ), l2 )
```

Figure 1.2: Scheme0 list append

In Scheme0 the variables are all bound to function parameters, no free nor global variables exist.

```
main() = fak( 4 )
fak( n ) = *( n, call fak ( -(n, 1) ) )
```

Figure 1.3: Complete Scheme0 program

1.4.3 Example

As a running example we use the Scheme0 implementation of Ackermann's function given in Figure 1.4.

```
ack( m, n ) =
  if( =( m, 0 ) )
    ( +( n, 1 ) )
    ( if( =( n, 0 ) )
        ( call ack ( -( m, 1 ), 1 ) )
        ( call ack ( -( m, 1 ), call ack ( m, -( n, 1 ) ) ) ) )
```

Figure 1.4: Ackermann's function in Scheme0

Ackermann's function is adequate for illustration as it uses all features of the language

1.5 Practical aspects of Partial Evaluation

In part two, two partial evaluators are developed. The partial evaluators are developed based on the collection of best practices presented here. There are certainly many ways to build a specializer, the approach here is the one advocated in the book by "Partial evaluation" by Jones, Gomard and Sestoft (Jones *et al.*, 1993).

The specialization process is conceptually split into a number of steps. Each step takes some form of input, and generate some output. Some of the steps might be bypassed.

- Binding time improvement: The act of transforming the program into another semantically equivalent, one for which the next task, the binding time analysis produces better results, better meaning more variables are declared as static, making more information available for the specialization step.
- Binding time analysis: Find the variables that depend statically on the statically provided input.
- Program annotation: Generate annotations for the program that will guide the specialization task.
- Specialization: The task of computing the static parts by abstract interpretation of the input program, and the task of generating code for the dynamic parts.
- Transition compression: Remove unnecessary transition in the program produced by the specializer.

The tasks that are executed before the specialization is categorized as the pre specialization phase, the tasks that are executed after is called the post specialization phase.

The partial evaluation literature divides partial evaluators into two categories, *online* and *offline*. The division is made based on how the specializer decides if a statement should be statically computed or residualized.

If the specializer makes the decision based on information that is given as input (typically as an annotated program or a division) the specializer is of the offline kind.

If the specializer makes the decision based on the actual staticness of the variables at the time the statement is encountered it is of the online kind.

The offline specializer depends on the analysis done in the pre specialization phase. If this analysis is not able to mark a statement as static, it will be residualized by the specializer. The specializer does not try to be clever. The online specializer will always make the decision based on the actual state of the abstract interpretation taking place during specialization. This makes it harder to detect loops to be residualized, as the specializer cannot have a global view of the specialized program. The online specializer is typically able to exploit more staticness of the input program, at the cost of more house keeping to avoid going into an endless specialization loop.

Online specialization A specializer is categorized as online if the pre specialization phase is integrated into the specialization task.

Offline specialization A specializer is categorized as offline if the pre specialization phase takes place at a different time than specialization.

In order to have a better flow of understanding the steps are presented "out of order".

1.5.1 Binding time analysis

Goal: Produce a description of the variables used by a program, tagging them as either static or dynamic.

Input: The program to analyze

output: a list of pairs of variables and tags.

How: Binding time analysis is the task of calculating which parts of a program that can be computed ahead of time given part of the input to the program. This information is used later when the specializer decides whether to calculate the value of an expression or construct a statement in the residual program representing the calculation.

In the example in Figure 1.4 the parameter m to Ackermann's function could be given the value of 2, and the job of the binding time analysis (BTA) is to determine which of the computations in Ackermann's functions that can be computed at specialization time.

To do this we need to define the points in the program that we want to specialize. In Scheme0 the obvious point is to specialize functions.

Doing BTA gives as a result a division of a functions input parameters into two categories, static and dynamic. This division tells if a given parameter is guarantied to be static during all calls, or if it will be dynamic. A dynamic parameter might be given static content, depending on the type of division produced.

The division is usually split into two types, a monovariant one and a polyvariant one. The monovariant division for a functions parameters, must mark an arguments as dynamic, if it at one point in the execution of the program will be bound to a dynamic value, if not it is marked as static. A polyvariant division keeps track of the division from the point of view of the call site, that is it is possible for a function to have many divisions, hence the name polyvariant division.

Monovariant division

In a monovariant devision a functions parameters is divided according to the least restrictive division, that is if a given parameter is static in some contexts and dynamic in other, it will be classified as dynamic. Constructing a division for the program in Figure 1.5 assuming that “s” is a static parameter and “d” is dynamic, then the parameters to the call of sum in the then branch would be (S, S, D), and in the else branch (S, D, D). As the monovariant division can only have one division pr. function, the sum function should be categorized as the least restrictive of the two (S, D, D).

```
run( s, d ) =
  if s = d
  then sum ( s, s, d )
  else sum ( s, d, d )

sum( a, b, c ) = a + b + c
```

Figure 1.5: Division example program

A monovariant division can usually be constructed starting at a division where all functions are totally static except for the first functions division which is given as part of the arguments to the specializer (Jones *et al.*, 1993, Section 5.2), and continue to propagate this information down the call from the function call, recording the dynamic nature of the parameters. This process is repeated until no more changes occur. This resembles the fix point iterations familiar from math i.e. Newton-Raphson approximation. The monovariant division is the smallest division possible, when smallest is a measure of how many dynamic parameters a function take.

Polyvariant division

In a polyvariant division the division of a functions parameters are made according to the staticness of the arguments in the calling context.

To extend the example from the last Section. A polyvariant division of the program in Figure 1.5 would lead to both (S, S, D) and (S, D, D) to be valid divisions.

The normal trick to handle this, is to make a number of copies of sum, one for each possible way of dividing the parameters, and use this “new” program in the rest of the specialization. The program in Figure 1.5 would be transformed into the program in Figure 1.6. After the transformation a monovariant division is constructed.

```
run( s, d ) =
  if s = d
  then sum_ssd ( s, s, d )
  else sum_sdd ( s, d, d )

sum_ssd( a, b, c ) = a + b + c

sum_sdd( a, b, c ) = a + b + c
```

Figure 1.6: Polyvariant division copies functions

This approach to polyvariant division is the cause of some code explosion. This explosion is not considered a problem as the number of possible copies of a given function is bounded be the number of arguments it take.

Example

The division for Ackermann’s function from Figure 1.4 when m is static is just (s, d), meaning that m is static and n is dynamic. This happens because the first parameter to Ackermann’s function in all internal calls depend only on m.

Bounded static variation

Bounded static variation happens when a variable appears dynamic at first glance, but when analyzed further it only takes on a finite number of values. When this happens, the specializer can use this information to generate code that switches among the possible values.

When a program has variables of bounded static variation these variables are marked static by the binding time analysis. The specializer must then generate code depending on the content of the variable, usually by using some sort of case statement.

1.5.2 Binding time improvement

Goal: Transform program into a new program which when run through the binding time analysis produce more variables tagged as static.

Input: Program to transform

Output: Transformed program

How: Binding time improvement is the process of changing the input program to a semantically equivalent one. The effect of the change is to improve the binding time analysis, marking more variables as static.

The improvement process can be manual or automatic. If it is not automatic, it necessarily takes place in the pre specialization phase.

Binding time improvement and binding time analysis often work together when done manually. A series of iterations of the following steps is done until the binding time analysis is “good” enough.

- do binding time analysis
- discover binding time improvements
- transform program based on binding time improvement

Binding time improvements come in many forms, some are:

- fold/unfold: Loops in the code are unfolded when possible, might delay the computations depending on the dynamic input.
- Variable splitting(one variable becomes two or more): A variable is used to store more than one thing, where one part might be static or of bounded static variation and the other dynamic.
- inlining: copy code inside branches if impossible to tell state after if statement.

1.5.3 Program annotation

Goal: Produce a program that is tagged with information to guide the specialization process.

Input: Program to specialize, and result of binding time analysis.

Output: Annotated program

How: The program is annotated with information that will guide the specialization process. The annotation traditionally takes the form of two-level code. Two level code has the same basic structure as the original program except that all constructs are annotated with a tag telling if it is static or dynamic.

The annotations are typically generated by abstract interpretation of the code, marking all the computations that only depend on static parts as static and the rest as dynamic.

The result of the binding time analysis is used to help decide the staticness of the program statement.

Scheme0 two level code

For Scheme0 the syntax of the two level code is given in Figure 1.7. For Scheme0 the two level approach is materialized as two new constructs for each old one i.e. if statements have two counterparts a static `ifs` for which the deciding expression must be computable at specialization time, and a dynamic `ifd` for which the decision must be postponed to the residual program.

Expr	::=	Constant
	|	Var
	|	`ifs` Expr Expr Expr
	|	`ifd` Expr Expr Expr
	|	`calls` FuncName (Expr, . . . , Expr)
	|	`calld` FuncName (Expr, . . . , Expr)
	|	`ops` (Expr, . . . , Expr)
	|	`opd` (Expr, . . . , Expr)
	|	`lift` Expr
SDArgs	::=	(Arglist) (Arglist)

Figure 1.7: Annotated expressions of Scheme0

The new keyword `lift` takes care of "lifting" a static expression that should result in a value to a dynamic residual expression that can be used in a dynamic context.

Using the obvious division for Ackermann's function from Figure 1.4 the two level program is given in Figure 1.8.

```
ack( m, n ) =
  ifs( =( m, 0 ) )
     ( +( n, 1 ) )
     ( ifd( =( n, 0 ) )
          ( calls ack ( -( m, 1 ), 1 ) )
          ( calld ack ( -( m, 1 ),
                        calld ack ( m, -( n, 1 ) )
                      )
          )
     )
```

Figure 1.8: 2 level code for Ackermann's function in Scheme0

1.5.4 Specialization

Goal: Partially evaluate program using partially available data

Input: Annotated program, static data

Output: Residual program

How: Specialization do two things.

1. Generates code for everything marked dynamic
2. compute everything that is marked static.

For a static if `ifs` the specializer decides which branch should be taken, and the specializer continues by specializing the code in the branch chosen. For a dynamic if `ifd`, an `if` is generated for the residual program, and each branch is specialized separately.

The specializer usually works by maintaining an abstract view of the store for running the program. This store is used to decide static decisions, like the condition in an if statement.

For Scheme0 a function is specialized once for each possible value of a static parameter. For the annotated program in Figure 1.8 the residual code produced is listed in Figure 1.9.

```
ack_2( n ) =
  if( =( n, 0 )
    call ack_1 ( 1 )
    call ack_1 ( call ack_2 -( n, 1 ) )

ack_1( n ) =
  if( =( n, o )
    call ack_0 ( 1 )
    call ack_0 ( call ack_1 -( n, 1 ) )

ack_0( n ) =
  +( n, 1 )
```

Figure 1.9: Ackermann specialized to m = 2

1.5.5 Transition compression

This step attempts to use program transformations to make the program run faster.

Goal: Remove all unnecessary state transitions.

Input: A program

Output: A faster program

How: This depends largely on the language chosen to work in. For Scheme0 the obvious transition compression occurs when function `a` calls function `b` that immediately calls function `c`. Here the call to `b` in `a` can be eliminated by inserting a call to `c` in `b`'s place.

For languages that does total updates of stores, two sequential updates can be combined to a single update.

This step is *not* just an optimizing phase. Optimization is certainly a large part of it, but the aim is a global optimization, which can require the use of non-optimizing transformations. These non-optimizing transformations usually enable more or more powerful optimizations.

Program transformations come in many forms, some of them makes the program execute faster and some of them makes the program less complex i.e. a program with nested loops is more complex than a program that does not have nested loops. Program transformations that then “flatten” the nested loops by sequentializing them transform a complex program to a less complex one. The less complex program can often be optimized with a better result than the complex one.

One point of interesting insight is that if a transformation cannot be characterized as an optimization it may very well be the cause of problem for Jones Optimality. A transformation of this kind is found in Section 2.6.12

1.5.6 Specializing self interpreters

Part two focuses on the specialization of self interpreters. The specialization of self interpreters poses some general considerations regardless of the language used.

The goal of specializing the self interpreter `sint` with program `p` is a residual program that resemples `p` as much as possible. The goal can be divided into two orthogonal parts:

- Obtain a control flow structure that resembles the structure of `p`
- Obtain a data flow that resembles that of `p`

With these two orthogonal parts the specialization process can be viewed as happening in two stages.

1. Specialize `sint` with `p` to get program `p'` that have control flow similar to `p`, and dataflow similar to `sint`. This is usually done by the specializer task in Section 1.5.4
2. Optimize `p'` to program `p''` that have both control and data flow of `p`. This is typically the job of the transition compression task in section 1.5.5

1.6 Summary

Chapter one gives a short introduction to the theoretical background of specialization. This background includes an introduction to some of the historical highlights of the field of specialization. This introduction list some desirable properties of a specializer, including the JOI property (See Definition 1.2.12).

Interpretation is looked at from the perspective of producing interpreters that are suitable for specialization. The requirements for a language to have the PIIO property (See definition 1.3.1) are discussed.

The practical aspects of specialization is then discussed using Scheme0 as an example language.

Chapter 2

Achieving PIIO and JOI

The main goal of the project is to provide a language which have both the JOI and the PIIO property. The PIIO property is defined in Definition 1.3.1 and the JOI property is defined in Definition 1.2.12.

The search for the language starts by analysing $\mathcal{I}$ which has the PIIO property. A self interpreter and a specializer for $\mathcal{I}$ is implemented (See Appendix A). A suite of sample programs (See Appendix A.2) with input and expected output is used to assert the quality of the implemented specializer. The results of executing the sample programs is used to identify the features of $\mathcal{I}$ that hinder the JOI property.

It could be argued that it would be easier to start with a language that has the JOI property, and then remove the parts that hinder the PIIO property. The reason for not doing this is largely that I stated to explore the PIIO property and it seemed easier to continue down that track once started.

The identification of problematic features of $\mathcal{I}$ leads to the definition of another language $\mathcal{IF}$. The analysis of $\mathcal{IF}$ uses a specializer and a self interpreter (See Appendix B). The analysis is made on the basis of timed executions of various programs (See Appendix B.2) written in $\mathcal{IF}$.

The goal of a language with both the PIIO and the JOI property is not satisfied with $\mathcal{IF}$. The timed tests (See Section 2.9) shows that for certain small programs the specialization of the self interpreter does not produce optimal results.

2.1 $\mathcal{I}$

The language $\mathcal{I}$ as presented by Jones in "Computability and Complexity" (Jones, 1997, Chapter 3.7) has program independent interpretation overhead. The programming language has a simple structure and a self interpreter is easily realized. $\mathcal{I}$ is used as the first language to try to prove both JOI and PIIO property.

After the presentation of the interpreters and the specializer a series of test and experiments are conducted and the results are shown and analysed.

2.1.1 Syntax

The syntax for $\mathcal{I}$ is presented in Figure 2.1 in EBNF.

```
Command     ::=  X := Expression
              |  Command ; Command
              |  while Expression do Command
Expression  ::=  X
              |  val Value
              |  cons Expression Expression
              |  hd Expression
              |  tl Expression
              |  =? Expression Expression
Value       ::=  nil
              |  ( Value, Value )
```

Figure 2.1: $\mathcal{I}$ syntax

The values of $\mathcal{I}$ are binary trees with `nil` as the only element in the set of possible leafs. The set of possible values are called $\mathbb{D}$ by Jones (Jones, 1997, Definition 2.1.1).

2.1.2 Semantics

The store used for execution of programs in $\mathcal{I}$ contains only one variable `X`, which contains a single value in $\mathbb{D}$.

The semantics for $\mathcal{I}$ is given by a big-step operational semantics with two judgments, one for commands and one for expressions. The two judgments are

- $< c, X > \rightarrow X$, see Figure 2.2

$$\frac{< \mathtt{e}, X > \rightarrow v}{< \mathtt{X := e}, X > \rightarrow X[v]} \qquad \frac{< c_0, X > \rightarrow X'' \qquad < c_1, X'' > \rightarrow X'}{< c_0; c_1, X > \rightarrow X'}$$

$$\frac{< e, X > \rightarrow \mathtt{(v1.v2)} \qquad < c, X > \rightarrow X'' \qquad < \mathtt{while}\ e\ \mathtt{do}\ c, X'' > \rightarrow X'}{< \mathtt{while}\ e\ \mathtt{do}\ c, X > \rightarrow X'}$$

$$\frac{< e, X > \rightarrow N}{< \mathtt{while}\ e\ \mathtt{do}\ c, X > \rightarrow X}$$

Figure 2.2: Command judgment: $< c, X > \rightarrow X'$

- $< e, X > \rightarrow v$, see Figure 2.3

Where c is a command in $\mathcal{I}$ and e is an expression in $\mathcal{I}$.

2.1.3 Interpretation in ML

In order to enable the comparison of execution times of different $\mathcal{I}$ programs, a timed interpreter for $\mathcal{I}$ is implemented in ML. This interpreter is based on the big step semantics of Section 2.1.2.

Each Non-Terminal in the gramma for $\mathcal{I}$ in Figure 2.1 is implemented as a structure in ML (See Appendix A.1.3 and A.1.4), with a datatype and various functions that take the datatype as arguments.

The timed interpreter for $\mathcal{I}$ is made up of functions defined in these data structures.

- `Value.applyHd` See Appendix A.1.1
- `Value.applyTl` See Appendix A.1.1
- `Expression.evaluate` See Appendix A.1.3
- `Command.timed_interpret` See Appendix A.1.4

The $\mathcal{I}$ programs are written usign the ML datatypes of the three structures `Value`, `Expression` and `Command`.

The interpreter count the steps necessary to execute the $\mathcal{I}$ program according to this timing function:

$$\frac{}{< v, X > \to v} \qquad \frac{}{< X, X > \to v} X := v$$

$$\frac{< e, X > \to N}{< \texttt{hd}\ e, X > \to N} \qquad \frac{< e, X > \to (v1, v2)}{< \texttt{hd}\ e, X > \to v1}$$

$$\frac{< e, X > \to N}{< \texttt{tl}\ e, X > \to N} \qquad \frac{< e, X > \to (v1, v2)}{< \texttt{tl}\ e, X > \to v2}$$

$$\frac{< e1, X > \to v1 \qquad < e2, X > \to v2}{< \texttt{cons}\ e1\ e2, X > \to (v1, v2)}$$

$$\frac{< e1, X > \to v1 \qquad < e2, X > \to v2}{< \texttt{=?}\ \ e1\ e2, X > \to N} v1 \neq v2$$

$$\frac{< e1, X > \to v1 \qquad < e2, X > \to v2}{< \texttt{=?}\ \ e1\ e2, X > \to (N, N)} v1 = v2$$

Figure 2.3: Expression judgment: $< e, X > \to v$

$$\begin{aligned}
time_c(\texttt{X := e}) &= time_e(\texttt{e}) + 1 \\
time_c(\texttt{c}_1; \texttt{c}_2) &= time_c(\texttt{c}_1) + time_c(\texttt{c}_2) + 1 \\
time_c(\texttt{while e do c}) &= time_e(\texttt{e}) + time_c(\texttt{c}) + 1
\end{aligned}$$

$$\begin{aligned}
time_e(\texttt{X}) &= 1 \\
time_e(\texttt{v}) &= 1 \\
time_e(\texttt{hd e}) &= time_e(\texttt{e}) + 1 \\
time_e(\texttt{tl e}) &= time_e(\texttt{e}) + 1 \\
time_e(\texttt{cons(e}_1\texttt{, e}_2\texttt{)}) &= time_e(\texttt{e}_1) + time_e(\texttt{e}_2) + 1 \\
time_e(\texttt{=?(e}_1\texttt{, e}_2\texttt{)}) &= time_e(\texttt{e}_1) + time_e(\texttt{e}_2) + 1
\end{aligned}$$

At first glance the timing of the comparison operation `=?` is quite a thing to postulate. Jones discusses the justification of it in "Computability and Complexity" (Jones, 1997, Chapter 17).

2.1.4 Programming in $\mathcal{I}$

$\mathcal{I}$ lacks some features that exists in many high level languages. The basic ones are:

- if-then-else constructs
- empty commands or skip

I is a subset of language WHILE, also from the book "computability and Complexity" by Neil Jones. The language is simple because of the need to prove various properties in the book, while expressive enough to program useful algorithms in it. The reason for leaving out the if-then-else and skip command are to keep these proofs managable.

Both of the constructs are useful when programming a self interpreter, and equivalent constructions in $\mathcal{I}$ can be programmed.

Skip

The skip instruction can be coded as this:

```
X := X
```

If-then-else

The if-statement is encoded as suggested for the WHILE language by Jones (Jones, 1997, Section 2.1.4).

The code:

```
if( b ) then e1 else e2;
c
```

is transformed into

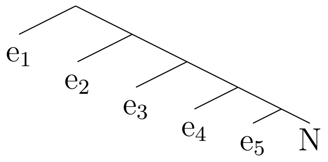
```
Z := b
W := true
while( Z ) do
  e1
  Z := false;
  W := false;
while( W ) do
  e2
  W := false;
c
```

Since $\mathcal{I}$ only has one variable, a stack of IF-variables must be allocated in the tree for the variable. An example of this where the if-stack is located in the left child of the X, and the "real" variable to do computations on are located in the right.

```
/* push new "if pair" onto if-stack,
 * branch condition b is evaluated */

X := cons( cons( cons( b,(N,N) ), hd( X ) ), tl( X ) );

/* run then part is b is true, reset
 * if pair to (N,N) when done.
while( hd( hd( hd( X ) ) ) )
  e1;
  X := cons( cons( (N,N), tl hd X ), tl X )

/* run else part if b is not true */
while( tl( hd( hd( X ) ) ) )
  e2
  X := cons( cons( (N,N), tl hd X ), tl X )

/* pop used if pair from if-stack */
X := cons( tl hd X, tl X );

c
```

Lists

The list `[ e1, e2, e3, e4, e5, e6 ]` is encoded in the following tree



List append example

The list append function written in I:

The program takes the two lists `x` and `y` as the pair `(x.y)` in `X`. It returns the appended list in `X`

```
X := cons( N, X );
```

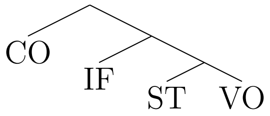


Figure 2.4: Basic data structure for self interpreter

```
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

The first while reverses the first list, and the second while adds the elements of the reversed list to the second list, one by one.

2.1.5 Self interpretation

Self interpretation of $\mathcal{I}$ is modeled on the self interpreter for WHILE given by Jones (Jones, 1997, Section 4.1.1). The variables used for this self interpreter is packed into a single variable using the simple technique also presented by Jones (Jones, 1997, p. 60).

The self interpreter makes use of the variables: `CO`, `IF`, `ST` and `VO`, they are packed is as in Figure 2.4. The four variables models these concepts:

`CO` code stack

`IF` if stack

`ST` computation stack

`VO` Variable of interpreted program.

The encoding of $\mathcal{I}$ programs as $\mathcal{I}$ data makes use of some constants that identify various parts of the program to be interpreted. These constants are defined `(prefix.index)` as in Figure 2.5.

An $\mathcal{I}$ program is a single $\mathcal{I}$ command, and it is encoded as the following tree:

	prefix		
`index`	N	(N,N)	(N,(N,N))
N	var	assign	doassign
(N,N)	value	sequence	dowhile
((N,N),N)	cons	while	dohd
(((N,N),N),N)	hd		docons
((((N,N),N),N),N)	tl		dotl
(((((N,N),N),N),N),N)	eq		doeq

Figure 2.5: $\mathcal{I}$ interpreter constants

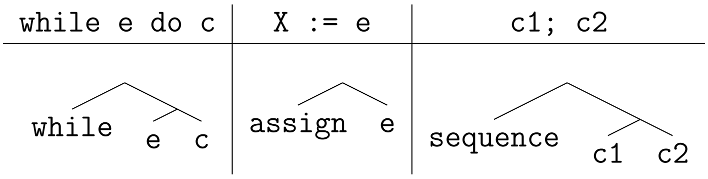
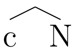
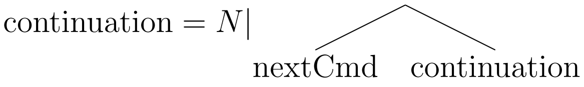


Figure 2.6: $\mathcal{I}$ command encoding



During interpretation the tree will change structure and at any point in the interpretation the tree will look like this:



The encoding of the $\mathcal{I}$ commands are given in Figure 2.6 and the expressions are encoded as shown in Figure 2.7.

The working of the self interpreter is best viewed as the series of state transformations in Figure 2.8

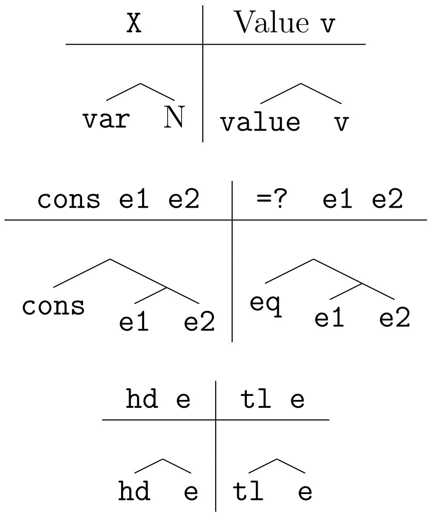


Figure 2.7: $\mathcal{I}$ expression encoding

```
(((sequence.(c1.c2)).Cs).(If.(St.Vo)))    ⇒  ((c1.(c2.Cs)).(If.(St.Vo)))
(((assign.e).Cs).(If.(St.Vo)))            ⇒  ((e.((doassign.N).Cs)).(If.(St.Vo)))
(((while.(e.c)).Cs).(If.(St.Vo)))         ⇒  ((e.((dowhile.N).((while.(e.c)).Cs))).
                                               (If.(St.Vo)))

(((var.N).Cs).(If.(St.Vo)))               ⇒  (cs.(If.((Vo.St).Vo)))
(((val.v).Cs).(If.(St.Vo)))               ⇒  (cs.(If.((v.St).Vo)))
(((hd.e).Cs).(If.(St.Vo)))                ⇒  ((e.((dohd.N).cs)).(If.(St.Vo)))
(((tl.e).Cs).(If.(St.Vo)))                ⇒  ((e.((dotl.N).cs)).(If.(St.Vo)))
(((cons.(e1.e2)).Cs).(If.(St.Vo)))        ⇒  ((e1.(e2.((docons.N).cs))).(If.(St.Vo)))
(((eq.(e1.e2)).Cs).(If.(St.Vo)))          ⇒  ((e1.(e2.((doeq.N).cs))).(If.(St.Vo)))

(((doassign.N).Cs).(If.((v.St).Vo)))      ⇒  (Cs.(If.(St.v)))
(((dowhile.N).((while.(e.c)).Cs)).
 (If.((N.St).Vo)))                        ⇒  (Cs.(If.(St.Vo)))
(((dowhile.N).(.Cs)).(If.((v.St).Vo)))    ⇒  ((c.((while.(e.c)).Cs)).(If.(St.Vo)))
(((dohd.N).Cs).(If.((v.St).Vo)))          ⇒  (Cs.(If.((hd v.St).Vo)))
(((dotl.N).Cs).(If.(St.Vo)))              ⇒  (Cs.(If.((tl v.St).Vo)))
(((docons.N).Cs).(If.((v2.(v1.St)).Vo)))  ⇒  (Cs.(If.(((v1.v2).St).Vo)))
(((doeq.N).Cs).(If.((v2.(v1.St)).Vo)))    ⇒  (Cs.(If.(((=?  v1 v2 ).St).Vo)))

(N.(If.(St.Vo)))                          ⇒  Vo
```

Figure 2.8: $\mathcal{I}$ state transformations

The state transitions are implemented as a series of nested “if” statements, where the if statements are coded as in Section 2.1.4. The complete self interpreter is given in appendix A.1.5

Interpretation overhead

$\mathcal{I}$ has the following characteristics:

1. Finite set of constructors: `cons` and `nil`.
2. Finite data branching: `cons` has arity 2, `nil` has arity 0
3. Finite number of variables: `X`
4. Finite number of functions: 1
5. Finite branching of code: `while` has 2 branches

By discussion in Section 1.3.2 this is sufficient for $\mathcal{I}$ to have the PIIO property. Table 2.1 supports the claim. In that table the ratio of the slowdown is computed and listed.

	time			ratio	
`name`	`int`	`sint`	`sint2`	`sint / int`	`sint2 / sint`
Assign val	2	1206	1269657	603	1052
Sequence	7	4216	4451042	602	1055
while-none	2	1466	1544382	733	1053
while once	8	6668	7037121	833	1055
While many	13	12715	13422077	978	1055
VAR	2	1138	1198361	569	1053
CONS	4	2804	2960177	701	1055
HD value	3	2553	2690925	851	1054
TL VAR	3	2689	2834945	896	1054
EQ true	4	3212	3390333	803	1055
EQ false	4	3212	3390333	803	1055
Rev List	40	38058	40186010	951	1055
AppendSD	130	134038	141526681	1031	1055
AppendDS	134	138602	146346349	1034	1055
DeadCode	26	24641	26015317	947	1055

Table 2.1: $\mathcal{I}$ interpretation slowdown ratio

It appears that the slowdown is bounded from above with a factor of approximately 1055. Looking at the self interpreter in Appendix A.1.5, this overhead seems resonable.

The reason for the relatively large difference between the `sint / int` ratios of the small programs versus the large programs, are based on the fact that some of the constructs of $\mathcal{I}$ requires multiple passes through the main loop of the self interpreter, and large programs use more of these constructs.

This difference in the `sint / int` ratio could be wieved as if the self interpreter does not have the PIIO property. If we look at the same programs when the self interpreter interprets itself and compute the `sint2 / sint` ratio, the difference is eliminated. Adding an extra layer of Interpretation should not hide the problem with the PIIO property if it was there, but rather enhance the visibility of it.

2.2 Specializing $\mathcal{I}$

2.2.1 Specialization

Since $\mathcal{I}$ only have one variable it must be dynamic except for the most simple programs. Since it is dynamic we must look for something else to exploit if we are to expect an increase in efficiency.

This raises the question of the validity of limiting of the number of variables to one. This problem is with the PIIO property. If we had a greater but fixed number of variables available, I would still have the PIIO property and we might be better of when specializing.

This argument is valid except that it will always be possible to write programs where the number of fixed variables are not enough and the technique of packing variables introduces in Section 2.1.5 must be used to account for the rest. The specializer apears to be simpler to implement if the one variable restriction is maintained.

Extending the runtime environment with a fixed number of read only variables could make it easier to identify the static parts, as the static parts would be prime candidates for data to be stored in the read only variables. The problem is again that any given program might use more static parts than there are read only variables, and then we would need to revert to the packing solution anyway. It just seem simpler to stay with the one variable approach.

The structure of the variable can be exploited, if one or more subtrees are static or of bounded static variation, then this can be used when specializing $\mathcal{I}$. The specialization of $\mathcal{I}$ raises other problems (See Section 2.4.1). These problems are so severe that the specialization attempt was abbandoned in favor of IF before exploiting the variable structure became relevant.

The specializer must be given information about which parts of the input to the program to specialize that are static. This could be done in at least two ways

- Static division: The left child of the input is the static part, the right contains the dynamic part.
- Pass an argument with the division.

The first aproach is used for specializing I.

The specializer will need to do an abstract interpretation of the program to specialize. Since it cannot track a complete state, as the input to the program is partial, an abstract view of the store is needed. This abstract view is encoded in a data structure called an Abstract Value.

Before the specialization is described, the data type used to track the the state of the variable, called an AbstractValue in the presence of unknown data.

The specialization of the actual specialization is based on the tasks identified in Section 1.5.

AbstractValue

The Abstract Value is a data structure for tracking changes to a binary tree with unknown parts. The idea behind the Abstract Value is to enable us to record changes to the tree, even if the changes involve the unknown part. As such is can be viewed as a hybrid between the the Expressions and Values of I.

The six constructors of the Abstract Value is listed in Table 2.2.

`AD`	The placeholder for dynamic content
`AV v`	lift a Value v to an Abstract Value
`ACONS( av1, av2 )`	construct a pair of two abstract values.
`AEQ( av1, av2 )`	compare two abstract values
`AHD( av1 )`	project the first component of an abstract value
`ATL( av1 )`	project the second component of an abstract value

Table 2.2: $\mathcal{I}$'s Abstract Value constructors

The relation between Values and Abstract Values can be expressed usign the standard subset notation.

$$\text{Value} \subset \text{Abstract Value}$$

When combining an abstract value with the missing dynamic value, an ordinary value is obtained.

Specialization of I

I programs can be viewed as a sequence of two types of constructs, one that manipulates the store, and one that controls the program flow.

This allow us to view the specialization as in the general considerations for specializing a self interpreter in Section 1.5.6.

The five tasks of specialization from Section 1.5 do this:

Binding time improvements This step consists of a program transformation that orders the sequencs of the program in the following way. The program `c1; c2; c3` can be encoded in I either like

1. `( c1, ( c2, c3 ) )` or
2. `( ( c1, c2 ), c3 )`

The binding time improvement step orders all sequencing like 1.

Binding time analysis Binding time analysis is not used by the $\mathcal{I}$ specializer.

Program annotation The $\mathcal{I}$ specializer is an online one, therefore the program annotation is not used. Decisions are made for while commands on the fly, based on the staticness of the condition expression of the while command.

Specializing The specializing function `spec( source )` is defined by the entries in the Table 2.3. The column "`source`" is the program snippet to specialize, the column "condition" is the predicates to be satisfied in order for that row to be used and the column "`target`" is the result of the specialization.

`source`	condition	`target`
`X := e`		`X := e`
`c1; c2`		`spec( c1 ); spec( c2 )`
`while e do c`	e is static, e = N	`X := X`
`while e do c`	e is static, e <> N	`spec( c; while e do c )`
`while e do c`	e is dynamic	`while e do c`

Table 2.3: $\mathcal{I}$ command specialization

It could rightly be argued that this is not really specialization, but only static loop unrolling. The reason nothing more is feasible is that it is next to impossible to move code in or out of a while body.

The problem is mainly that while commands are used to implement if-then-else commands. The particular implementation used in the self interpreter from Appendix A.1.5, could be found by the specializer and exploited. This analysis would be bound to that particular if-then-else implementation, implementing it in any other manner would make the analysis pointless.

Transition compression Two types of transition compression is done

- Skip commands are eliminated when possible.
- A sequence of assignments are collapsed into one assignment.

Specializer implementation The specializer is implemented in ML and it is listed in Appendix A.1.6

2.2.2 Example: Specializing append

The classical append program gives rise to two different specializations, one where the first of the two input lists is static and the other dynamic, and one where the first list is dynamic and the second is the static one.

Since the specializer assumes that the left part of the input is the static part and the right part is the dynamic one. The specialization of append with the first list as dynamic requires us to append the program with an assignment that switches the input around.

Append(S, D)

The append program given in Figure 2.9 is a program that takes two argument lists, one in each branch of the input variable.

```
X := cons( N, X );
while( hd( tl( X ) ) ) do
  X := cons(
          cons( hd( hd( tl( X ) ) ), hd( X ) ),
          cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )


static data: <<N.N>.<<N.<N.N>>.<<N.<N.<N.N>>>.N>>>
```

Figure 2.9: $\mathcal{I}$ append program

Specializing this program witht the static input also given in Figure 2.9 gives the residual program given in Figure 2.10. In the first while loop the conditional depend on the static part of the input. The body of the loop does not do any computation on the dynamic part, it is just passed along to later parts of the program. We would expect this first loop to be completely unrolled by the specializer as it does not depend on the dynamic part. After the first while loop, the reversed first list is accessed as `HD X` and the second list as `TL( TL( X ) )`. The second while loop is controlled by the reversed first list, and the body of the second while loop only references the whole second list. The second while loop is also expected to be completely unrolled. The expectation to the residual program in Figure 2.10 is that it would not use any while loops, which it does not.

```
X := cons( <N.N>, cons( <N.<N.N>>, cons( <N.<N.<N.N>>>, X ) ) )
```

Figure 2.10: $\mathcal{I}$ append, residual program

Append(D, S)

Since the specializer assumes the static parts of the input to the program is in the left child, switching the data must be done by an assignment statement before executing the append program, this results in the program in Figure 2.11.

Specializing this program gives the residual program in Figure 2.12. This is not unexpected as now the two while loops are controlled by the dynamic list. Therefore the residual program should contain the two while loops.

```
X := cons( N, cons( tl( X ), hd( X ) ) );
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )

Static data: <<<N.N>.N>.<<<<N.N>.N>.N>.N>>
```

Figure 2.11: $\mathcal{I}$ append program with data switched

```
X := cons( N, cons( X, <<<N.N>.N>.<<<<N.N>.N>.N>.N>> ) );
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

Figure 2.12: $\mathcal{I}$ append, residual program, data switched

2.3 $\mathcal{I}$ implementation correctness

The correctness tests for the $\mathcal{I}$ implementation is split into two seperate steps, one for the interpreters and one for the specializer.

2.3.1 Interpreter correctness

The subset, in Table 2.4 of the example programs in Appendix A.2 is used to test the implementation of each language construction in both the timed interpreter in ML and the self interpreter. The result is given in Table 2.5

The programs are so simple that the expected output can be hand generated. The input and expected output is also given in Appendix A.2. The program input is constructed from the static and dynamic part by constructing a pair `(static.dynamic)`.

Name	Description
Assign val	Use of a static value
Sequence	sequencing of commands
while none	while body not executed
while once	while body executed once
while many	while body executed more than once
VAR	reference current variable content.
CONS	Test pair construction
HD VAR	hd of variable
HD value	take hd of value
TL VAR	take tl of variable
EQ true	comparison returns true
EQ false	comparison returns false

Table 2.4: $\mathcal{I}$ implementation correctness tests

2.3.2 Specializer correctness

The correctness of the specializer is not proven formally. Instead we exercise the specializer using all the programs in Appendix A.2. The specializer is exercised both on the programs themselves and on the self interpreter interpreting the programs or the self interpreter.

The summary of the execution is given in table 2.6

	result	
name	`int`	`sint`
Assign val	OK	OK
Sequence	OK	OK
while-none	OK	OK
while once	OK	OK
While many	OK	OK
VAR	OK	OK
CONS	OK	OK
HD value	OK	OK
TL VAR	OK	OK
EQ true	OK	OK
EQ false	OK	OK

Table 2.5: $\mathcal{I}$ implementation correctness test results

	result			
`name`	`int`	`spec`	`spec-sint`	`spec-sint2`
Assign val	OK	OK	OK	OK
Sequence	OK	OK	OK	OK
while-none	OK	OK	OK	OK
while once	OK	OK	OK	OK
While many	OK	OK	OK	OK
VAR	OK	OK	OK	OK
CONS	OK	OK	OK	OK
HD value	OK	OK	OK	OK
TL VAR	OK	OK	OK	OK
EQ true	OK	OK	OK	OK
EQ false	OK	OK	OK	OK
Rev List	OK	OK	OK	OK
AppendSD	OK	OK	OK	OK
AppendDS	OK	OK	OK	OK
DeadCode	OK	OK	OK	OK

Table 2.6: $\mathcal{I}$ specialization correcness test result

2.4 $\mathcal{I}$ experiments

To test implementation and generate data for comparison in order to establish the JOI property a number of small test programs was written. The test programs and their input data are listed in appendix A.2. Table 2.7 summarizes the execution of the tests.

All tests are run on the timed ML interpreter.

The following combinations are used in the test:

- `i`: execute `p` directly on the timed interpreter.
- `si`: execute the self interpreter on the program `sint` with the input `(p.d)`.
- `si2`: execute the self interpreter `sint` with the input `(sint.(p.d))`. That is the self interpreter interprets itself.
- `s`: Let `p'` = `spec(p.s)`, and execute `p'` with input `d`.
- `s-si`: Let `sint'` = `spec(sint.p)`, and execute `sint'` with input `d`.
- `s-si2`: Let `sint2` = `spec(sint.sint)`, and execute `sint2` with input `d`.

2.4.1 Status of $\mathcal{I}$ specialization

Looking at the execution time data in Table 2.7 it is clear that the problem of specialization occurs when while commands are specialized. This is not surprising, as this is the only place where the code branches.

Specialization problems identified

Specializing $\mathcal{I}$ gives rise to the following problem.

Problem: After a while command it is impossible to say anything about the content of `X`.

The reason for this is that we have only one variable. This makes all updates of the state be updates of the total state. It is possible that an assignment in a while body while change the structure of the store completely.

This makes it impossible to do any specialization of the commands that comes after a while command.

	time					
`p`	`i`	`si`	`si2`	`s`	`s-si`	`s-si2`
Assign val	2	1206	1269657	2	2	1042531
Sequence	7	4216	4451042	4	4	3678552
while-none	2	1466	1544382	4	2	1270174
while once	8	6668	7037121	2	3855	5821349
While many	13	12715	13422077	4	8867	11111830
VAR	2	1138	1198361	4	2	983453
CONS	4	2804	2960177	8	4	2443263
HD value	3	2553	2690925	2	2	2220174
TL VAR	3	2689	2834945	2	3	2339506
EQ true	4	3212	3390333	6	4	2799691
EQ false	4	3212	3390333	6	4	2799691
Rev List	40	38058	40186010	40	27067	33287960
AppendSD	130	134038	141526681	8	105322	117257227
AppendDS	134	138602	146346349	132	105329	121250733
DeadCode	26	24641	26015317	31	18427	21546362

Table 2.7: $\mathcal{I}$ time test

Problem: Coding if-then-else as described in Section 2.1.4 makes it more difficult to obtain Jones Optimality.

The reason is that the IF stack described in Section 2.1.4 must be removed by the specializer if the encoded if is a static if.

Example: The self interpreter in Section 2.1.5 makes use of an if stack. The structure of the IF stack depends solely on the self interpreter. This makes it a prime candidate for elemination during specialization. One of the difficult thing with this removal is that the condition for choosing the `then` branch might depend on the unknown input, and therefore the content of the IF stack must be declared dynamic, and as such it cannot be expected to be removed by the specialization task.

Solution: Add if-then-else construct to language.

Problem: A while conditional will be the result of evaluating an expression. Since expressions allow the `=?` construct, we cannot in general say anything about the content of the store either before or after the loop. This will complicate the specializing.

Possible solution: Change Expressions to allow only lookup of data in the variable. Remove the comparison expression.

2.5 $\mathcal{IF}$

$\mathcal{IF}$ is a new language build by extending $\mathcal{I}$ with if-then-else commands and removing the comparison expression. This is two of the changes suggested in Section 2.4.1.

As it was done for I, the syntax and semantics of $\mathcal{IF}$ is shown. After seing a small coding example of a program in $\mathcal{IF}$, the interpretation of $\mathcal{IF}$ is presented. The interpretation section includes a discussion of $\mathcal{IF}$ and the PIIO property.

Before the specialization of if is show, a list of program transformations for $\mathcal{IF}$ is listed. The program transformations are needed by various steps of the specialization process.

The presentation of the specialization process follows roughly the outline in Section 1.5.

The $\mathcal{IF}$ analysis is concluded with a series of tests that justify the correctness of the implemented interpreters and the specializer. A list of problems with the $\mathcal{IF}$ language is listed together with possible solutions.

2.5.1 Syntax

The syntax for $\mathcal{IF}$ is presented in Figure 2.13 in EBNF.

```
Command     ::=  X := Expression
              |  Command ; Command
              |  while Expression do Command
              |  if Expression Command Command
Expression  ::=  X
              |  val Value
              |  cons Expression Expression
              |  hd Expression
              |  tl Expression
Value       ::=  nil
              |  ( Value, Value )
```

Figure 2.13: $\mathcal{IF}$ syntax

The values for $\mathcal{IF}$ are simple binary trees with nil as the only possible leaf value. These values make up all possible binary trees, and it is called $\mathbb{D}$ by Jones (Jones, 1997, Definition 2.1.1).

```
X := cons( N, X );
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

Figure 2.14: $\mathcal{IF}$ list append implementation

Append implementation

The list append program in Figure 2.14 is the $\mathcal{IF}$ implementation corresponding to the $\mathcal{I}$ implementation of append in Figure 2.9

2.5.2 Semantics

The semantics of $\mathcal{IF}$ builds upon the semantics of I. The command judgment is extended with two rules for the if-then-else construct and the compare operator of the expression judgment is removed. The $\mathcal{I}$ command semantics is given in Figure 2.2. The two new rules are show in Figure 2.15

$$\frac{< e, X > \rightarrow N \qquad < c_2, X > \rightarrow X'}{< \texttt{if}(e, c_1, c_2), X > \rightarrow X'}$$

$$\frac{< e, X > \rightarrow v \qquad < c_1, X > \rightarrow X'}{< \texttt{if}(e, c_1, c_2), X > \rightarrow X'} v \neq N$$

Figure 2.15: $\mathcal{IF}$ Command semantics, based on I

2.5.3 Interpretation

The $\mathcal{IF}$ interpreter written in ML is build upon the interpreter for I, with new cases for the if-then-else construct. The cases for expression evaluation of comparison are removed. It works as one would expect. The timed interpretation is build from the following functions

- `Value.applyHd`, Appendix B.1.1
- `Value.applyTl`, Appendix B.1.1
- `Expression.timed_evaluate`, Appendix B.1.3
- `Command.timed_interpret`, Appendix B.1.4

2.5.4 Self interpretation

The self interpreter is build using the self interpreter for $\mathcal{I}$ as inspiration. Since the language now supports if-then-else commands directly we do not need the IF-stack used by I. The Packed variables for the $\mathcal{IF}$ self interpreter is structured as shown in Figure 2.16.

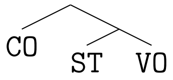


Figure 2.16: $\mathcal{IF}$ self interpreter variable packing

If commands are encoded as shown in Figure 2.17.

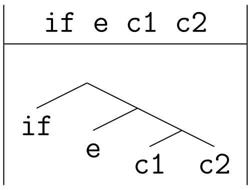


Figure 2.17: If command encoding

The state transformation for $\mathcal{IF}$ are given in Figure 2.18.

```
(((sequence.(c1.c2)).Cs).(St.Vo))       ⇒  ((c1.(c2.Cs)).(St.Vo))
(((assign.e).Cs).(St.Vo))               ⇒  ((e.((doassign.N).Cs)).(St.Vo))
(((while.(e.c)).Cs).(St.Vo))            ⇒  ((e.((dowhile.N).((while.(e.c)).Cs))).(St.Vo))
(((if.(e.(c1.c2))).Cs).(St.Vo))         ⇒  ((e.(((doif.N).(c1.c2)).Cs)).(St.Vo))

(((var.N).Cs).(St.Vo))                  ⇒  (cs.((Vo.St).Vo))
(((val.v).Cs).(St.Vo))                  ⇒  (cs.((v.St).Vo))
(((hd.e).Cs).(St.Vo))                   ⇒  ((e.((dohd.N).cs)).(St.Vo))
(((tl.e).Cs).(St.Vo))                   ⇒  ((e.((dotl.N).cs)).(St.Vo))
(((cons.(e1.e2)).Cs).(St.Vo))           ⇒  ((e1.(e2.((docons.N).cs))).(St.Vo))
(((eq.(e1.e2)).Cs).(St.Vo))             ⇒  ((e1.(e2.((doeq.N).cs))).(St.Vo))

(((doassign.N).Cs).((v.St).Vo))         ⇒  (Cs.(If.(St.v)))
(((dowhile.N).((while.(e.c)).Cs)).
 ((N.St).Vo))                           ⇒  (Cs.(St.Vo))
(((dowhile.N).(.Cs)).((v.St).Vo))       ⇒  ((c.((while.(e.c)).Cs)).(St.Vo))
(((dohd.N).Cs).((v.St).Vo))             ⇒  (Cs.((hd v.St).Vo))
(((dotl.N).Cs).(St.Vo))                 ⇒  (Cs.((tl v.St).Vo))
(((docons.N).Cs).((v2.(v1.St)).Vo))     ⇒  (Cs.(((v1.v2).St).Vo))
(((doeq.N).Cs).((v2.(v1.St)).Vo))       ⇒  (Cs.(((=?  v1 v2 ).St).Vo))
((((doif.N).(c1.c2)).Cs).((N.St).Vo))   ⇒  ((c2.Cs).(St.Vo))
((((doif.N).(c1.c2)).Cs).((v.St).Vo))   ⇒  ((c1.Cs).(St.Vo))

(N.(St.Vo)))                            ⇒  Vo
```

Figure 2.18: $\mathcal{IF}$ state transformations

Interpretation overhead

$\mathcal{IF}$ has the following characteristics:

1. Finite set of constructors: cons and nil.
2. Finite set of data branching: cons has arity 2, nil has arity 0
3. Finite number of variables: X
4. Finite number of functions: 1
5. Finite branching of code: while has 2 branches, if has 2 branches

By Section 1.3.2 this is sufficient for $\mathcal{IF}$ to have the PIIO property. The Table 2.8 summarizes the ratios of comparing the time for executing various programs directly (`int`), by the self interpreter (`sint`) and by self interpreting twice (`sint2`).

	time			ratio	
`name`	`int`	`sint`	`sint2`	`sint / int`	`sint2 / sint`
Assign value	2	221	47010	110	212
Sequence	2	221	47010	110	212
while none	2	262	56199	131	214
while once	8	1177	257438	147	218
while many	43	7677	1697250	178	221
VAR	2	210	44550	105	212
CONS	4	520	113116	130	217
HD VAR	3	448	97345	149	217
TL value	3	496	108487	165	218
IF false	2	221	47010	110	212
IF true	2	221	47010	110	212
List Reverse	40	6916	1529445	172	221
DeadCode	2	221	47010	110	212
While opt fails	5	903	198110	180	219
Append	130	24279	5378410	186	221
Append 2	134	25113	5563708	187	221
Compare false	37	6699	1479723	181	220
Compare true	150	31151	6908181	207	221
KMP	26	4005	881712	154	220
KMP 1	31	5181	1142815	167	220
KMP 2	119	23566	5223900	198	221
KMP 3	131	26346	5840554	201	221
KMP 4	150	30020	6653764	200	221
KMP large true	488	102420	22723479	209	221
KMP large false	783	165586	36746676	211	221

Table 2.8: $\mathcal{IF}$ interpretation slowdown ratio

2.6 $\mathcal{IF}$ Program transformations

During the specialization a number of program transformations are used. This section is a catalog of the transformations. For each transformation is introduced with the following information:

- Source: The code the transformation is applied to
- Target: The code after the transformation
- Condition: The predicates that needs to be satisfied for this transformation to apply
- Optimization: It is analysed if the transformation is an optimization. If the target code executes using fewer steps than the source code it is categorised as an optimization. Possible choises: Yes, No, Equal. If a transformation has No for optimization, then the inverse of the transformation must be an optimization.

Some of the transformation substitute one expression `e1` for `X` in another expression `e2`. This is written as `e2( e1 )`

Each transformation is coded as a single function in `Command.sml` (see Appendix B.1.4). All the optimizing transformations are collected into the function `optimize`. This function keeps optimizing as long as each run through all the optimizations change the program. Since none of the optimizing transformations undo each other, the optimization process must terminate at some point.

2.6.1 Sequence order

The operational semantics of the sequence command gives us that the sequence command is associative. That is given three commands `c1`, `c2` and `c3` to be executed is this order, the sequence of the commands can be given as `(c1; c2); c3` or as `c1; (c2; c3 )`.

Source: `(c1; c2); c3`

Target: `c1; (c2; c3)`

Condition: None

Optimization: Equal

2.6.2 Inline

This transformation inlines the commands after an if command into both branches of the if command.

Source:

```
( if e
  then c1
  else c2 );
c3
```

Target:

```
if e
then c1; c3
else c2; c3
```

Condition: None

Optimization: Equal

2.6.3 Remove skip lhs

This transformation removes a skip instruction if it occurs in the left leg of a sequence command.

Source:

```
X := X;
c1
```

Target:

```
c1
```

Condition: None

Optimization: Yes

2.6.4 Remove skip rhs

Remove a skip instruction if it occurs in the right leg of a sequence command.

Source:

```
c1;
X := X
```

Target:

```
c1
```

Condition: None

Optimization: Yes

2.6.5 Join assigments

When two assignments follow one another, the first canbe substituted for X in the second.

Source:

```
X := e1;
X := e2;
c1
```

Target:

```
X := e2( e1 );
c1
```

Condition: None

Optimization: Yes or Equal

2.6.6 Push pre assign into if

This transformation pushed an assignment occuring right before an if statement into both branches of the if, while it substitutes the assigned expression into the if selection expression.

Source:

```
X := e1;
if e2
then c1
else c2
```

Target:

```
if e2( e1 )
then X := e1; c1
else X := e2; c1
```

Condition: None

Optimization: Yes or Equal

2.6.7 Interpret rest

If at any point an assignment of a static value occurs, the code after the assignment can be fully evaluated.

Source:

```
X := v;
c1
```

Target:

```
X := interpret( c1, v )
```

Condition: None

Optimization: Yes

2.6.8 if condition nil

If the condition in the if statement is a static nil value, then remove if in favor of else branch.

Source:

```
if e
then c1
else c2
```

Target:

```
c2
```

Condition: e = N

Optimization: Yes

2.6.9 if conditional true

If the if conditional is either a value different than nil or a cons expression, then the then branch is always taken.

Source:

```
if e
then c1
else c2
```

Target:

```
c1
```

Condition: e = cons(e1, e2) or (e = v and v <> N)

Optimization: Yes

2.6.10 Nested while

If a while body is an if command, and the else branch of that if contains another while, and the condition of the two whiles are the same, then we know that the execution of the else branch will break the outer while loop.

Source:

```
while e1
  if e2
  then c1
  else
    c2;
    while e3
      c3
```

Target:

```
if e1
then
  while e2
    c1;
  c2;
  while e3
    c3
else
  X := X
```

Condition: e1 = e3

Optimization: No

2.6.11 While if to if while

If a while commands body is an if which have an else branch which sets the while condition to N, then the while and if can be combined to a single while.

Source:

```
while e1
  if e2
  then c1
  else X := e3
```

Target:

```
if e1
then
  while e2
    c1;
  X := e3
else
  X := X
```

Condition: `e1( e3 ) = N`

Optimization: No

2.6.12 While body once

If the body of a while is an assignment, and that assignment causes the while loop to terminate, then it can be unrolled

Source:

```
while e1
  X := e2
```

Target:

```
if e1
then X := e2
else X := X
```

Condition: `e1( e2 ) = N`

Optimization: This is undecideable. If e1 is false it is not an optimization, if e1 is true, it is an optimization. This will actually cause problems for a certain programs.

This problem can be removed by introduction of a new command, the if-then command. If that is done, then this transformation is execution time neutral. The addition should not pose any difficulty for the specialization.

2.6.13 Remove dead left side of sequence

If the right side of a sequence command is assignment of a static value, then the left part is dead code.

Source:

```
c1;
X := v
```

Target:

```
X := v
```

Condition: None

Optimization: Yes

2.7 Specializing $\mathcal{IF}$

2.7.1 Specialization

The five tasks of Section 1.5 is used as a foundation for specializing $\mathcal{IF}$.

Since we only have one variable we need to exploit parts of `X` that have bounded static variation. The single variable `X` of $\mathcal{IF}$ also means a small change to how the specializer works when compared to a traditional specializer.

The traditional specializer works by keeping track of a store with current values for all the variables of the specialized program. Since we have only X we need to do something else.

The program to specialize is augmented to incorporate the static parts of the data. If the arguments to the specializer is `(prog.static)` the augmented program is `X:=cons( static, X ); prog`. This program is semantically equivalent to the result of the specialization. The rest of the steps in the specialization can be viewed as a series of program transformations.

Binding time improvement

The binding time improvements for $\mathcal{IF}$ falls into two categories:

- ordered sequences: There are two ways to handle the sequences of $\mathcal{IF}$, one is to handle it implicitly using some feature of the language the interpreter is implemented in or explicitly by coding the handling into the interpreter. Since we need a self interpreter at some point, it apears to be easier to implement it explicitly. The point is that the sequencing of three commands `c1; c2; c3` can be done in two ways, depeding on how the sequence commands are places.

  1. SEQ(c1, SEQ(c2, c3))
  2. SEQ(SEQ(c1, c2), c3)

  The semantics of the language is indifferent, as the execution sequence of the commands are always the same. The first one resembles the idea that a command is followed by the rest of the program / body of while / branch of if. The first binding time improvement is to order the sequences as in the first example.

- inline code following an if command into the two branches: The code

  ```
  if( e )
  ```

```
then c1
else c2
c3
```

is transformed into

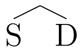
```
if( e )
then c1; c3
else c2; c3
```

Binding time analysis

The binding time analysis figures out which parts of `X` that are static or dynamic. This information is used when deciding what to do about `if` and `while` commands.

Traditionally the binding time analysis results in a division, which splits variables into the categories: static and dynamic. Since $\mathcal{IF}$ have one variable we use a different approach, the result of the binding time analysis is a *partition* of `X` into the parts which are either static or have bounded static variation and the parts that depend on the dynamic part of input.

Partition: a Partition is a binary tree with the leafs S for static and D for dynamic. The specializer assumes that the partition of the data to the program to specialize is



The assignment command is the only one that can change the partition of `X`.

The binding time analysis works by pushing a given partition through a command, resulting in a new partition. Since the language supports code branching the partitions must be combined in some way. The partition transitions for each type of command is given in Figure 2.19 and the partition combination rules are given in Figure 2.21. The Command transitions makes use of the expression transitions from Figure 2.20. The function call notation is used to apply a command or expression to a partition, the result of the application is a in both cases a new partition.

Program annotation

The while commands are annotated with partition information. The while command partition is generated as shown in Figure 2.19 is an invariant for the while command.

Command	input partition	output partition
`X := e`	`p`	`e( p )`
`c1; c2`	`p`	`c2( c1( p ) )`
`if( e, c1, c2 )`	`p`	`combine( c1( p ), c2( p ) )`
`while( e, c )`	`p`	`combine( p, c( p ) )`

Figure 2.19: $\mathcal{IF}$ command partition transitions

Expression	input partition	output partition
`X`	`p`	`p`
`value`	`p`	`S`
`hd e`	`S`	`S`
`hd e`	`D`	`D`
`hd e`	`[p1.p2]`	`p1`
`tl e`	`S`	`S`
`tl e`	`D`	`D`
`tl e`	`[p1.p2]`	`p2`
`cons( e1, e2 )`	`p`	`[e1( p ).  e2( p )]`
`cons( e1, e2 )`	`p`	`S if e1( p ) = S and e2( p ) = S`

Figure 2.20: $\mathcal{IF}$ Expression partition transitions

p1	p2	result
`S`	`S`	`S`
`S`	`D`	`D`
`S`	`[pc.pd]`	`[pc.pd]`
`D`	`S`	`D`
`D`	`D`	`D`
`D`	`[pc.pd]`	`[pc.pd]`
`[pa.pb]`	`S`	`[pa.pb]`
`[pa.pb]`	`D`	`[pa.pb]`
`[pa.pb]`	`[pc.pd]`	`[combine(pa,pc).combine(pb,pd)]`

Figure 2.21: Combining two partitions

```
X := cons( N, X );
while [[ S, [ S, D ] ]] ( hd( tl( X ) ) ) do
  X := cons(
  cons( hd( hd( tl( X ) ) ), hd( X ) ),
  cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
 );
X := cons( hd( X ), tl( tl( X ) ) );
while [[ S, [ S, D ] ]] ( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

Figure 2.22: $\mathcal{IF}$ append (SD) program with annotations

```
X := cons( N, X );
while [[ [ D, S ], [ D, S ] ]] ( hd( tl( X ) ) ) do
  X := cons(
  cons( hd( hd( tl( X ) ) ), hd( X ) ),
  cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
 );
X := cons( hd( X ), tl( tl( X ) ) );
while [[ [ D, S ], [ D, S ] ]] ( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

Figure 2.23: $\mathcal{IF}$ append (DS) program with annotations

An example of a program with annotations is the familiar append function in Figure 2.22

The annotation of the same program but with the staticness of the input reversed is given in Figure 2.23

The effect of the annotation is quite clear, and if working through a the body of one of the while loops it can be verifies that the partition given is indeed an invariant for the while.

Specialization

The major issue when specializing $\mathcal{IF}$ is how to detect loops occuring during the abstract interpretation of the program to specialize.

Loop detection: Loop detection is the act of detecting when to put a loop into the residual program. When specializing a while command there are basically three possible actions:

- while condition is statically determined to be `N`
- while condition is statically determined to be `(v1.v2)`
- while condition is dynamic.

The first and the last case is the "easy" ones. If the condition is `N`, then the while command is specialized with a `skip` command and the loop body is not entered. If the while condition is dynamic, we copy the while loop into the residual program, without futher work done.

In case the while condition is a static pair, the obvious thing to do is to unroll the loop once and specialize the new sequence of the body followed by the while loop. This is the case where the loop detection is needed. This is illustrated by the specialization of the self interpreter. When specializing the self interpreter with a program with loops, we need to stop the specialization when the code part of the self interpreter state contains a value we have seen before. This of cource depends on the particular implementation of the self interpreter.

The problem here is to find a condition that is strong enough to detect all loops, while allowing all necessary specialization to take place.

The condition used here is to use the partition information the loops are annotated with, and for each unrolling of the loop we remember the values of the paths of the state that are marked static by the partition. When we reach a point in the unrolling where the static parts of the state are equal to a state seen before, we have reached a loop.

This works well for the self interpreter, as a loop in the interpreted program will cause the code part of the self interpreter state to grow, and at some point it will be back to when the loop was encountered.

Specializing $\mathcal{IF}$ The specializer works by abstractly interpreting the program while it maintains an abstract view of the store, and the current partition.

The partition is used when decisions about if and while commands are made. If the partition says the condition is dynamic, it is treated as dynamic regardless of the abstract value of the condition. If it is static by the partition the actual value is examined to determine the correct behavior.

The specializer store representation is the same abstract value used when specializing $\mathcal{I}$ (see Section 2.2.1), with the modification that the AEQ constructor is removed as $\mathcal{IF}$ does not support the comparison operation.

The specialization of the commands are listed in Table 2.9.

program `p`	condition	action
`X := e`		`X := e`
`c1; c2`		`spec( c1 ); spec( c2 )`
`while e do c`	partition(e) = S e = N	`X := X`
`while e do c`	partition(e) = S e <> N loop *not* detected	`spec( c; while e do c )`
`while e do c`	partition(e) = S e <> N loop detected	`X := X`
`while e do c`	partition(e) = D	`while e do spec( c )`
`if e c1 c2`	partition(e) = S e = N	`spec( c2 )`
`if e c1 c2`	partition(e) = S e <> N	`spec( c1 )`
`if e c1 c2`	partition(e) = D	`if e spec( c1 ) spec( c2 )`

Table 2.9: $\mathcal{IF}$ command specialization: `spec( p )`

Transition compression

The transition compression is a number of specific program transformations whose goal is to reduce the number of transitions in the program. The catalog of transformations is given in Section 2.6.

Specialization process

The general algorithm for specializing $\mathcal{IF}$ is:

```
spec( prog, static ) =
  augment prog
  order sequences
  compress transitions
  inline code following if statements
  annotate whiles
  specialize
  compress transitions
```

Implementation

The implementation (See Appendix B.1.9) of the specialization process follows roughtly the outline above. As an extra safety it is possible to specify the maximum number of while loop unrollings the specializer is allowed to use. This is done to cope with the problem that the implementation of a string matching algorithm makes the specializer loop.

2.7.2 Example

The example of the $\mathcal{IF}$ specialization is done usign the append program from Figure 2.14. The example covers both the specialization with the two lists given with the static first and the dynamic first. The last is the result of specializing the self interpreter with the append program.

Append [S, D]

Specializing the append program in Figure 2.14 with the static list

```
static data: <<N.N>.<<N.<N.N>>.<<N.<N.<N.N>>>.N>>>
```

as the first input list gives the residual program

```
X := cons( <N.N>, cons( <N.<N.N>>, cons( <N.<N.<N.N>>>, X ) ) )
```

As would be expected it is a single assignment, which conses the elements of the first list to the second list, one by one. This is exactly as it happended for the $\mathcal{I}$ specialization of the corresponding append implementation in $\mathcal{I}$.

Append [D,S]

Specializing the append program with a static list as the second input, requires a small addition to the append program. The program specialized is:

```
X := cons( N, cons( tl( X ), hd( X ) ) );
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

The result of the specialization is:

```
X := cons( N, cons( X, <<<N.N>.N>.<<<<N.N>.N>.N>.N>> ) );
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

Which is similar to the result for I, not much can be precomputed, and the result is basically the input program.

Specializing self interpreter with append

The last interesting example is the result of specializing the self interpreter with the append program. This gives the following residual code:

```
X := cons( N, X );
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

This is an exact copy of the append program illustrating that the original append program is an optimal implementation of append.

A common problem when specializing self interpreters is the limits the interpreter imposes on the residual program. If these limites are not coped with in some way the residual program will not be optimal.

One of the more obvious inherited limits in this case is the number of while loops. If the interpreter imposed this inherited limitation on the residual program it would not be possible to generate the two loops as shown.

2.8 $\mathcal{IF}$ implementation correctness

The test for correctness of the implementation is again split into two parts, one for the interpreters and one for the specializer.

2.8.1 Interpreter correctness

The test for correctness of the implementation of the interpreters is again done using a subset of the test programs in Appendix B.2. The execution of the subset is summarized in Table 2.10. The subset of the implemented tests is chosen in such a way that each language construction of $\mathcal{IF}$ is exercised at least once. For branching constructs the tests cover all branches.

	result	
name	`int`	`sint`
Assign value	OK	OK
Sequence	OK	OK
while none	OK	OK
while once	OK	OK
while many	OK	OK
VAR	OK	OK
CONS	OK	OK
HD VAR	OK	OK
TL value	OK	OK
IF false	OK	OK
IF true	OK	OK

Table 2.10: $\mathcal{IF}$ implementation correctness test results

2.8.2 Specializer correctness

The specializer is executed for various combinations of the test programs (see Appendix B.2) and the self interpreter (See Appendix B.1.8). The result is summarized in Table 2.11.

The results of the correctness tests for the specializer shows that when specializing an implementation of a string matching algorithm (KMP), the specializer loops. This is indicated with $\perp$ in the table.

From the results of the table we see that it must be some property of the string matching program that causes the specializer to loop as the program runs fine on the result of specializing the self interpreter with itself. This is expected behavior if a program makes the specializer loop.

The string matching program makes use of nested while loops, and it is the only one of the implemented programs that does that. Futher analysis of the situation has showed that the problem is with the loop detection mechanism.

2.8.3 Optimizer correctness

The execution of the optimizer is again tested using various combinations of the test programs in Appendix B.2 and the self interpreter from Appendix B.1.8. The result is summarized in Table 2.12

For all the timing tests, the optimizer results in equal or lower execution time. The optimizations implemented backs the expectations from the analysis in the Section on program transformation (Section 2.6).

	result			
`name`	`int`	`spec`	`spec-sint`	`spec-sint2`
Assign value	OK	OK	OK	OK
Sequence	OK	OK	OK	OK
while none	OK	OK	OK	OK
while once	OK	OK	OK	OK
while many	OK	OK	OK	OK
VAR	OK	OK	OK	OK
CONS	OK	OK	OK	OK
HD VAR	OK	OK	OK	OK
TL value	OK	OK	OK	OK
IF false	OK	OK	OK	OK
IF true	OK	OK	OK	OK
List Reverse	OK	OK	OK	OK
DeadCode	OK	OK	OK	OK
While opt fails	OK	OK	OK	OK
Append	OK	OK	OK	OK
Append 2	OK	OK	OK	OK
Compare false	OK	OK	OK	OK
Compare true	OK	OK	OK	OK
KMP	OK	OK	OK	OK
KMP 1	OK	$\perp$	$\perp$	OK
KMP 2	OK	$\perp$	$\perp$	OK
KMP 3	OK	$\perp$	$\perp$	OK
KMP 4	OK	$\perp$	$\perp$	OK
KMP large true	OK	$\perp$	$\perp$	OK
KMP large false	OK	$\perp$	$\perp$	OK

Table 2.11: $\mathcal{IF}$ specialization correctness test result

	result			
`name`	`int`	`opt int`	`sint`	`opt( sint )( p )`
Assign value	2	2	533	221
Sequence	5	2	1110	463
while none	2	2	623	262
while once	8	8	2877	1177
while many	43	43	18635	7677
VAR	2	2	503	210
CONS	4	4	1258	520
HD VAR	3	3	1093	448
TL value	3	3	1180	496
IF false	4	2	1289	598
IF true	4	2	1289	598
List Reverse	40	40	16683	6916
DeadCode	26	2	10747	4469
While opt fails	5	5	1996	903
Append	130	130	58836	24279
Append 2	134	134	60858	25113
Compare false	37	37	15544	6699
Compare true	150	150	72962	31151
KMP	26	26	9556	4005
KMP 1	31	31	12226	5181
KMP 2	121	119	52668	22257
KMP 3	133	131	59101	25037
KMP 4	154	150	69279	29283
KMP large true	498	488	235701	99637
KMP large false	797	783	379079	160185

Table 2.12: $\mathcal{IF}$ optimizer correctness test result

2.9 $\mathcal{IF}$ experiments

The experiments for asserting the quality of the specializer is listed in Appendix B.2. The programs are executed in a number of combinations. The combinations are:

- `i`: execute the optimized program `p` directly on the timed interpreter.
- `si`: execute the optimized self interpreter on the optimized program.
- `si2`: execute the self interpreter with the self interpreter with the program. All programs are optimized.
- `s`: Execute the result of specializing the optimized program with the supplied static data.
- `s-si`: execute the specialization of the optimized self interpreter with the optimized program.
- `s-si2`: execute the result of specializing the optimized self interpreter with itself on the optimized program and provided data.

The summary of the executions are listed in Table 2.13.

If the specializer and self interpreter pair is Jones Optimal, the times in the `i` column should be greater or equal to the `s-si` column. The column `si` should also be greater or equal to the column `s-si2`.

time						
`p`	`i`	`si`	`si2`	`s`	`s-si`	`s-si2`
Assign value	2	221	47010	2	2	221
Sequence	2	221	47010	2	2	221
while none	2	262	56199	4	2	262
while once	8	1177	257438	2	5	1177
while many	43	7677	1697250	4	43	7677
VAR	2	210	44550	4	2	210
CONS	4	520	113116	8	4	520
HD VAR	3	448	97345	2	3	448
TL value	3	496	108487	2	3	496
IF false	2	221	47010	2	2	221
IF true	2	221	47010	2	2	221
List Reverse	40	6916	1529445	42	40	6916
DeadCode	2	221	47010	2	2	221
While opt fails	5	903	198110	6	7	854
Append	130	24279	5378410	8	130	24279
Append 2	134	25113	5563708	132	134	25113
Compare false	37	6699	1479723	7	35	6503
Compare true	150	31151	6908181	18	148	30465
KMP	26	4005	881712	2	2	3956
KMP 1	31	5181	1142815	⊥	⊥	5083
KMP 2	119	23566	5223900	⊥	⊥	23223
KMP 3	131	26346	5840554	⊥	⊥	25905
KMP 4	150	30020	6653764	⊥	⊥	29530
KMP large true	488	102420	22723479	⊥	⊥	100754
KMP large false	783	165586	36746676	⊥	⊥	162989

Table 2.13: $\mathcal{IF}$ time test

2.10 Conclusion

In Section 2.5 the language $\mathcal{IF}$ is introduced. A self interpreter for $\mathcal{IF}$ having the PIIO properter is produced in section 2.5.4 and in Section 2.7 a specializer for IF is described. The pair of the self interpreter and the specializer is extremely close to having the JOI property, failing only in two minor areas.

The correctness tests for the specializer in Section 2.8.2 revealed a problem with the loop detection mechanism. The discussion of the program transformations for $\mathcal{IF}$ in Section 2.6 showed that one of the transformations (While body once, Section 2.6.12) used causes problems with JOI.

Program transformation problem

The problem with the program transformation "while body once" is that it is impossible to classify it as an optimization. Depending on the condition used in either the while command or the corresponding if command, it will be an optimization to do the transformation in both directions.

Solution: To solve this problem I propose that another language construction is added to $\mathcal{IF}$. The new construction is an if with no alternative, a if-then command. The addition of this construction lets us reformulate the program transformation in question to

source:

```
while e1
  X := e2
```

target:

```
if e1
then X := e2
```

Condition: `e1( e2 ) = N`

Optimization: Yes

The addition of this new construct should not pose any problem to the specialization already discussed, as it is "just" a specialized version of the general if-then-else. The implementation could transform a program using the new if-then construct into a program using only the if-then-else construct, then specialize this program, and transform the residual program back to one using the if-then command where possible.

The effect of this problem is visible in the Table 2.13, the row with program "While opt fails". In this row the execution of the specialization of the self interpreter with the program takes longer than just executing the program. Therefore the pair of the specializer and the self interpreter does not have the JOI property.

Nested whiles problem

The problem with the specialization of the nested whiles is more complicated to solve. Multiple approaches are possible.

The loop detection mechanism works by comparing the static parts of the state. Analysis of the runtime state of the specializer when working on the string maching programs (KMP) have revealed that the loop detection is causing the problems. The specializer is allowed to continue to long. The problem being that when specializing the KMP the static parts of the store never reach the same state.

Two things can solve this problem.

1. Find a better way to detect loops, especially in the presense of nested loops.
2. Change the language to disallow nested loops.

Case 1 will certainly take some time, the problem is not trivial.

Case 2 can be done easily, and it can be shown that every program with nested loops can be transformed into one with only one loop. This is actually noted by Jones in the book "Computability and complexity" in Chapter 8.2 (Jones, 1997, Chapter 8.2).

The identification of the optimizing problem with program transformations is an important insight into what makes and breaks Jones Optimality.

2.10.1 Future work

The language $\mathcal{IF}$ presented here should be extended with the if-then construction as described above. This of cource requires implementation of an interpreter, a self interpreter and a specializer, to show that this new language indeed solves the problem.

The loop detection problem should also be solved either by one of the approaches listed in above.

Appendix A

$\mathcal{I}$ source code

A.1 $\mathcal{I}$ implementation

A.1.1 Value structure

```
structure Value =
struct

open ITools;

exception ValueException of string;

datatype Value = N
       | C of Value * Value
;

fun toString( C(v1, v2) ) =
    "<" ^ toString( v1 ) ^ "." ^ toString( v2 ) ^ ">"
  | toString( N ) = "N"
;

fun toStringPP( C(v1, v2), max, n ) =
    let
        val s1 = toStringPP( v1, max, n + 1 );
        val totalSize = n + 3 +
                        String.size( s1 );
    in
        if( totalSize > !maxWidth )
        then "<" ^ s1 ^ ".\n" ^ indent_string( n + 1 )
             ^ toStringPP( v2, max, n + 1 ) ^ ">"
        else "<" ^ s1 ^ "." ^ toStringPP( v2, max, totalSize ) ^ ">"
    end
  | toStringPP( N, max, n ) = "N"
;
```

```
fun size( N ) = 1
  | size( C( v1, v2 ) ) = 1 + size( v1 ) + size( v2 )
;

fun applyHd( N ) =
    raise ValueException( "cannot take HD of nil" )
  | applyHd( C( v1, v2 ) ) = v1

fun applyTl( N ) =
    raise ValueException( "cannot take TL of nil" )
  | applyTl( C( v1, v2 ) ) = v2

end
;
```

A.1.2 Abstract Value structure

```
structure AbstractValue =
struct

datatype AbstractValue = AD
       | AV of Value.Value
       | ACONS of AbstractValue * AbstractValue
       | ATL of AbstractValue
       | AHD of AbstractValue
       | AEQ of AbstractValue * AbstractValue
;

fun toString( AD ) = "D"
  | toString( AV v ) = Value.toString( v )
  | toString( ACONS( e1, e2 ) ) =
    "(" ^ toString( e1 ) ^ "." ^ toString( e2 ) ^ ")"
  | toString( AHD e ) = "hd( " ^ toString( e ) ^ ")"
  | toString( ATL e ) = "tl( " ^ toString( e ) ^ ")"
  | toString( AEQ( e1, e2 ) ) =
    "=?(" ^ toString( e1 ) ^ "." ^ toString( e2 ) ^ ")"
;

end
;
```

A.1.3 Expression structure

```
structure Expression =
struct
```

```
open Value;
open ITools;

datatype Expression = VAR
    | VA of Value.Value
    | CONS of Expression * Expression
    | HD of Expression
    | TL of Expression
    | EQ of Expression * Expression
;

fun toString( VAR ) = varname
  | toString( VA v ) = Value.toString( v )
  | toString( CONS( e1, e2 ) ) =
    "cons( " ^ toString( e1 ) ^ ", " ^ toString( e2 ) ^ " )"
  | toString( HD e ) = "hd( " ^ toString( e ) ^ " )"
  | toString( TL e ) = "tl( " ^ toString( e ) ^ " )"
  | toString( EQ( e1, e2 ) ) =
    "=?( " ^ toString( e1 ) ^ ", " ^ toString( e2 ) ^ " )"
;

fun toStringPP( VAR, max, n ) = varname
  | toStringPP( VA v, max, n ) = Value.toStringPP( v, max, n )
  | toStringPP( CONS( e1, e2 ), max, n ) =
    let
        val e1s = toString( e1 )
        val e2s = toString( e2 )
        val totalSize = n + 9 +
                        String.size( e1s ) +
                        String.size( e2s );
    in
        if totalSize > max
        then "cons(\n" ^
             indent_string( n + 2 ) ^
             toStringPP( e1, max, n + 2 ) ^ ",\n" ^
             indent_string( n + 2 ) ^
             toStringPP( e2, max, n + 2 ) ^ "\n" ^
             indent_string( n ) ^ " )"
        else "cons( " ^ e1s ^ ", " ^ e2s ^ " )"
    end
  | toStringPP( HD e, max, n ) =
    "hd( " ^ toStringPP( e, max, n + 4 ) ^ " )"
  | toStringPP( TL e, max, n ) =
    "tl( " ^ toStringPP( e, max, n + 4 ) ^ " )"
  | toStringPP( EQ( e1, e2 ), max, n ) =
    let
        val e1s = toString( e1 )
        val e2s = toString( e2 )
        val totalSize = n + 7 +
                        String.size( e1s ) +
                        String.size( e2s );
    in
        if totalSize > max
        then "=?( " ^ e1s ^ ",\n" ^
             indent_string( n + 4 ) ^ e2s ^ " )"
        else "=?( " ^ e1s ^ ", " ^ e2s ^ " )"
    end
;

fun evaluate( VAR, var, time ) = ( var, time + 1 )
  | evaluate( VA v, var, time ) = ( v, time + 1 )
  | evaluate( CONS( e1, e2 ), var, time ) =
    let
val ( v1, t1 ) = evaluate( e1, var, time );
val ( v2, t2 ) = evaluate( e2, var, t1 );
    in
( C( v1, v2 ), t2 + 1 )
    end
  | evaluate( HD e, var, time ) =
    let
val ( v1, t1 ) = evaluate( e, var, time );
    in
( Value.applyHd( v1 ), t1 + 1 )
    end
  | evaluate( TL e, var, time ) =
    let
val ( v1, t1 ) = evaluate( e, var, time );
    in
( Value.applyTl( v1 ), t1 + 1 )
    end
  | evaluate( EQ( e1, e2 ), var , time) =
    let
val ( v1, t1 ) = evaluate( e1, var, time );
val ( v2, t2 ) = evaluate( e2, var, t1 );
    in
if( v1 = v2 )
then ( C( N, N ), t2 + 1 )
else ( N, t2 + 1 )
    end
;

fun compact( VAR ) = VAR
  | compact( VA v ) = VA v
  | compact( HD( e ) ) = compact_hd( compact( e ) )
  | compact( TL( e ) ) = compact_tl( compact( e ) )
  | compact( CONS( e1, e2 ) ) =
    CONS( compact( e1 ), compact( e2 ) )
  | compact( EQ( e1, e2 ) ) =
    let
        val e1c = compact( e1 );
        val e2c = compact( e2 );
    in
        EQ( e1c, e2c )
    end

and compact_hd( VA( C( v1, v2 ) ) ) = VA v1
  | compact_hd( CONS( e1, e2 ) ) = compact( e1 )
  | compact_hd( e ) = HD( compact( e ) )

and compact_tl( VA( C( v1, v2 ) ) ) = VA v2
  | compact_tl( CONS( e1, e2 ) ) = compact( e2 )
  | compact_tl( e ) = TL( compact( e ) )
;

fun substitute( e1, VAR ) = e1
  | substitute( e1, VA v ) = VA v
  | substitute( e1, HD( e2 ) ) = HD( substitute( e1, e2 ) )
  | substitute( e1, TL( e2 ) ) = TL( substitute( e1, e2 ) )
  | substitute( e1, CONS( e2, e3 ) ) =
    CONS( substitute( e1, e2 ), substitute( e1, e3 ) )
  | substitute( e1, EQ( e2, e3 ) ) =
    EQ( substitute( e1, e2 ), substitute( e1, e3 ) )
;

end
;
```

A.1.4 Command structure

```
structure Command =
struct

open Expression;
open ITools;

datatype Command = ASSIGN of Expression.Expression
 | SEQ of Command * Command
 | WHILE of Expression.Expression * Command
;

fun toString( ASSIGN e ) =
    varname ^ " := " ^ Expression.toString( e )
  | toString( SEQ( c1, c2 ) ) =
    toString( c1 ) ^ ";" ^ newline() ^ toString( c2 )
  | toString( WHILE( e, c ) ) =
    "while( " ^ Expression.toString( e )
    ^ " ) do" ^ startBlock() ^ newline()
    ^ toString( c ) ^ endBlock()
;

fun toStringPP( ASSIGN e, max, n ) =
    varname ^ " := " ^ Expression.toStringPP( e, max, !indent + 5 )
  | toStringPP( SEQ( c1, c2 ), max, n ) =
    toStringPP( c1, max, n ) ^ ";" ^ newline() ^ toStringPP( c2, max, n )
  | toStringPP( WHILE( e, c ), max, n ) =
    "while( " ^ Expression.toStringPP( e, max, !indent + 7 )
    ^ " ) do" ^ startBlock() ^ newline()
    ^ toStringPP( c, max, n ) ^ endBlock()

fun timed_interpret( ASSIGN( e ), var, time ) =
    evaluate( e, var, time + 1 )
  | timed_interpret( SEQ( c1, c2 ), var, time ) =
    let
val ( v1, t1 ) = timed_interpret( c1, var, time )
    in
timed_interpret( c2, v1, t1 + 1 )
    end
  | timed_interpret( WHILE( e, c ), var, time ) =
    let
val ( ve, t ) = evaluate( e, var, time );
val ( v1, t1 ) = if( ve = N )
 then ( var, t + 1 )
 else timed_interpret( c, var, t + 1 )
    in
if( ve = N )
then ( v1, t1 )
else timed_interpret( WHILE( e, c ), v1, t1 )
    end
;

fun timed_execution( p, d ) = timed_interpret( p, d, 0 );

fun order_seq( ASSIGN e ) =
    ASSIGN( e )
  | order_seq( SEQ( SEQ( c1, c2 ), c3 ) ) =
    SEQ( c1, order_seq( SEQ( c2, c3 ) ) )
  | order_seq( SEQ( c1, c2 ) ) =
    SEQ( c1, order_seq( c2 ) )
  | order_seq( WHILE( e, c ) ) =
    WHILE( e, order_seq( c ) )
;

fun compress_transition( SEQ( ASSIGN( VAR ), c2 ) ) =
    compress_transition( c2 )
  | compress_transition( SEQ( ASSIGN( e1 ), ASSIGN( e2 ) ) ) =
    ASSIGN( compact( substitute( e1, e2 ) ) )
  | compress_transition( SEQ( ASSIGN( e1 ),
                              SEQ( ASSIGN( e2 ), c3 ) ) ) =
    let
        val e2r = compact( substitute( e1, e2 ) );
    in
        SEQ( ASSIGN( e2r ), compress_transition( c3 ) )
    end
  | compress_transition( SEQ( c1, c2 ) ) =
    SEQ( compress_transition( c1 ), compress_transition( c2 ) )
  | compress_transition( WHILE( e, c ) ) =
    WHILE( e, compress_transition( c ) )
  | compress_transition( p ) = p
;

fun compress_transitions( prog ) =
    let
        val new_prog = compress_transition( prog );
        val result = if prog = new_prog
                     then new_prog
                     else compress_transitions( new_prog )
    in
        result
    end
;

end
;
```

A.1.5 Self interpreter program

```
X := cons( hd( X ), cons( N, cons( N, tl( X ) ) ) );
while( hd( X ) ) do
  X := cons(
         hd( X ),
         cons( cons( cons( =?( hd( hd( hd( X ) ) ), <<N.N>.N> ), <N.N> ), hd( tl( X ) ) ), tl( tl( X ) ) )
        );
  while( hd( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons( cons( tl( hd( hd( X ) ) ), cons( <<<N.<N.N>>.N>.N>, tl( hd( X ) ) ) ), tl( X ) );
  while( tl( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons(
           hd( X ),
           cons(
             cons( cons( =?( hd( hd( hd( X ) ) ), <<N.N>.<N.N>> ), <N.N> ), hd( tl( X ) ) ),
             tl( tl( X ) )
            )
          );
    while( hd( hd( hd( tl( X ) ) ) ) ) do
      X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
      X := cons( cons( hd( tl( hd( hd( X ) ) ) ), cons( tl( tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ) ), tl( X ) );
    while( tl( hd( hd( tl( X ) ) ) ) ) do
      X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
      X := cons(
             hd( X ),
             cons(
               cons( cons( =?( hd( hd( hd( X ) ) ), <<N.N>.<<N.N>.N>> ), <N.N> ), hd( tl( X ) ) ),
               tl( tl( X ) )
              )
            );
```

```
while( hd( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         cons(
           hd( tl( hd( hd( X ) ) ) ),
           cons( cons( <<N.<N.N>>.<N.N>>, N ), cons( hd( hd( X ) ), tl( hd( X ) ) ) )
          ),
         tl( X )
        );
while( tl( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons( cons( cons( =?( hd( hd( hd( X ) ) ), <N.N> ), <N.N> ), hd( tl( X ) ) ), tl( tl( X ) ) )
        );
  while( hd( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons( tl( hd( X ) ), tl( X ) );
    X := cons(
           hd( X ),
           cons(
             hd( tl( X ) ),
             cons( cons( tl( tl( tl( X ) ) ), hd( tl( tl( X ) ) ) ), tl( tl( tl( X ) ) ) )
            )
          );
  while( tl( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons(
           hd( X ),
           cons(
             cons( cons( =?( hd( hd( hd( X ) ) ), <N.<N.N>> ), <N.N> ), hd( tl( X ) ) ),
             tl( tl( X ) )
        )
      );
while( hd( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons(
           hd( tl( X ) ),
           cons( cons( tl( hd( hd( X ) ) ), hd( tl( tl( X ) ) ) ), tl( tl( tl( X ) ) ) )
          )
        );
  X := cons( tl( hd( X ) ), tl( X ) );
while( tl( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons(
           cons( cons( =?( hd( hd( hd( X ) ) ), <N.<<N.N>.N>> ), <N.N> ), hd( tl( X ) ) ),
           tl( tl( X ) )
          )
        );
  while( hd( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons(
           cons(
             hd( tl( hd( hd( X ) ) ) ),
             cons(
               tl( tl( hd( hd( X ) ) ) ),
               cons( cons( <<N.<N.N>>.<<<N.N>.N>.N>>, N ), tl( hd( X ) ) )
              )
            ),
           tl( X )
        );
while( tl( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons(
           cons( cons( =?( hd( hd( hd( X ) ) ), <N.<<<N.N>.N>.N>> ), <N.N> ), hd( tl( X ) ) ),
           tl( tl( X ) )
          )
        );
  while( hd( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons(
           cons( tl( hd( hd( X ) ) ), cons( cons( <<N.<N.N>>.<<N.N>.N>>, N ), tl( hd( X ) ) ) ),
           tl( X )
          );
  while( tl( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons(
           hd( X ),
           cons(
             cons( cons( =?( hd( hd( hd( X ) ) ), <N.<<<<N.N>.N>.N>.N>> ), <N.N> ), hd( tl( X ) ) ),
             tl( tl( X ) )
            )
          );
    while( hd( hd( hd( tl( X ) ) ) ) ) do
      X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
      X := cons(
             cons(
               tl( hd( hd( X ) ) ),
               cons( cons( <<N.<N.N>>.<<<<N.N>.N>.N>.N>>, N ), tl( hd( X ) ) )
              ),
         tl( X )
        );
while( tl( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons(
           cons(
             cons( =?( hd( hd( hd( X ) ) ), <N.<<<<<N.N>.N>.N>.N>.N>> ), <N.N> ),
             hd( tl( X ) )
            ),
           tl( tl( X ) )
          )
        );
  while( hd( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons(
           cons(
             hd( tl( hd( hd( X ) ) ) ),
             cons(
               tl( tl( hd( hd( X ) ) ) ),
               cons( cons( <<N.<N.N>>.<<<<<N.N>.N>.N>.N>.N>>, N ), tl( hd( X ) ) )
              )
            ),
           tl( X )
          );
  while( tl( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons(
           hd( X ),
           cons(
             cons( cons( =?( hd( hd( hd( X ) ) ), <<N.<N.N>>.N> ), <N.N> ), hd( tl( X ) ) ),
         tl( tl( X ) )
        )
      );
while( hd( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons( tl( hd( X ) ), tl( X ) );
  X := cons(
         hd( X ),
         cons( hd( tl( X ) ), cons( hd( tl( tl( X ) ) ), hd( hd( tl( tl( X ) ) ) ) ) )
        );
  X := cons(
         hd( X ),
         cons( hd( tl( X ) ), cons( tl( hd( tl( tl( X ) ) ) ), tl( tl( tl( X ) ) ) ) )
        );
while( tl( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons(
           cons(
             cons( =?( hd( hd( hd( X ) ) ), <<N.<N.N>>.<N.N>> ), <N.N> ),
             hd( tl( X ) )
            ),
           tl( tl( X ) )
          )
        );
  while( hd( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons(
           hd( X ),
           cons(
             cons( cons( hd( hd( tl( tl( X ) ) ) ), <N.N> ), hd( tl( X ) ) ),
           tl( tl( X ) )
          )
        );
  while( hd( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons(
           cons(
             tl( tl( hd( tl( hd( X ) ) ) ) ),
             cons( hd( tl( hd( X ) ) ), tl( tl( hd( X ) ) ) )
            ),
           tl( X )
          );
    X := cons(
           hd( X ),
           cons( hd( tl( X ) ), cons( tl( hd( tl( tl( X ) ) ) ), tl( tl( tl( X ) ) ) ) )
          );
  while( tl( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons( tl( tl( hd( X ) ) ), tl( X ) );
    X := cons(
           hd( X ),
           cons( hd( tl( X ) ), cons( tl( hd( tl( tl( X ) ) ) ), tl( tl( tl( X ) ) ) ) )
          );
  X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
while( tl( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons(
           cons(
             cons( =?( hd( hd( hd( X ) ) ), <<N.<N.N>>.<<N.N>.N>> ), <N.N> ),
             hd( tl( X ) )
          ),
         tl( tl( X ) )
        )
      );
while( hd( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons(
           hd( tl( X ) ),
           cons(
             cons( hd( hd( hd( tl( tl( X ) ) ) ) ), tl( hd( tl( tl( X ) ) ) ) ),
             tl( tl( tl( X ) ) )
            )
          )
        );
  X := cons( tl( hd( X ) ), tl( X ) );
while( tl( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons(
           cons(
             cons( =?( hd( hd( hd( X ) ) ), <<N.<N.N>>.<<<N.N>.N>.N>> ), <N.N> ),
             hd( tl( X ) )
            ),
           tl( tl( X ) )
          )
        );
  while( hd( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons( tl( hd( X ) ), tl( X ) );
  X := cons(
         hd( X ),
         cons(
           hd( tl( X ) ),
           cons(
             cons(
               cons( hd( tl( hd( tl( tl( X ) ) ) ) ), hd( hd( tl( tl( X ) ) ) ) ),
               tl( tl( hd( tl( tl( X ) ) ) ) )
              ),
             tl( tl( tl( X ) ) )
            )
          )
        );
while( tl( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons(
           cons(
             cons( =?( hd( hd( hd( X ) ) ), <<N.<N.N>>.<<<<N.N>.N>.N>.N>> ), <N.N> ),
             hd( tl( X ) )
            ),
           tl( tl( X ) )
          )
        );
  while( hd( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons(
           hd( X ),
           cons(
             hd( tl( X ) ),
             cons(
             cons( tl( hd( hd( tl( tl( X ) ) ) ) ), tl( hd( tl( tl( X ) ) ) ) ),
             tl( tl( tl( X ) ) )
            )
          )
        );
  X := cons( tl( hd( X ) ), tl( X ) );
while( tl( hd( hd( tl( X ) ) ) ) ) do
  X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
  X := cons(
         hd( X ),
         cons(
           cons(
             cons(
               =?( hd( hd( hd( X ) ) ), <<N.<N.N>>.<<<<<N.N>.N>.N>.N>.N>> ),
               <N.N>
              ),
             hd( tl( X ) )
            ),
           tl( tl( X ) )
          )
        );
  while( hd( hd( hd( tl( X ) ) ) ) ) do
    X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
    X := cons( tl( hd( X ) ), tl( X ) );
    X := cons(
           hd( X ),
           cons(
             hd( tl( X ) ),
             cons(
               cons(
                 =?( hd( tl( hd( tl( tl( X ) ) ) ) ), hd( hd( tl( tl( X ) ) ) ) ),
                 tl( tl( hd( tl( tl( X ) ) ) ) )
```

```
                                            ),
                                           tl( tl( tl( X ) ) )
                                          )
                                        )
                                      );
                              while( tl( hd( hd( tl( X ) ) ) ) ) do
                                X := cons( hd( X ), cons( cons( <N.N>, tl( hd( tl( X ) ) ) ), tl( tl( X ) ) ) );
                                X := X;
                              X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
                            X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
                          X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
                        X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
                      X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
                    X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
                  X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
                X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
              X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
            X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
          X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
        X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
      X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
    X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
  X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) );
X := tl( tl( tl( X ) ) )
```

A.1.6 Specializer structure

```
structure Specializer =
struct

open Value;
open AbstractValue;
open Expression;
open Command;

open ITools;

datatype WhileConditionType = SN | SC | D;

fun while_condition( AD ) = D
  | while_condition( AV N ) = SN
  | while_condition( AV( C( v1, v2 ) ) ) = SC
  | while_condition( ACONS( av1, av2 ) ) = SC
  | while_condition( e ) = D
;

fun reduce( ASSIGN e, data ) =
    let
        val er = reduce_exp( e, data );
    in
        ( ASSIGN( e ), er )
    end
  | reduce( SEQ( c1, c2 ), data ) =
    let
        val ( p1, d1 ) = reduce( c1, data );
        val ( p2, d2 ) = reduce( c2, d1 );
    in
        ( SEQ( p1, p2 ), d2 )
    end
  | reduce( WHILE( e, c ), data ) =
    let
        val wc = while_condition( reduce_exp( e, data ) );
    in
        reduce_while( wc, e, c, data )
    end

and reduce_while( SN, e, c, data ) =
    ( ASSIGN( VAR ), data )
  | reduce_while( SC, e, c, data ) =
    reduce( SEQ( c, WHILE( e, c ) ), data )
  | reduce_while( D, e, c, data ) =
    ( WHILE( e, c ), AD )

and reduce_exp( VA v, data ) = AV v
```

```
  | reduce_exp( VAR, data ) = data
  | reduce_exp( CONS( e1, e2 ), data ) =
    let
        val e1r = reduce_exp( e1, data );
        val e2r = reduce_exp( e2, data );
    in
        reduce_cons( e1r, e2r )
    end
  | reduce_exp( HD e, data ) =
    let
        val er = reduce_exp( e, data );
    in
        reduce_hd( er )
    end
  | reduce_exp( TL e, data ) =
    reduce_tl( reduce_exp( e, data ) )
  | reduce_exp( EQ( e1, e2 ), data ) =
    let
        val e1r = reduce_exp( e1, data );
        val e2r = reduce_exp( e2, data );
        val result = reduce_eq( e1r, e2r );
    in
        result
    end

and reduce_cons( AV v1, AV v2 ) = AV( C(v1, v2 ) )
  | reduce_cons( e1, e2 ) = ACONS( e1, e2 )

and reduce_hd( AD ) = AHD( AD )
  | reduce_hd( AV N ) = AV N
  | reduce_hd( AV( C( v1, v2 ) ) ) = AV v1
  | reduce_hd( ACONS( av1, av2 ) ) = av1
  | reduce_hd( AHD( v ) ) = AHD( AHD v )
  | reduce_hd( ATL( v ) ) = AHD( ATL v )
  | reduce_hd( AEQ( v1, v2 ) ) = AHD( AEQ( v1, v2 ) )

and reduce_tl( AD ) = ATL( AD )
  | reduce_tl( AV N ) = AV N
  | reduce_tl( AV( C( v1, v2 ) ) ) = AV v2
  | reduce_tl( ACONS( av1, av2 ) ) = av2
  | reduce_tl( AHD( v ) ) = ATL( AHD v )
  | reduce_tl( ATL( v ) ) = ATL( ATL v )
  | reduce_tl( AEQ( v1, v2 ) ) = ATL( AEQ( v1, v2 ) )

and reduce_eq( AD, AD ) = AV( C(N,N) )
  | reduce_eq( AV v1, AV v2 ) =
    if v1 = v2
    then AV( C(N,N) )
    else AV( N )
  | reduce_eq( ACONS( e1, e2 ), ACONS( f1, f2 ) ) =
    if( reduce_eq( e1, f1 ) = reduce_eq( e2, f2 ) )
    then AV( C(N,N) )
    else AV( N )
  | reduce_eq( ACONS( e1, e2 ), AV( C( f1, f2 ) ) ) =
    if( reduce_eq( e1, AV f1 ) = reduce_eq( e2, AV f2 ) )
    then AV( C(N,N) )
    else AV( N )
  | reduce_eq( AV( C( e1, e2 ) ), ACONS( f1, f2 ) ) =
    if( reduce_eq( AV e1, f1 ) = reduce_eq( AV e2, f2 ) )
    then AV( C(N,N) )
    else AV( N )
  | reduce_eq( AHD( e1 ), AHD( e2 ) ) = reduce_eq( e1, e2 )
  | reduce_eq( ATL( e1 ), ATL( e2 ) ) = reduce_eq( e1, e2 )
  | reduce_eq( v1, v2 ) = AEQ( v1, v2 )
;

fun specializer( program, static ) =
    let
        val basedata = ACONS( AV static, AD );
        val ordered_prog = order_seq( program );
        val annprog = reduce( ordered_prog, basedata );
        val residual = SEQ( ASSIGN( CONS( VA static, VAR ) ),
                            #1( annprog ) );
        val result = compress_transitions( residual );
    in
        result
    end

end
;
```

A.1.7 Tools structure

```
structure ITools =
struct

val varname = "X";

val indent = ref 0;

val maxWidth = ref 60;

fun indent_string( 0 ) = ""
  | indent_string( n ) = " " ^ indent_string( n - 1 )
;
```

```
fun toString_indent() =
    indent_string( !indent )
;

fun newline() = "\n" ^ toString_indent();

fun startBlock() =
    let
val _ = indent := !indent + 2
    in
""
    end
;

fun endBlock() =
    let
val _ = indent := !indent - 2
    in
""
    end
;

fun string_append( s1, s2 ) = s1 ^ s2

end
;
```

A.2 I tests

A.2.1 Assign val

program

```
X := <N.N>
```

data

```
static   N
dynamic  <N.N>
expected <N.N>
```

A.2.2 Sequence

program

```
X := <N.N>;
X := cons( X, N )
```

data

```
static   N
dynamic  <N.N>
expected <<N.N>.N>
```

A.2.3 while-none

program

```
while( N ) do
  X := <N.<N.N>>
```

data

```
static   N
dynamic  N
expected <N.N>
```

A.2.4 while once

program

```
while( hd( X ) ) do
  X := <N.N>
```

data

```
static   <N.N>
dynamic  <N.N>
expected <N.N>
```

A.2.5 While many

program

```
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), tl( X ) )
```

data

```
static   <<N.N>.N>
dynamic  <N.N>
expected <N.<N.N>>
```

A.2.6 VAR

program

```
X := X
```

data

```
static   N
dynamic  N
expected <N.N>
```

A.2.7 CONS

program

```
X := cons( X, X )
```

data

```
static   N
dynamic  N
expected <<N.N>.<N.N>>
```

A.2.8 HD value

program

```
X := hd( <<N.N>.N> )
```

data

```
static   <N.N>
dynamic  N
expected <N.N>
```

A.2.9 TL VAR

program

```
X := tl( X )
```

data

```
static   N
dynamic  <N.N>
expected <N.N>
```

A.2.10 EQ true

program

```
X := =?( X, <N.N> )
```

data

```
static   N
dynamic  N
expected <N.N>
```

A.2.11 EQ false

program

```
X := =?( X, N )
```

data

```
static   N
dynamic  N
expected N
```

A.2.12 Rev List

program

```
X := cons( N, X );
while( tl( X ) ) do
  X := cons( cons( hd( tl( X ) ), hd( X ) ), tl( tl( X ) ) );
X := hd( X )
```

data

```
static   <N.N>
dynamic  <N.N>
expected <N.<<N.N>.N>>
```

A.2.13 AppendSD

program

```
X := cons( N, X );
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
```

```
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

data

```
static   <<N.N>.<<N.<N.N>>.<<N.<N.<N.N>>>.N>>>
dynamic  <<<N.N>.N>.<<<<N.N>.N>.N>.N>>
expected <<N.N>.<<N.<N.N>>.<<N.<N.<N.N>>>.<<<N.N>.N>.
                                              <<<<N.N>.
                                                 N>.
                                                N>.
                                               N>>>>>
```

A.2.14 AppendDS

program

```
X := cons( N, cons( tl( X ), hd( X ) ) );
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

data

```
static   <<<N.N>.N>.<<<<N.N>.N>.N>.N>>
dynamic  <<N.N>.<<N.<N.N>>.<<N.<N.<N.N>>>.N>>>
expected <<N.N>.<<N.<N.N>>.<<N.<N.<N.N>>>.<<<N.N>.N>.
                                              <<<<N.N>.
                                                 N>.
                                                N>.
                                               N>>>>>
```

A.2.15 DeadCode

program

```
while( tl( X ) ) do
  X := cons( hd( X ), tl( tl( X ) ) );
X := <N.N>
```

data

```
static   <N.<N.N>>
dynamic  <N.<N.N>>
expected <N.N>
```

Appendix B

$\mathcal{IF}$ source code

B.1 $\mathcal{IF}$ implementation

B.1.1 Value structure

```
structure Value =
struct

open Tools;

exception ValueException of string;

datatype Value = N
       | C of Value * Value
;

fun toString( C( v1, v2 ) ) =
    "<" ^ toString( v1 ) ^
    "." ^ toString( v2 ) ^ ">"
  | toString( N ) = "N"
;

fun toStringPP( C(v1, v2), max, n ) =
    let
        val s1 = toString( v1 );
        val s2 = toString( v2 );
        val totalSize = n + 3 +
                        String.size( s1 ) +
                        String.size( s2 );
    in
        if( totalSize > max )
        then "<" ^ s1 ^ ".\n" ^ indent_string( n + 1 )
             ^ toStringPP( v2, max, n + 1 ) ^ ">"
        else "<" ^ s1 ^ "." ^ s2 ^ ">"
    end
  | toStringPP( N, max, n ) = "N"
;

fun size( N ) = 1
  | size( C( v1, v2 ) ) = 1 + size( v1 ) + size( v2 )
;

fun applyHd( N ) =
    raise ValueException( "cannot take HD of nil" )
  | applyHd( C( v1, v2 ) ) = v1

fun applyTl( N ) =
    raise ValueException( "cannot take TL of nil" )
  | applyTl( C( v1, v2 ) ) = v2

end
;
```

B.1.2 AbstractValue structure

```
structure AbstractValue =
struct

open Value;
open Expression;

datatype AbstractValue = AD
       | AV of Value
       | ACONS of AbstractValue * AbstractValue
       | ATL of AbstractValue
       | AHD of AbstractValue
;

fun toString( AD ) = "D"
  | toString( AV v ) = Value.toString( v )
  | toString( ACONS( e1, e2 ) ) =
    "(" ^ toString( e1 ) ^ "." ^ toString( e2 ) ^ ")"
  | toString( AHD e ) = "hd( " ^ toString( e ) ^ ")"
  | toString( ATL e ) = "tl( " ^ toString( e ) ^ ")"
;

fun toExpression( AD ) =
    VAR
  | toExpression( AV v ) =
    VA v
  | toExpression( AHD( av ) ) =
    HD( toExpression( av ) )
  | toExpression( ATL( av ) ) =
    TL( toExpression( av ) )
  | toExpression( ACONS( av1, av2 ) ) =
    CONS( toExpression( av1 ), toExpression( av2 ) )
;

fun applyHd( AD ) = AHD( AD )
  | applyHd( AV N ) = AV N
  | applyHd( AV( C(v1, v2) ) ) = AV v1
  | applyHd( ACONS( av1, av2 ) ) = av1
  | applyHd( av ) = AHD( av )
;

fun applyTl( AD ) = ATL( AD )
  | applyTl( AV N ) = AV N
  | applyTl( AV( C(v1, v2) ) ) = AV v2
  | applyTl( ACONS( av1, av2 ) ) = av2
  | applyTl( av ) = ATL( av )
;

fun reduceCons( AV v1, AV v2 ) = AV( C(v1,v2 ) )
  | reduceCons( AD, AD ) = AD
  | reduceCons( av1, av2 ) = ACONS( av1, av2 )
;

datatype Choise = DD | DC | SN | SC;

fun toString_Choise( DD ) = "DD"
  | toString_Choise( DC ) = "DC"
  | toString_Choise( SC ) = "SC"
  | toString_Choise( SN ) = "SN"
;

fun choose( AD ) = DD
  | choose( AV N ) = SN
  | choose( AV( C( v1, v2 ) ) ) = SC
  | choose( ACONS( av1, av2 ) ) = DC
  | choose( AHD( av ) ) = DD
  | choose( ATL( av ) ) = DD
;

end
;
```

B.1.3 Expression structure

```
structure Expression =
struct
```

```
open Value;
open Partition;
open Tools;

datatype Expression = VAR
    | VA of Value
    | CONS of Expression * Expression
    | HD of Expression
    | TL of Expression
;

fun toString( VAR ) = "X"
  | toString( VA v ) = Value.toString( v )
  | toString( CONS( e1, e2 ) ) =
    "cons( " ^ toString( e1 ) ^ ", " ^
    toString( e2 ) ^ " )"
  | toString( HD( e ) ) =
    "hd( " ^ toString( e ) ^ " )"
  | toString( TL( e ) ) =
    "tl( " ^ toString( e ) ^ " )"
;

fun toStringPP( VAR, max, n ) = "X"
  | toStringPP( VA v, max, n ) = Value.toStringPP( v, max, n )
  | toStringPP( CONS( e1, e2 ), max, n ) =
    let
        val e1s = toString( e1 );
        val e2s = toString( e2 );
        val firstSize = n + 7 + String.size( e1s );
        val totalSize = n + 9 +
                        String.size( e1s ) +
                        String.size( e2s );
    in
        if totalSize < max
        then "cons( " ^ e1s ^ ", " ^ e2s ^ " )"
(*        else if firstSize < max
        then "cons( " ^ e1s ^ ",\n" ^
             indent_string( n + 6 ) ^
             toStringPP( e2, max, n + 6 ) ^ " )"
*)
        else "cons(\n" ^
             indent_string( n + 2 ) ^
             toStringPP( e1, max, n + 2 ) ^ ",\n" ^
             indent_string( n + 2 ) ^
             toStringPP( e2, max, n + 2 ) ^ "\n" ^
             indent_string( n ) ^ " )"
    end
  | toStringPP( HD( e ), max, n ) =
    "hd( " ^ toStringPP( e, max, n + 4 ) ^ " )"
  | toStringPP( TL( e ), max, n ) =
    "tl( " ^ toStringPP( e, max, n + 4 ) ^ " )"
;

fun timed_evaluate( VAR, var, time ) = ( var, time + 1 )
  | timed_evaluate( VA v, var, time ) = ( v, time + 1 )
  | timed_evaluate( CONS( e1, e2 ), var, time ) =
    let
val( v1, t1 ) = timed_evaluate( e1, var, 0 );
val( v2, t2 ) = timed_evaluate( e2, var, 0 );
    in
( C( v1, v2 ), time + t1 + t2 + 1 )
    end
  | timed_evaluate( HD( e ), var, time ) =
    let
val( v1, t1 ) = timed_evaluate( e, var, 0 );
    in
( Value.applyHd( v1 ), time + t1 + 1 )
    end
  | timed_evaluate( TL( e ), var, time ) =
    let
val( v1, t1 ) = timed_evaluate( e, var, 0 );
    in
( Value.applyTl( v1 ), time + t1 + 1 )
    end
;

fun compact( CONS( VA v1, VA v2 ) ) =
    VA( C( v1, v2 ) )
  | compact( CONS( e1, e2 ) ) =
    CONS( compact( e1 ), compact( e2 ) )
  | compact( HD( e ) ) =
    compact_hd( compact( e ) )
  | compact( TL( e ) ) =
    compact_tl( compact( e ) )
  | compact( e ) = e

and compact_tl( VA( C( v1, v2 ) ) ) = VA v2
  | compact_tl( CONS( e1, e2 ) ) = compact( e2 )
  | compact_tl( e ) = TL( e )

and compact_hd( VA( C( v1, v2 ) ) ) = VA v1
  | compact_hd( CONS( e1, e2 ) ) = compact( e1 )
  | compact_hd( e ) = HD( e )
;

fun substitute( e, VAR ) = e
  | substitute( e, VA v ) = VA v
  | substitute( e, CONS( e1, e2 ) ) =
    CONS( substitute( e, e1 ), substitute( e, e2 ) )
  | substitute( e, HD( e1 ) ) =
    HD( substitute( e, e1 ) )
  | substitute( e, TL( e1 ) ) =
    TL( substitute( e, e1 ) )
;

fun reduce_cons_partition( S, S ) =
    S
  | reduce_cons_partition( D, D ) = D
  | reduce_cons_partition( p1, p2 ) = P( p1, p2 )
;

fun reduce_partition( VAR, part ) = part
  | reduce_partition( VA v, part ) = S
  | reduce_partition( HD( p ), part ) =
    Partition.applyHd( reduce_partition( p, part ) )
  | reduce_partition( TL( p ), part ) =
    Partition.applyTl( reduce_partition( p, part ) )
  | reduce_partition( CONS( p1, p2 ), part ) =
    let
val p1r = reduce_partition( p1, part );
val p2r = reduce_partition( p2, part );
    in
        reduce_cons_partition( p1r, p2r )
    end
;

fun isValue( VA v ) = true
  | isValue( e ) = false
;

fun truthValue( VA N ) = false
  | truthValue( VA v ) = true
  | truthValue( e ) = false
;

end
;
```

B.1.4 Command structure

```
structure Command =
struct

exception CommandMaxLoopCount;
```

```
open Expression;
open Tools;

val loopCount = ref 0;
val maxLoopCount = 1000000;

datatype Command = ASSIGN of Expression
 | SEQ of Command * Command
 | WHILE of Expression * Command
 | IF of Expression * Command * Command
;

fun toString( ASSIGN e ) =
    varname ^ " := " ^
    Expression.toString( e )
  | toString( SEQ( c1, c2 ) ) =
    toString( c1 ) ^ ";" ^ newline() ^
    toString( c2 )
  | toString( WHILE( e, c ) ) =
    "while( " ^ Expression.toString( e ) ^
    " ) do" ^ startBlock() ^ newline() ^
    toString( c ) ^ endBlock()
  | toString( IF( e, c1, c2 ) ) =
    "if( " ^ Expression.toString( e ) ^
    " ) then " ^ startBlock() ^ newline() ^
    toString( c1 ) ^ endBlock() ^ newline() ^
    "else" ^ startBlock() ^ newline() ^
    toString( c2 ) ^ endBlock() ^ newline()
;

fun toStringPP( ASSIGN e, max, n ) =
    varname ^ " := " ^
    Expression.toStringPP( e, max, !indent + 5 )
  | toStringPP( SEQ( c1, c2 ), max, n ) =
    toStringPP( c1, max, n ) ^ ";" ^ newline() ^
    toStringPP( c2, max, n )
  | toStringPP( WHILE( e, c ), max, n ) =
    "while( " ^ Expression.toStringPP( e, max, !indent + 7 ) ^
    " ) do" ^ startBlock() ^ newline() ^
    toStringPP( c, max, n ) ^ endBlock()
  | toStringPP( IF( e, c1, c2 ), max, n ) =
    "if( " ^ Expression.toStringPP( e, max, !indent + 4 ) ^
    " ) then " ^ startBlock() ^ newline() ^
    toStringPP( c1, max, n ) ^ endBlock() ^ newline() ^
    "else" ^ startBlock() ^ newline() ^
    toStringPP( c2, max, n ) ^ endBlock() ^ newline()
;

fun timed_interpret( ASSIGN( e ), var, time ) =
    Expression.timed_evaluate( e, var, time + 1 )
  | timed_interpret( SEQ( c1, c2 ), var, time ) =
    let
val (v, t) = timed_interpret( c1, var, 0 );
    in
timed_interpret( c2, v, time + t + 1 )
    end
  | timed_interpret( WHILE( e, c ), var, time ) =
    let
        val _ = if(  !loopCount > maxLoopCount )
                then raise CommandMaxLoopCount
                else ();
        val _ = loopCount := !loopCount + 1;
val (ve,te) = Expression.timed_evaluate( e, var, 0 );
val (cv, ct) = if( ve = N )
       then ( var, 0 )
       else timed_interpret( c, var, 0 );
        val new_time = time + te + ct + 1;

        val _ = loopCount := !loopCount - 1;
    in
        if( ve = N )
        then ( var, new_time )
        else timed_interpret( WHILE( e, c ), cv, new_time )
    end
  | timed_interpret( IF( e, c1, c2 ), var, time ) =
    let
val (ve,te) = Expression.timed_evaluate( e, var, 0 );
    in
if( ve = N )
then timed_interpret( c2, var, time + te + 1 )
else timed_interpret( c1, var, time + te + 1 )
    end

and timed_int_prog( p, d ) =
    let
val ( v, t ) = timed_interpret( p, d, 0 );
    in
( v, t )
    end

and int_prog( p, d ) =
    let
val( v, t ) = timed_interpret( p, d, 0 )
    in
v
    end
;
```

```
fun order_seq( ASSIGN e ) =
    ASSIGN( e )
  | order_seq( SEQ( SEQ( c1, c2 ), c3 ) ) =
    order_seq( SEQ( c1, order_seq( SEQ( c2, c3 ) ) ) )
  | order_seq( SEQ( c1, c2 ) ) =
    SEQ( order_seq( c1 ), order_seq( c2 ) )
  | order_seq( WHILE( e, c ) ) =
    WHILE( e, order_seq( c ) )
  | order_seq( IF( e, c1, c2 ) ) =
    IF( e, order_seq( c1 ), order_seq( c2 ) )
;

fun reverse_order_seq( ASSIGN e ) =
    ASSIGN( e )
  | reverse_order_seq( SEQ( c1, SEQ( c2, c3 ) ) ) =
    reverse_order_seq( SEQ( reverse_order_seq( SEQ( c1, c2 ) ),
                            c3 ) )
  | reverse_order_seq( SEQ( c1, c2 ) ) =
    SEQ( reverse_order_seq( c1 ),
         reverse_order_seq( c2 ) )
  | reverse_order_seq( WHILE( e, c ) ) =
    WHILE( e, reverse_order_seq( c ) )
  | reverse_order_seq( IF( e, c1, c2 ) ) =
    IF( e,
        reverse_order_seq( c1 ),
        reverse_order_seq( c2 ) )
;

fun removeSkipLhs( ASSIGN( e ) ) = ASSIGN e
  | removeSkipLhs( SEQ( ASSIGN( VAR ), c2 ) ) =
    removeSkipLhs( c2 )
  | removeSkipLhs( SEQ( c1, c2 ) ) =
    SEQ( removeSkipLhs( c1 ), removeSkipLhs( c2 ) )
  | removeSkipLhs( WHILE( e, c ) ) =
    WHILE( e, removeSkipLhs( c ) )
  | removeSkipLhs( IF( e, c1, c2 ) ) =
    IF( e, removeSkipLhs( c1 ), removeSkipLhs( c2 ) )
;

fun removeSkipRhs( ASSIGN( e ) ) = ASSIGN e
  | removeSkipRhs( SEQ( c1, ASSIGN( VAR ) ) ) =
    removeSkipRhs( c1 )
  | removeSkipRhs( SEQ( c1, c2 ) ) =
    SEQ( removeSkipRhs( c1 ), removeSkipRhs( c2 ) )
  | removeSkipRhs( WHILE( e, c ) ) =
    WHILE( e, removeSkipRhs( c ) )
  | removeSkipRhs( IF( e, c1, c2 ) ) =
    IF( e, removeSkipRhs( c1 ), removeSkipRhs( c2 ) )
;
```

```
fun joinAssigns( ASSIGN( e ) ) = ASSIGN( e )
  | joinAssigns( SEQ( ASSIGN( e1 ), ASSIGN( e2 ) ) ) =
    ASSIGN( compact( substitute( e1, e2 ) ) )
  | joinAssigns( SEQ( ASSIGN( e1 ),
                      SEQ( ASSIGN( e2 ), c3 ) ) ) =
    SEQ( ASSIGN( compact( substitute( e1, e2 ) ) ), c3 )
  | joinAssigns( SEQ( c1, c2 ) ) =
    SEQ( joinAssigns( c1 ), joinAssigns( c2 ) )
  | joinAssigns( WHILE( e, c ) ) =
    WHILE( e, joinAssigns( c ) )
  | joinAssigns( IF( e, c1, c2 ) ) =
    IF( e, joinAssigns( c1 ), joinAssigns( c2 ) )
;

fun pushAssignIntoIf( ASSIGN( e ) ) = ASSIGN( e )
  | pushAssignIntoIf( SEQ( ASSIGN( e1 ), IF( e2, c1, c2 ) ) ) =
    IF( compact( substitute( e1, e2 ) ),
        SEQ( ASSIGN( e1 ), c1 ),
        SEQ( ASSIGN( e1 ), c2 ) )
  | pushAssignIntoIf( SEQ( c1, c2 ) ) =
    SEQ( pushAssignIntoIf( c1 ), pushAssignIntoIf( c2 ) )
  | pushAssignIntoIf( WHILE( e, c ) ) =
    WHILE( e, pushAssignIntoIf( c ) )
  | pushAssignIntoIf( IF( e, c1, c2 ) ) =
    IF( e, pushAssignIntoIf( c1 ), pushAssignIntoIf( c2 ) )
;

fun interpretRest( ASSIGN( e ) ) = ASSIGN( e )
  | interpretRest( SEQ( ASSIGN( VA v ), c2 ) ) =
    let
        val rest = SEQ( ASSIGN( VA v ), c2 );
        val (res, t ) = timed_interpret( rest, N, 0 );
    in
        ASSIGN( VA res )
    end
  | interpretRest( SEQ( c1, c2 ) ) =
    SEQ( interpretRest( c1 ), interpretRest( c2 ) )
  | interpretRest( WHILE( e, c ) ) =
    WHILE( e, interpretRest( c ) )
  | interpretRest( IF( e, c1, c2 ) ) =
    IF( e, interpretRest( c1 ), interpretRest( c2 ) )
;

fun ifValue( ASSIGN( e ) ) = ASSIGN( e )
  | ifValue( SEQ( c1, c2 ) ) =
    SEQ( ifValue( c1 ), ifValue( c2 ) )
  | ifValue( WHILE( e, c ) ) =
    WHILE( e, ifValue( c ) )
  | ifValue( IF( VA v, c1, c2 ) ) =
    if( v = N )
    then ifValue( c2 )
    else ifValue( c1 )
  | ifValue( IF( e, c1, c2 ) ) =
    IF( e, ifValue( c1 ), ifValue( c2 ) )
;

fun ifCons( ASSIGN( e ) ) = ASSIGN( e )
  | ifCons( SEQ( c1, c2 ) ) =
    SEQ( ifCons( c1 ), ifCons( c2 ) )
  | ifCons( WHILE( e, c ) ) =
    WHILE( e, ifCons( c ) )
  | ifCons( IF( CONS( e1, e2 ), c1, c2 ) ) =
    ifCons( c1 )
  | ifCons( IF( e, c1, c2 ) ) =
    IF( e, ifCons( c1 ), ifCons( c2 ) )

fun removeDeadLhs( ASSIGN( e ) ) = ASSIGN( e )
  | removeDeadLhs( SEQ( c1, ASSIGN( VA v ) ) ) =
    ASSIGN( VA v )
  | removeDeadLhs( SEQ( c1, c2 ) ) =
    SEQ( removeDeadLhs( c1 ), removeDeadLhs( c2 ) )
  | removeDeadLhs( WHILE( e, c ) ) =
    WHILE( e, removeDeadLhs( c ) )
  | removeDeadLhs( IF( e, c1, c2 ) ) =
    IF( e, removeDeadLhs( c1 ), removeDeadLhs( c2 ) )
;

fun ifElseSkip( ASSIGN( e ) ) = ASSIGN( e )
  | ifElseSkip( SEQ( c1, c2 ) ) =
    SEQ( ifElseSkip( c1 ), ifElseSkip( c2 ) )
  | ifElseSkip( WHILE( e, c ) ) =
    WHILE( e, ifElseSkip( c ) )
  | ifElseSkip( IF( e1, ASSIGN( e2 ), ASSIGN( VAR ) ) ) =
    let
        val tmp = compact( substitute( e2, e1 ) );
    in
        if( tmp = VA N )
        then WHILE( e1, ASSIGN( e2 ) )
        else IF( e1, ASSIGN( e2 ), ASSIGN( VAR ) )
    end
  | ifElseSkip( IF( e, c1, c2 ) ) =
    IF( e, ifElseSkip( c1 ), ifElseSkip( c2 ) )
;

fun ifWhileToWhileIf( ASSIGN( e ) ) = ASSIGN( e )
  | ifWhileToWhileIf( SEQ( c1, c2 ) ) =
    SEQ( ifWhileToWhileIf( c1 ), ifWhileToWhileIf( c2 ) )
  | ifWhileToWhileIf( WHILE( e, c ) ) =
    WHILE( e, ifWhileToWhileIf( c ) )
  | ifWhileToWhileIf( IF( e1, SEQ( WHILE( e2, c ),
                                   ASSIGN( e3 ) ) ,
                          ASSIGN( VAR ) ) ) =
    let
        val tmp = compact( substitute( e3, e1 ) );
    in
        if( tmp = VA N )
        then WHILE( e1, IF( e2, c, ASSIGN( e2 ) ) )
        else IF( e1, SEQ( WHILE( e2, c ),
                          ASSIGN( e3 ) ), ASSIGN( VAR ) )
    end
  | ifWhileToWhileIf( IF( e, c1, c2 ) ) =
    IF( e, ifWhileToWhileIf( c1 ), ifWhileToWhileIf( c2 ) )
;

fun forward_optimize( ASSIGN( e ) ) = ASSIGN( compact e )
  | forward_optimize( SEQ( ASSIGN( VAR ), c1 ) ) = c1
  | forward_optimize( SEQ( ASSIGN( VA v ), c2 ) ) =
    let
val prog = SEQ( ASSIGN( VA v ), c2 );
val (v1, t1 ) = timed_interpret( prog, N, 0 );
    in
ASSIGN( VA v1 )
    end

  | forward_optimize( SEQ( ASSIGN e1, ASSIGN e2 ) ) =
    ASSIGN( compact( substitute( e1, e2 ) ) )
  | forward_optimize( SEQ( ASSIGN e1,
                           SEQ( ASSIGN e2, c2 ) ) ) =
    let
val tmp = substitute( e1, e2 );
val assign_exp = compact( tmp );
        val new_code = SEQ( ASSIGN( assign_exp ), c2 );
    in
forward_optimize( order_seq( new_code ) )
    end
  | forward_optimize( SEQ( ASSIGN e1, IF( e2, c1, c2 ) ) ) =
    forward_optimize( IF( compact( substitute( e1, e2 ) ),
                          SEQ( ASSIGN( e1 ), c1 ),
                          SEQ( ASSIGN e1, c2 ) ) )
  | forward_optimize( SEQ( c1, ASSIGN( VAR ) ) ) = c1
  | forward_optimize( SEQ( IF( e, c1, c2 ), c3 ) ) =
    forward_optimize( IF( e, order_seq( SEQ( c1, c3 ) ),
                          order_seq( SEQ( c2, c3 ) ) ) )
  | forward_optimize( SEQ( c1, c2 ) ) =
    SEQ( forward_optimize( c1 ), forward_optimize( c2 ) )
  | forward_optimize( WHILE( VA N, c ) ) =
    ASSIGN( VAR )
  | forward_optimize( WHILE( e, c ) ) =
    WHILE( e, forward_optimize( c ) )

  | forward_optimize( IF( VA N, c1, c2 ) ) =
    forward_optimize( c2 )
  | forward_optimize( IF( VA v, c1, c2 ) ) =
    forward_optimize( c1 )
  | forward_optimize( IF( CONS( e1, e2 ), c1, c2 ) ) =
    forward_optimize( c1 )
  | forward_optimize( IF( e, c1, c2 ) ) =
    IF( e, forward_optimize( c1 ), forward_optimize( c2 ) )
;

fun backward_optimize( ASSIGN( e ) ) =
    ASSIGN( e )
  | backward_optimize( SEQ( c1, c2 ) ) =
    SEQ( backward_optimize( c1 ), backward_optimize( c2 ) )
  | backward_optimize( WHILE( e1,
                              IF( e2,
                                  c1,
                                  SEQ( c2,
                                       WHILE( e3, c3 )
                                     ) ) ) ) =
    let
        val whiles_are_same = e1 = e3;
    in
        if( whiles_are_same )
        then IF( e1,
                 SEQ( WHILE( e2, c1 ),
                      SEQ( c2, WHILE( e3, c3 ) ) ),
                 ASSIGN( VAR ) )
        else WHILE( e1, IF( e2,
                            c1,
                            SEQ( c2, WHILE( e3, c3 ) ) ) )
    end
  | backward_optimize( WHILE( e1,
                              IF( e2, c, ASSIGN( e4 ) ) ) ) =
    let
        val e1e4 = compact( substitute( e4, e1 ) );
    in
        if e1e4 = VA N
        then IF( e1, SEQ( WHILE( e2, c ),
                          ASSIGN( e4 ) ), ASSIGN( VAR ) )
        else WHILE( e1, IF( e2, c, ASSIGN( e4 ) ) )
    end
  | backward_optimize( WHILE( e1, ASSIGN( e2 ) ) ) =
    let
```

```
        val e1e2 = compact( substitute( e2, e1 ) );
        val isNil = e1e2 = VA N;
    in
        if( isNil )
        then IF( e1, ASSIGN( e2 ), ASSIGN( VAR ) )
        else WHILE( e1, ASSIGN( e2 ) )
    end
  | backward_optimize( WHILE( e, c ) ) =
    WHILE( e, backward_optimize( c ) )
  | backward_optimize( IF( e, c1, c2 ) ) =
    IF( e, backward_optimize( c1 ), backward_optimize( c2 ) )
;

fun hdUsedExp( HD( VAR ) ) = true
  | hdUsedExp( HD( e ) ) = hdUsedExp( e )
  | hdUsedExp( TL( e ) ) = hdUsedExp( e )
  | hdUsedExp( CONS( e1, e2 ) ) =
    hdUsedExp( e1 ) orelse hdUsedExp( e2 )
  | hdUsedExp( VAR ) = false
  | hdUsedExp( VA v ) = false
;

fun hdUsed1( ASSIGN( CONS( e1, e2 ) ) ) =
    hdUsedExp( e2 )
  | hdUsed1( ASSIGN( e ) ) =
    hdUsedExp( e )
  | hdUsed1( SEQ( c1, c2 ) ) =
    hdUsed1( c1 ) orelse hdUsed1( c2 )
  | hdUsed1( WHILE( e, c ) ) =
    hdUsedExp( e ) orelse hdUsed1( c )
  | hdUsed1( IF( e, c1, c2 ) ) =
    hdUsedExp( e ) orelse hdUsed1( c1 ) orelse hdUsed1( c2 )
;

fun hdUsed( SEQ( ASSIGN( e ), c2 ) ) = hdUsed1( c2 )
  | hdUsed( p ) = hdUsed1( p )
;

fun tlUsedExp( TL( VAR ) ) = true
  | tlUsedExp( HD( e ) ) = tlUsedExp( e )
  | tlUsedExp( TL( e ) ) = tlUsedExp( e )
  | tlUsedExp( CONS( e1, e2 ) ) =
    tlUsedExp( e1 ) orelse tlUsedExp( e2 )
  | tlUsedExp( VAR ) = false
  | tlUsedExp( VA v ) = false
;

fun tlUsed1( ASSIGN( CONS( e1, e2 ) ) ) =
    tlUsedExp( e1 )
  | tlUsed1( ASSIGN( e ) ) =
    tlUsedExp( e )
  | tlUsed1( SEQ( c1, c2 ) ) =
    tlUsed1( c1 ) orelse tlUsed1( c2 )
  | tlUsed1( WHILE( e, c ) ) =
    tlUsedExp( e ) orelse tlUsed1( c )
  | tlUsed1( IF( e, c1, c2 ) ) =
    tlUsedExp( e ) orelse tlUsed1( c1 ) orelse tlUsed1( c2 )
;

fun tlUsed( SEQ( ASSIGN( e ), c2 ) ) = tlUsed1( c2 )
  | tlUsed( p ) = tlUsed1( p )
;

fun removeHdExp( VAR ) = VAR
  | removeHdExp( VA v ) = VA v
  | removeHdExp( TL VAR ) = VAR
  | removeHdExp( TL( e ) ) = TL( removeHdExp( e ) )
  | removeHdExp( HD VAR ) = VAR
  | removeHdExp( HD( e ) ) = HD( removeHdExp( e ) )
  | removeHdExp( CONS( e1, e2 ) ) =
    CONS( removeHdExp( e1 ), removeHdExp( e2 ) )
;

fun removeHd1( ASSIGN( CONS( e1, e2 ) ) ) =
    ASSIGN( removeHdExp( e2 ) )
  | removeHd1( ASSIGN( VA( C( v1, v2 ) ) ) ) =
    ASSIGN( VA v1 )
  | removeHd1( ASSIGN( e ) ) =
    ASSIGN( removeHdExp( e ) )
  | removeHd1( SEQ( c1, c2 ) ) =
    SEQ( removeHd1( c1 ), removeHd1( c2 ) )
  | removeHd1( WHILE( e, c ) ) =
    WHILE( removeHdExp( e ), removeHd1( c ) )
  | removeHd1( IF( e, c1, c2 ) ) =
    IF( removeHdExp( e ), removeHd1( c1 ), removeHd1( c2 ) )
;

fun removeHd( SEQ( ASSIGN( CONS( e1, e2 ) ), c1 ) ) =
    SEQ( ASSIGN( e2 ), removeHd1( c1 ) )
  | removeHd( p ) = removeHd1( p )
;

fun removeTlExp( VAR ) = VAR
  | removeTlExp( VA v ) = VA v
  | removeTlExp( TL VAR ) = VAR
  | removeTlExp( TL( e ) ) = TL( removeTlExp( e ) )
  | removeTlExp( HD VAR ) = VAR
  | removeTlExp( HD( e ) ) = HD( removeTlExp( e ) )
  | removeTlExp( CONS( e1, e2 ) ) =
    CONS( removeTlExp( e1 ), removeTlExp( e2 ) )
;

fun removeTl1( ASSIGN( CONS( e1, e2 ) ) ) =
    ASSIGN( removeTlExp( e1 ) )
  | removeTl1( ASSIGN( VA( C( v1, v2 ) ) ) ) =
    ASSIGN( VA v1 )
  | removeTl1( ASSIGN( e ) ) =
    ASSIGN( removeTlExp( e ) )
  | removeTl1( SEQ( c1, c2 ) ) =
    SEQ( removeTl1( c1 ), removeTl1( c2 ) )
  | removeTl1( WHILE( e, c ) ) =
    WHILE( removeTlExp( e ), removeTl1( c ) )
  | removeTl1( IF( e, c1, c2 ) ) =
    IF( removeTlExp( e ), removeTl1( c1 ), removeTl1( c2 ) )
;

fun removeTl( SEQ( ASSIGN( CONS( e1, e2 ) ), c1 ) ) =
    SEQ( ASSIGN( e1 ), removeTl1( c1 ) )
  | removeTl( p ) = removeTl1( p )
;

fun lowerArity( ASSIGN( e ) ) =
    ASSIGN( e )
  | lowerArity( SEQ( c1, ASSIGN( VA v ) ) ) =
    ASSIGN( VA v )
  | lowerArity( SEQ( c1, ASSIGN( CONS( e1, e2 ) ) ) ) =
    SEQ( c1, ASSIGN( CONS( e1, e2 ) ) )
  | lowerArity( SEQ( c1, ASSIGN( VAR ) ) ) =
    c1
  | lowerArity( SEQ( c1, ASSIGN( e ) ) ) =
    let
        val hu = hdUsed( order_seq( SEQ( c1, ASSIGN( e ) ) ) );
        val tu = tlUsed( order_seq( SEQ( c1, ASSIGN( e ) ) ) );
    in
        if( not hu )
        then removeHd( order_seq( SEQ( c1, ASSIGN( e ) ) ) )
        else if( not tu )
        then removeTl( order_seq( SEQ( c1, ASSIGN( e ) ) ) )
        else SEQ( c1, ASSIGN( e ) )
    end
  | lowerArity( SEQ( c1, c2 ) ) = SEQ( c1, c2 )
  | lowerArity( WHILE( e, c ) ) = WHILE( e, c )
  | lowerArity( IF( e, c1, c2 ) ) = IF( e, c1, c2 )
;

fun compressTransitions( prog ) =
    let
```

```
        val prog1 = forward_optimize( order_seq( prog ) );
        val prog1r = reverse_order_seq( prog1 )
        val prog2 = backward_optimize( prog1r );
        val prog2r = reverse_order_seq( prog2 );
        val prog3 = lowerArity( prog2r );

        val new_prog = prog3;
    in
        if( new_prog = prog )
        then new_prog
        else compressTransitions( new_prog )
    end
;

fun optimize( prog ) =
    let
        val prog1 = removeSkipLhs( order_seq( prog ) );
        val prog2 = removeSkipRhs( order_seq( prog1 ) );
        val prog3 = joinAssigns( order_seq( prog2 ) );
        val prog4 = pushAssignIntoIf( order_seq( prog3 ) );
        val prog5 = interpretRest( order_seq( prog4 ) );
        val prog6 = ifValue( order_seq( prog5 ) );
        val prog7 = ifCons( order_seq( prog6 ) );
        val prog8 = removeDeadLhs( order_seq( prog7 ) );
        val prog9 = ifWhileToWhileIf( order_seq( prog8 ) );

        val new_prog = prog9;
    in
        if( new_prog = prog )
        then new_prog
        else optimize( new_prog )
    end
;

end
;
```

B.1.5 Partition structure

```
structure Partition =
struct

datatype Partition = P of Partition * Partition
   | D                 (* Depend on input *)
   | S                 (* is of BSV *)
;

fun toString( P( p1, p2 ) ) =
    "[ " ^ toString( p1 ) ^ ", " ^
    toString( p2 ) ^ " ]"
  | toString( D ) = "D"
  | toString( S ) = "S"
;

fun applyHd( P( p1, p2 ) ) = p1
  | applyHd( D ) = D
  | applyHd( S ) = S
;

fun applyTl( P( p1, p2 ) ) = p2
  | applyTl( D ) = D
  | applyTl( S ) = S
;

fun combine_pair( S, S ) = S
  | combine_pair( p1, p2 ) = P( p1, p2 )
;

fun combine( S, S ) = S
  | combine( S, D ) = D
  | combine( S, P( p1, p2 ) ) =
    combine_pair( combine( S, p1 ), combine( S, p2 ) )
  | combine( D, D ) = D
  | combine( D, S ) = D
  | combine( D, P( p1, p2 ) ) = P( p1, p2 )
  | combine( P( p1, p2 ), S ) =
    combine_pair( combine( p1, S ), combine( p2, S ) )
  | combine( P( p1, p2 ), D ) = P( p1, p2 )
  | combine( P( p1, p2 ), P( p3, p4 ) ) =
    P( combine( p1, p3 ), combine( p2, p4 ) )
;

end
;
```

B.1.6 CommandPartition structure

```
structure CommandPartition =
struct

open Expression;
open Command;
open Tools;

datatype CommandPartition = CPASSIGN of Expression
  | CPSEQ of CommandPartition *
```

```
                                    CommandPartition
  | CPWHILE of Partition.Partition *
                                      Expression *
                                      CommandPartition
  | CPIF of Expression *
                                   CommandPartition *
                                   CommandPartition

fun annotate( ASSIGN( e ), partition ) =
    ( CPASSIGN( e ),
      Expression.reduce_partition( e, partition ) )
  | annotate( SEQ( p1, p2 ), partition ) =
    let
val ( cp1, par1 ) = annotate( p1, partition );
val ( cp2, par2 ) = annotate( p2, par1 );
    in
( CPSEQ( cp1, cp2 ), par2 )
    end
  | annotate( WHILE( e, p ), partition ) =
    let
val ( cp, par ) = annotate( p, partition );
val cpar = Partition.combine( partition, par );
    in
( CPWHILE( cpar, e, cp ), cpar )
    end
  | annotate( IF( e, p1, p2 ), partition ) =
    let
val( cp1, par1 ) = annotate( p1, partition );
val( cp2, par2 ) = annotate( p2, partition );
        val new_par = Partition.combine( par1, par2 );
    in
( CPIF( e, cp1, cp2 ), new_par )
    end
;

fun toString( CPASSIGN e ) = varname ^ " := " ^
   Expression.toString( e )
  | toString( CPSEQ( c1, c2 ) ) =
    toString c1 ^ ";" ^ newline() ^
    toString c2
  | toString( CPWHILE( p, e, c ) ) =
    "while [" ^ Partition.toString( p ) ^ "] ( " ^
    Expression.toString( e )
    ^ " ) do" ^ startBlock() ^ newline()
    ^ toString( c ) ^ endBlock()
  | toString( CPIF( e, c1, c2 ) ) =
    "if( " ^ Expression.toString( e ) ^
    " ) then " ^ startBlock() ^ newline() ^
    toString( c1 ) ^ endBlock() ^ newline() ^
    "else" ^ startBlock() ^ newline() ^
    toString( c2 ) ^ endBlock() ^ newline()
;

fun toStringPP( CPASSIGN e, max, n ) = varname ^ " := " ^
       Expression.toStringPP( e, max, n )
  | toStringPP( CPSEQ( c1, c2 ), max, n ) =
    toStringPP( c1, max, n ) ^ ";" ^ newline() ^
    toStringPP( c2, max, n )
  | toStringPP( CPWHILE( p, e, c ), max, n ) =
    "while [" ^ Partition.toString( p ) ^ "] ( " ^
    Expression.toStringPP( e, max, n )
    ^ " ) do" ^ startBlock() ^ newline()
    ^ toStringPP( c, max, n ) ^ endBlock()
  | toStringPP( CPIF( e, c1, c2 ), max, n ) =
    "if( " ^ Expression.toStringPP( e, max, n ) ^
    " ) then " ^ startBlock() ^ newline() ^
    toStringPP( c1, max, n ) ^ endBlock() ^ newline() ^
    "else" ^ startBlock() ^ newline() ^
    toStringPP( c2, max, n ) ^ endBlock() ^ newline()
;

fun order_seq( CPASSIGN e ) = CPASSIGN( e )
  | order_seq( CPSEQ( CPSEQ( c1, c2 ), c3 ) ) =
    CPSEQ( c1, order_seq( CPSEQ( c2, c3 ) ) )
  | order_seq( CPSEQ( c1, c2 ) ) =
    CPSEQ( order_seq( c1 ), order_seq( c2 ) )
  | order_seq( CPWHILE( p, e, c ) ) =
    CPWHILE( p, e, order_seq( c ) )
  | order_seq( CPIF( e, c1, c2 ) ) =
    CPIF( e, order_seq( c1 ), order_seq( c2 ) )
;

fun inline( CPSEQ( CPASSIGN( e ), c1 ) ) =
    CPSEQ( CPASSIGN( e ), inline( c1 ) )
  | inline( CPSEQ( CPIF( e, c1, c2 ), c3 ) ) =
    CPIF( e, inline( order_seq( CPSEQ( c1, c3 ) ) ),
          inline( order_seq( CPSEQ( c2, c3 ) ) ) )
  | inline( CPASSIGN( e ) ) = CPASSIGN( e )
  | inline( CPSEQ( c1, c2 ) ) =
    order_seq( CPSEQ( inline( c1 ), inline( c2 ) ) )
  | inline( CPWHILE( p, e, c ) ) =
    CPWHILE( p, e, inline( c ) )
  | inline( CPIF( e, c1, c2 ) ) =
    CPIF( e, inline( c1 ), inline( c2 ) )
;

fun compress_transition( CPASSIGN( e ) ) =
    CPASSIGN( compact e )
  | compress_transition( CPSEQ( CPASSIGN e1, CPASSIGN e2 ) ) =
    CPASSIGN( compact( substitute( e1, e2 ) ) )
  | compress_transition( CPSEQ( CPASSIGN e1,
                                CPSEQ( CPASSIGN e2, c2 ) ) ) =
    let
val tmp = substitute( e1, e2 );
val assign_exp = compact( tmp );
    in
CPSEQ( CPASSIGN( assign_exp ), c2 )
    end
  | compress_transition( CPSEQ( c1, CPASSIGN( VAR ) ) ) = c1
  | compress_transition( CPSEQ( CPIF( e, c1, c2 ), c3 ) ) =
    CPIF( e, CPSEQ( c1, c3 ), CPSEQ( c2, c3 ) )
  | compress_transition( CPSEQ( c1, c2 ) ) =
    CPSEQ( compress_transition( c1 ),
           compress_transition( c2 ) )
  | compress_transition( CPWHILE( p, e, c ) ) =
    CPWHILE( p, e, compress_transition( c ) )
  | compress_transition( CPIF( e, c1, c2 ) ) =
    CPIF( e,
  compress_transition( c1 ),
  compress_transition( c2 ) )

and compress_transitions( prog ) =
    let
val new_prog = compress_transition( prog );
    in
if prog = new_prog
then new_prog
else compress_transitions( order_seq( new_prog ) )
    end
;

fun erase( CPASSIGN( e ) ) =
    ASSIGN( e )
  | erase( CPSEQ( p1, p2 ) ) =
    SEQ( erase( p1 ), erase( p2 ) )
  | erase( CPWHILE( p, e, c ) ) =
    WHILE( e, erase( c ) )
  | erase( CPIF( e, p1, p2 ) ) =
    IF( e, erase( p1 ), erase( p2 ) )
;

end
;
```

B.1.7 Tools

```
structure Tools =
struct

val varname = "X";

val indent = ref 0;

fun indent_string( 0 ) = ""
  | indent_string( n ) = " " ^ indent_string( n - 1 )
;

fun toString_indent() =
indent_string( !indent )
;

fun newline() = "\n" ^ toString_indent();

fun startBlock() =
    let
val _ = indent := !indent + 2
    in
""
    end
;

fun endBlock() =
    let
val _ = indent := !indent - 2
    in
""
    end
;

fun string_append( s1, s2 ) = s1 ^ s2;

end
;
```

B.1.8 SelfInterpreter program

```
X := cons( hd( X ), cons( N, tl( X ) ) );
while( hd( X ) ) do
 if( hd( hd( hd( X ) ) ) ) then
  X := cons(
      cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
      cons( hd( tl( X ) ), tl( tl( X ) ) )
     );
  if( hd( hd( hd( X ) ) ) ) then
   X := cons(
       cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
       cons( hd( tl( X ) ), tl( tl( X ) ) )
      );
   if( hd( hd( hd( X ) ) ) ) then
    X := cons(
        cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
        cons( hd( tl( X ) ), tl( tl( X ) ) )
       );
    if( hd( hd( hd( X ) ) ) ) then
     X := cons(
         cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
         cons( hd( tl( X ) ), tl( tl( X ) ) )
        );
     if( hd( hd( hd( X ) ) ) ) then
      X := cons(
          cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
          cons( hd( tl( X ) ), tl( tl( X ) ) )
         );
      if( hd( hd( hd( X ) ) ) ) then
       X := cons(
           cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
       cons( hd( tl( X ) ), tl( tl( X ) ) )
      );
if( hd( hd( hd( X ) ) ) ) then
  X := cons(
         cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
         cons( hd( tl( X ) ), tl( tl( X ) ) )
        );
  if( hd( hd( hd( X ) ) ) ) then
    X := cons(
           cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
           cons( hd( tl( X ) ), tl( tl( X ) ) )
          );
    if( hd( hd( hd( X ) ) ) ) then
      X := cons(
             cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
             cons( hd( tl( X ) ), tl( tl( X ) ) )
            );
      if( hd( hd( hd( X ) ) ) ) then
        X := cons(
               cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
               cons( hd( tl( X ) ), tl( tl( X ) ) )
              );
        if( hd( hd( hd( X ) ) ) ) then
          X := cons(
                 cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
                 cons( hd( tl( X ) ), tl( tl( X ) ) )
                );
          if( hd( hd( hd( X ) ) ) ) then
            X := cons(
                   cons( cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ),
                   cons( hd( tl( X ) ), tl( tl( X ) ) )
                  );
```

```
if( hd( hd( hd( X ) ) ) ) then
 X := cons(
     cons(
      cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ),
      tl( hd( X ) )
     ),
     cons( hd( tl( X ) ), tl( tl( X ) ) )
    );
 if( hd( hd( hd( X ) ) ) ) then
  X := cons(
      cons(
       cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ),
       tl( hd( X ) )
      ),
      cons( hd( tl( X ) ), tl( tl( X ) ) )
     );
  if( hd( hd( hd( X ) ) ) ) then
   X := cons(
       cons(
        cons( hd( hd( hd( hd( X ) ) ) ), tl( hd( hd( X ) ) ) ),
        tl( hd( X ) )
       ),
       cons( hd( tl( X ) ), tl( tl( X ) ) )
      );
   X := X
  else
   if( hd( hd( tl( X ) ) ) ) then
    X := cons(
        cons( hd( hd( tl( hd( X ) ) ) ), tl( tl( hd( X ) ) ) ),
        cons( hd( tl( X ) ), tl( tl( X ) ) )
       )
   else
```

```
          X := cons(
                 cons( tl( hd( tl( hd( X ) ) ) ), tl( tl( hd( X ) ) ) ),
                 cons( hd( tl( X ) ), tl( tl( X ) ) )
                )
        ;
        X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) )

    else
      X := cons(
             hd( X ),
             cons(
               cons( tl( hd( hd( tl( X ) ) ) ), tl( hd( tl( X ) ) ) ),
               tl( tl( X ) )
              )
            );
      X := cons( tl( hd( X ) ), cons( hd( tl( X ) ), tl( tl( X ) ) ) )

  else
    X := cons( tl( hd( X ) ), cons( hd( tl( X ) ), tl( tl( X ) ) ) );
    X := cons(
           hd( X ),
           cons(
             cons(
               cons( hd( tl( hd( tl( X ) ) ) ), hd( hd( tl( X ) ) ) ),
               tl( tl( hd( tl( X ) ) ) )
              ),
             tl( tl( X ) )
            )
          )

else
  X := cons(
               hd( X ),
               cons( cons( hd( hd( hd( tl( X ) ) ) ), tl( hd( tl( X ) ) ) ), tl( tl( X ) ) )
              );
        X := cons( tl( hd( X ) ), cons( hd( tl( X ) ), tl( tl( X ) ) ) )

    else
      if( hd( hd( tl( X ) ) ) ) then
        X := cons(
               cons(
                 tl( tl( hd( tl( hd( X ) ) ) ) ),
                 cons( hd( tl( hd( X ) ) ), tl( tl( hd( X ) ) ) )
                ),
               cons( hd( tl( X ) ), tl( tl( X ) ) )
              );
        X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) )
      else
        X := cons( tl( tl( hd( X ) ) ), cons( hd( tl( X ) ), tl( tl( X ) ) ) );
        X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) )


  else
    X := cons( tl( hd( X ) ), cons( hd( tl( X ) ), tl( tl( X ) ) ) );
    X := cons( hd( X ), cons( hd( tl( X ) ), hd( hd( tl( X ) ) ) ) );
    X := cons( hd( X ), cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) ) )

else
  X := cons(
         cons(
           tl( hd( hd( X ) ) ),
           cons(
             cons( <<<<<<<<<<<<<N.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>, N ),
             tl( hd( X ) )
```

```
                    )
                  ),
                 cons( hd( tl( X ) ), tl( tl( X ) ) )
                )

      else
        X := cons(
               cons(
                 tl( hd( hd( X ) ) ),
                 cons( cons( <<<<<<<<<<<N.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>, N ), tl( hd( X ) ) )
                ),
               cons( hd( tl( X ) ), tl( tl( X ) ) )
              )

    else
      X := cons(
             cons(
               hd( tl( hd( hd( X ) ) ) ),
               cons(
                 tl( tl( hd( hd( X ) ) ) ),
                 cons( cons( <<<<<<<<<<<<N.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>, N ), tl( hd( X ) ) )
                )
              ),
             cons( hd( tl( X ) ), tl( tl( X ) ) )
            )

  else
    X := cons( hd( X ), cons( cons( tl( hd( hd( X ) ) ), hd( tl( X ) ) ), tl( tl( X ) ) ) );
    X := cons( tl( hd( X ) ), cons( hd( tl( X ) ), tl( tl( X ) ) ) )

else
  X := cons( tl( hd( X ) ), cons( hd( tl( X ) ), tl( tl( X ) ) ) );
```

```
    X := cons( hd( X ), cons( cons( tl( tl( X ) ), hd( tl( X ) ) ), tl( tl( X ) ) ) )

  else
   X := cons(
       cons(
        hd( tl( hd( hd( X ) ) ) ),
        cons(
         cons( <<<<<<<<<<<<<<N.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>, N ),
         cons( tl( tl( hd( hd( X ) ) ) ), tl( hd( X ) ) )
        )
       ),
       cons( hd( tl( X ) ), tl( tl( X ) ) )
      )

 else
  X := cons(
      cons(
       hd( tl( hd( hd( X ) ) ) ),
       cons(
        cons( <<<<<<<<<<N.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>, N ),
        cons( cons( <<N.N>.N>, tl( hd( hd( X ) ) ) ), tl( hd( X ) ) )
       )
      ),
      cons( hd( tl( X ) ), tl( tl( X ) ) )
     )

else
 X := cons(
     cons( hd( tl( hd( hd( X ) ) ) ), cons( tl( tl( hd( hd( X ) ) ) ), tl( hd( X ) ) ) ),
     cons( hd( tl( X ) ), tl( tl( X ) ) )
    )
  else
    X := cons(
           cons( tl( hd( hd( X ) ) ), cons( <<<<<<<<<<N.N>.N>.N>.N>.N>.N>.N>.N>.N>.N>, tl( hd( X ) ) ) ),
           cons( hd( tl( X ) ), tl( tl( X ) ) )
          )
  ;
X := tl( tl( X ) )
```

B.1.9 Specializer structure

```
structure Specializer =
struct

open Partition;
open CommandPartition;
open AbstractValue;

exception SpecializerError of string;
exception MaxLoopCount of int;

val whileUnrolls = ref 0;

val maxUnrolls = ref 50;

fun reduce_data( VAR, var) = var
  | reduce_data( VA v, var ) = AbstractValue.AV v
  | reduce_data( HD( e ), var ) =
    AbstractValue.applyHd( reduce_data( e, var ) )
  | reduce_data( TL( e ), var ) =
    AbstractValue.applyTl( reduce_data( e, var ) )
  | reduce_data( CONS( e1, e2 ), var ) =
    AbstractValue.reduceCons( reduce_data( e1, var ),
                              reduce_data( e2, var ) )
;

fun generateStaticPaths( D, e ) = []
  | generateStaticPaths( S, e ) = [ e( VAR ) ]
  | generateStaticPaths( P( l, r ), e ) =
    let
val lsp = generateStaticPaths( l, fn x => HD( e( x ) ) );
val rsp = generateStaticPaths( r, fn x => TL( e( x ) ) );
    in
lsp @ rsp
    end
;

fun generateStaticPathValues( partition, var ) =
    let
        val staticPaths = generateStaticPaths( partition, fn x => x );
    in
        map ( fn x => reduce_data( x, var ) ) staticPaths
    end

fun substitute_statics( CONS( e1, e2 ), var, partition ) =
    let
        val e1p = reduce_partition( e1, partition );
        val e2p = reduce_partition( e2, partition );
        val e1r = if( e1p = S )
                    then toExpression( reduce_data( e1, var ) )
                    else substitute_statics( e1, var, partition );
        val e2r = if( e2p = S )
                    then toExpression( reduce_data( e2, var ) )
                    else substitute_statics( e2, var, partition );
    in
        CONS( e1r, e2r )
    end
  | substitute_statics( e, var, partition ) = e
;

fun specialize_program( CPASSIGN( e ), var,
                        partition, seenStates ) =
    let
val new_data = reduce_data( e, var );
val new_partition = reduce_partition( e, partition );
        val er = substitute_statics( e, var, partition );
    in
( ASSIGN( er ), new_data, new_partition, false,
          generateStaticPathValues( partition, var ) )
    end
  | specialize_program( CPSEQ( c1, c2 ), var,
                        partition, seenStates ) =
    let
val ( p1, d1, part1, lf1, codeFound1 ) =
            specialize_program( c1, var, partition,
                                seenStates );
val ( p2, d2, part2, lf2, codeFound2 ) =
            specialize_program( c2, d1, part1,
                                seenStates );
val codeFound = if( lf1 )
                        then codeFound1
                        else if( lf2 )
                        then codeFound2
                        else generateStaticPathValues( partition, var )
    in
( SEQ( p1, p2 ), d2, part2, lf1 orelse lf2, codeFound )
    end
  | specialize_program( CPWHILE( p, e, c ), var,
                        partition, seenStates ) =
    let
        val _ = if( !whileUnrolls > !maxUnrolls )
                  then raise MaxLoopCount( !whileUnrolls )
                  else ();
        val _ = whileUnrolls := !whileUnrolls + 1;
val er = reduce_data( e, var );
val ePart = reduce_partition( e, Partition.combine( partition, p ) );
val choise = if( ePart = S ) then choose( er ) else DD;

val staticPathValues = generateStaticPathValues( p, var );

val new_seenStates = staticPathValues :: seenStates;

val found = ( List.find ( fn x => staticPathValues = x ) seenStates )
                    <> NONE;

val (p1,d1,part1,lf1,codeFound1) =
            if found
    then ( ASSIGN( VAR ), var, partition,
                   true, staticPathValues )
    else specialize_while( choise, p, e, c, var,
                                   Partition.combine( p, partition ),
                                   new_seenStates )

val ( p2, d2, part2, lf2, codeFound2 ) =
            if( ( not found ) andalso
                lf1 andalso
                ( staticPathValues = codeFound1 ) )
    then ( WHILE( e, p1 ), d1, part1, false, staticPathValues )
    else ( p1, d1, part1, lf1, codeFound1 )

        val _ = whileUnrolls := !whileUnrolls - 1;
    in
        ( p2, d2, part2, lf2, codeFound2 )
    end
  | specialize_program( CPIF( e, c1, c2 ), var,
                        partition, seenStates ) =
    let
val er = reduce_data( e, var );
val ePart = reduce_partition( e, partition );
val choise = if( ePart = S ) then choose( er ) else DD;
    in
specialize_if( choise, e, c1, c2, var,
                       partition, seenStates )
    end

and specialize_if( DD, e, c1, c2, var,
                   partition, seenStates ) =
    let
val er = reduce_data( e, var );
val ( p1, d1, part1, lf1, codeFound1 ) =
            specialize_program( c1, var, partition,
                                seenStates );
val ( p2, d2, part2, lf2, codeFound2 ) =
            specialize_program( c2, var, partition,
                                seenStates );
```

```
val returnedCode = if( lf1 andalso not lf2)
   then d1
   else if not lf1 andalso lf2
   then d2
   else if not lf1 andalso not lf2
   then AD
   else AD

val codeFound = if( lf1 )
                        then codeFound1
                        else if( lf2 )
                        then codeFound2
                        else generateStaticPathValues( partition, var )
    in
( IF( e, p1, p2 ), returnedCode,
          Partition.combine( part1, part2 ),
          lf1 orelse lf2, codeFound )
    end
  | specialize_if( DC, e, c1, c2, var,
                   partition, seenStates ) =
    specialize_if( SC, e, c1, c2, var,
                   partition, seenStates )
  | specialize_if( SC, e, c1, c2, var,
                   partition, seenStates ) =
    specialize_program( c1, var,
                        partition, seenStates )
  | specialize_if( SN, e, c1, c2, var,
                   partition, seenStates ) =
    specialize_program( c2, var, partition, seenStates )

and specialize_while( DD, p, e, c, var,
                      partition, seenStates ) =
    ( WHILE( e, erase( c ) ), AD,
      partition, false, generateStaticPathValues( partition, var ) )
  | specialize_while( DC, p, e, c, var,
                      partition, seenStates ) =
    specialize_while( SC, p, e, c, var,
                      partition, seenStates )
  | specialize_while( SC, p, e, c, var,
                      partition, seenStates ) =
    let
val unrolled =  inline( CommandPartition.order_seq(
                                CPSEQ( c, CPWHILE( p, e, c) ) ) );
    in
specialize_program( unrolled, var, partition, seenStates )
    end
  | specialize_while( SN, p, e, c, var,
                      partition, seenStates ) =
    let
```

```
        val spv = generateStaticPathValues( partition, var );
    in
        ( ASSIGN( VAR ), var, partition, false, spv )
    end
;

fun specializer( mu, program, static ) =
    let
        val oldMu = !maxUnrolls;
        val _ = maxUnrolls := mu
val source = SEQ( ASSIGN( CONS( VA static, VAR ) ), program );
val compressed_source = Command.optimize( source );
val ( annotated_source, par ) =
            CommandPartition.annotate( compressed_source, D );
val ordered_source = CommandPartition.order_seq( annotated_source );
val bta_source = CommandPartition.inline( ordered_source );
val compressed_bta_source =
            CommandPartition.compress_transitions(
            CommandPartition.order_seq( bta_source ) );
val ( residual, var, partition, lf, cf ) =
            specialize_program( compressed_bta_source, AD, D, [] );
        val target = Command.optimize( residual );
val target = Command.compressTransitions( target );
val target = Command.optimize( target );
        val _ = maxUnrolls := oldMu
    in
target
    end
    handle MaxLoopCount( c ) =>
           ASSIGN( VA( C( C(N,C(N,C(N,C(N,N)))),
                          C(C(C(C(N,N),N),N),N) ) ) )
;

end
;
```

B.2 IF tests

B.2.1 Assign value

program

```
X := <N.N>
```

data

```
static   N
dynamic  <N.N>
```

```
expected <N.N>
```

B.2.2 Sequence

program

```
X := <N.N>;
X := <<N.N>.N>
```

data

```
static   N
dynamic  <N.N>
expected <<N.N>.N>
```

B.2.3 while none

program

```
while( N ) do
  X := <N.<N.N>>
```

data

```
static   N
dynamic  N
expected <N.N>
```

B.2.4 while once

program

```
while( hd( X ) ) do
  X := <N.N>
```

data

```
static   <N.N>
dynamic  <N.N>
expected <N.N>
```

B.2.5 while many

program

```
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), tl( X ) )
```

data

```
static   <N.<N.<N.<N.N>>>>
dynamic  <<N.N>.N>
expected <N.<<N.N>.N>>
```

B.2.6 VAR

program

```
X := X
```

data

```
static   N
dynamic  N
expected <N.N>
```

B.2.7 CONS

program

```
X := cons( X, X )
```

data

```
static   N
dynamic  N
expected <<N.N>.<N.N>>
```

B.2.8 HD VAR

program

```
X := hd( X )
```

data

```
static   <N.N>
dynamic  N
expected <N.N>
```

B.2.9 TL value

program

```
X := tl( X )
```

data

```
static   N
dynamic  <N.N>
expected <N.N>
```

B.2.10 IF false

program

```
if( N ) then
  X := <N.N>
else
  X := <N.<N.N>>
```

data

```
static   N
dynamic  N
expected <N.<N.N>>
```

B.2.11 IF true

program

```
if( <N.N> ) then
  X := <N.N>
else
  X := <N.<N.N>>
```

data

```
static   N
dynamic  N
expected <N.N>
```

B.2.12 List Reverse

program

```
X := cons( N, X );
while( tl( X ) ) do
  X := cons( cons( hd( tl( X ) ), hd( X ) ), tl( tl( X ) ) );
X := hd( X )
```

data

```
static   <N.N>
dynamic  <N.N>
expected <N.<<N.N>.N>>
```

B.2.13 DeadCode

program

```
while( tl( X ) ) do
  X := cons( hd( X ), tl( tl( X ) ) );
X := <N.N>
```

data

```
static   <N.<N.N>>
dynamic  <N.<N.N>>
expected <N.N>
```

B.2.14 While opt fails

program

```
if( X ) then
  while( tl( X ) ) do
    X := <N.N>
else
  X := X
```

data

```
static   <N.N>
dynamic  N
expected <<N.N>.N>
```

B.2.15 Append

program

```
X := cons( N, X );
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

data

```
static   <<N.N>.<<N.<N.N>>.<<N.<N.<N.N>>>.N>>>
dynamic  <<<N.N>.N>.<<<<N.N>.N>.N>.N>>
expected <<N.N>.<<N.<N.N>>.<<N.<N.<N.N>>>.<<<N.N>.N>.<<<<N.N>.N>.N>.N>>>>>
```

B.2.16 Append 2

program

```
X := cons( N, cons( tl( X ), hd( X ) ) );
while( hd( tl( X ) ) ) do
  X := cons(
         cons( hd( hd( tl( X ) ) ), hd( X ) ),
         cons( tl( hd( tl( X ) ) ), tl( tl( X ) ) )
        );
X := cons( hd( X ), tl( tl( X ) ) );
while( hd( X ) ) do
  X := cons( tl( hd( X ) ), cons( hd( hd( X ) ), tl( X ) ) );
X := tl( X )
```

data

```
static   <<<N.N>.N>.<<<<N.N>.N>.N>.N>>
dynamic  <<N.N>.<<N.<N.N>>.<<N.<N.<N.N>>>.N>>>
expected <<N.N>.<<N.<N.N>>.<<N.<N.<N.N>>>.<<<N.N>.N>.<<<<N.N>.N>.N>.N>>>>>
```

B.2.17 Compare false

program

```
X := cons( <N.N>, X );
while( hd( X ) ) do
  if( hd( tl( X ) ) ) then
    if( tl( tl( X ) ) ) then
      if( hd( hd( tl( X ) ) ) ) then
        if( hd( tl( tl( X ) ) ) ) then
          X := cons(
                 <N.N>,
                 cons(
                   cons(
                     hd( hd( hd( tl( X ) ) ) ),
                     cons(
                       tl( hd( hd( tl( X ) ) ) ),
                       tl( hd( tl( X ) ) )
                      )
                    ),
                   cons(
                     hd( hd( tl( tl( X ) ) ) ),
                     cons(
                       tl( hd( tl( tl( X ) ) ) ),
                       tl( tl( tl( X ) ) )
                      )
                    )
                  )
```

```
                )
        else
          X := cons( N, N )

      else
        if( hd( tl( tl( X ) ) ) ) then
          X := cons( N, N )
        else
          X := cons(
                 <N.N>,
                 cons( tl( hd( tl( X ) ) ), tl( tl( tl( X ) ) ) )
                )


    else
      X := cons( N, N )

  else
    if( tl( tl( X ) ) ) then
      X := cons( N, N )
    else
      X := cons( N, <N.N> )

  ;
X := tl( X )
```

data

```
static   <<N.N>.N>
dynamic  <N.N>
expected N
```

B.2.18 Compare true

program

```
X := cons( <N.N>, X );
while( hd( X ) ) do
  if( hd( tl( X ) ) ) then
    if( tl( tl( X ) ) ) then
      if( hd( hd( tl( X ) ) ) ) then
        if( hd( tl( tl( X ) ) ) ) then
          X := cons(
                 <N.N>,
                 cons(
                   cons(
                     hd( hd( hd( tl( X ) ) ) ),
                     cons(
                       tl( hd( hd( tl( X ) ) ) ),
                       tl( hd( tl( X ) ) )
                      )
                    ),
                   cons(
                     hd( hd( tl( tl( X ) ) ) ),
                     cons(
                       tl( hd( tl( tl( X ) ) ) ),
                       tl( tl( tl( X ) ) )
                      )
                    )
                  )
                )
        else
          X := cons( N, N )

      else
        if( hd( tl( tl( X ) ) ) ) then
          X := cons( N, N )
        else
          X := cons(
                 <N.N>,
                 cons( tl( hd( tl( X ) ) ), tl( tl( tl( X ) ) ) )
                )


    else
      X := cons( N, N )

  else
    if( tl( tl( X ) ) ) then
      X := cons( N, N )
    else
      X := cons( N, <N.N> )

  ;
X := tl( X )
```

data

```
static   <<N.N>.N>
dynamic  <<N.N>.N>
expected <N.N>
```

B.2.19 KMP

program

```
X := cons( <N.N>, cons( X, X ) );
while( hd( X ) ) do
```

```
if( hd( hd( tl( X ) ) ) ) then
 if( tl( hd( tl( X ) ) ) ) then
  X := cons(
      cons( hd( hd( hd( tl( X ) ) ) ), hd( tl( hd( tl( X ) ) ) ) ),
      tl( X )
     );
  X := cons( <N.N>, X );
  while( hd( X ) ) do
   if( hd( hd( tl( X ) ) ) ) then
    if( tl( hd( tl( X ) ) ) ) then
     if( hd( hd( hd( tl( X ) ) ) ) ) then
      if( hd( tl( hd( tl( X ) ) ) ) ) then
       X := cons(
           <N.N>,
           cons(
            cons(
             cons(
              hd( hd( hd( hd( tl( X ) ) ) ) ),
              cons(
               tl( hd( hd( hd( tl( X ) ) ) ) ),
               tl( hd( hd( tl( X ) ) ) )
              )
             ),
             cons(
              hd( hd( tl( hd( tl( X ) ) ) ) ),
              cons(
               tl( hd( tl( hd( tl( X ) ) ) ) ),
               tl( tl( hd( tl( X ) ) ) )
              )
             )
            ),
            tl( tl( X ) )
           )
          )
      else
       X := cons( N, cons( N, tl( tl( X ) ) ) )

     else
      if( hd( tl( hd( tl( X ) ) ) ) ) then
       X := cons( N, cons( N, tl( tl( X ) ) ) )
      else
       X := cons(
           <N.N>,
           cons(
            cons(
             tl( hd( hd( tl( X ) ) ) ),
             tl( tl( hd( tl( X ) ) ) )
            ),
            tl( tl( X ) )
```

```
                        )
                      )


          else
            X := cons( N, cons( N, tl( tl( X ) ) ) )

        else
          if( tl( hd( tl( X ) ) ) ) then
            X := cons( N, cons( N, tl( tl( X ) ) ) )
          else
            X := cons( N, cons( <N.N>, tl( tl( X ) ) ) )

        ;
      X := tl( X );
      if( hd( X ) ) then
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( tl( hd( hd( X ) ) ), tl( tl( hd( X ) ) ) ),
                 tl( X )
                )
              )
      else
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) ),
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) )
                )
              )

    else
      X := cons( N, N )

  else
    X := cons( N, <N.N> )
  ;
X := tl( X )
```

data

```
static   N
dynamic  <N.N>
expected <N.N>
```

B.2.20 KMP 1

program

```
X := cons( <N.N>, cons( X, X ) );
while( hd( X ) ) do
  if( hd( hd( tl( X ) ) ) ) then
    if( tl( hd( tl( X ) ) ) ) then
      X := cons(
             cons( hd( hd( hd( tl( X ) ) ) ), hd( tl( hd( tl( X ) ) ) ) ),
             tl( X )
            );
      X := cons( <N.N>, X );
      while( hd( X ) ) do
        if( hd( hd( tl( X ) ) ) ) then
          if( tl( hd( tl( X ) ) ) ) then
            if( hd( hd( hd( tl( X ) ) ) ) ) then
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           cons(
                             hd( hd( hd( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( hd( hd( tl( X ) ) ) ) ),
                               tl( hd( hd( tl( X ) ) ) )
                              )
                            ),
                           cons(
                             hd( hd( tl( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( tl( hd( tl( X ) ) ) ) ),
                               tl( tl( hd( tl( X ) ) ) )
                              )
                            )
                          ),
                         tl( tl( X ) )
                        )
                      )
              else
                X := cons( N, cons( N, tl( tl( X ) ) ) )

            else
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons( N, cons( N, tl( tl( X ) ) ) )
              else
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           tl( hd( hd( tl( X ) ) ) ),
                           tl( tl( hd( tl( X ) ) ) )
                          ),
                         tl( tl( X ) )
                        )
                      )


          else
            X := cons( N, cons( N, tl( tl( X ) ) ) )

        else
          if( tl( hd( tl( X ) ) ) ) then
            X := cons( N, cons( N, tl( tl( X ) ) ) )
          else
            X := cons( N, cons( <N.N>, tl( tl( X ) ) ) )

        ;
      X := tl( X );
      if( hd( X ) ) then
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( tl( hd( hd( X ) ) ), tl( tl( hd( X ) ) ) ),
                 tl( X )
                )
              )
      else
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) ),
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) )
                )
              )

    else
      X := cons( N, N )

  else
    X := cons( N, <N.N> )
  ;
X := tl( X )
```

data

```
static   <<<N.N>.N>.N>
dynamic  N
expected N
```

B.2.21 KMP 2

program

```
X := cons( <N.N>, cons( X, X ) );
while( hd( X ) ) do
  if( hd( hd( tl( X ) ) ) ) then
    if( tl( hd( tl( X ) ) ) ) then
      X := cons(
             cons( hd( hd( hd( tl( X ) ) ) ), hd( tl( hd( tl( X ) ) ) ) ),
             tl( X )
            );
      X := cons( <N.N>, X );
      while( hd( X ) ) do
        if( hd( hd( tl( X ) ) ) ) then
          if( tl( hd( tl( X ) ) ) ) then
            if( hd( hd( hd( tl( X ) ) ) ) ) then
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           cons(
                             hd( hd( hd( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( hd( hd( tl( X ) ) ) ) ),
                               tl( hd( hd( tl( X ) ) ) )
                              )
                            ),
                           cons(
                             hd( hd( tl( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( tl( hd( tl( X ) ) ) ) ),
                               tl( tl( hd( tl( X ) ) ) )
                              )
                            )
                          ),
                         tl( tl( X ) )
                        )
                      )
              else
                X := cons( N, cons( N, tl( tl( X ) ) ) )

            else
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons( N, cons( N, tl( tl( X ) ) ) )
              else
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           tl( hd( hd( tl( X ) ) ) ),
                           tl( tl( hd( tl( X ) ) ) )
                          ),
                         tl( tl( X ) )
                        )
                      )


          else
            X := cons( N, cons( N, tl( tl( X ) ) ) )

        else
          if( tl( hd( tl( X ) ) ) ) then
            X := cons( N, cons( N, tl( tl( X ) ) ) )
          else
            X := cons( N, cons( <N.N>, tl( tl( X ) ) ) )

        ;
      X := tl( X );
      if( hd( X ) ) then
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( tl( hd( hd( X ) ) ), tl( tl( hd( X ) ) ) ),
                 tl( X )
                )
              )
      else
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) ),
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) )
                )
              )

    else
      X := cons( N, N )

  else
    X := cons( N, <N.N> )
  ;
X := tl( X )
```

data

```
static   <<<N.N>.N>.N>
dynamic  <N.N>
expected N
```

B.2.22 KMP 3

program

```
X := cons( <N.N>, cons( X, X ) );
while( hd( X ) ) do
  if( hd( hd( tl( X ) ) ) ) then
    if( tl( hd( tl( X ) ) ) ) then
      X := cons(
             cons( hd( hd( hd( tl( X ) ) ) ), hd( tl( hd( tl( X ) ) ) ) ),
             tl( X )
            );
      X := cons( <N.N>, X );
      while( hd( X ) ) do
        if( hd( hd( tl( X ) ) ) ) then
          if( tl( hd( tl( X ) ) ) ) then
            if( hd( hd( hd( tl( X ) ) ) ) ) then
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           cons(
                             hd( hd( hd( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( hd( hd( tl( X ) ) ) ) ),
                               tl( hd( hd( tl( X ) ) ) )
                              )
                            ),
                           cons(
                             hd( hd( tl( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( tl( hd( tl( X ) ) ) ) ),
                               tl( tl( hd( tl( X ) ) ) )
                              )
                            )
                          ),
                         tl( tl( X ) )
                        )
                      )
              else
                X := cons( N, cons( N, tl( tl( X ) ) ) )

            else
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons( N, cons( N, tl( tl( X ) ) ) )
              else
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           tl( hd( hd( tl( X ) ) ) ),
                           tl( tl( hd( tl( X ) ) ) )
                          ),
                         tl( tl( X ) )
                        )
                      )


          else
            X := cons( N, cons( N, tl( tl( X ) ) ) )

        else
          if( tl( hd( tl( X ) ) ) ) then
            X := cons( N, cons( N, tl( tl( X ) ) ) )
          else
            X := cons( N, cons( <N.N>, tl( tl( X ) ) ) )

        ;
      X := tl( X );
      if( hd( X ) ) then
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( tl( hd( hd( X ) ) ), tl( tl( hd( X ) ) ) ),
                 tl( X )
                )
              )
      else
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) ),
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) )
                )
              )
    else
      X := cons( N, N )

  else
    X := cons( N, <N.N> )
  ;
X := tl( X )
```

data

```
static   <<<N.N>.N>.N>
dynamic  <<N.N>.N>
expected N
```

B.2.23 KMP 4

program

```
X := cons( <N.N>, cons( X, X ) );
while( hd( X ) ) do
  if( hd( hd( tl( X ) ) ) ) then
    if( tl( hd( tl( X ) ) ) ) then
      X := cons(
             cons( hd( hd( hd( tl( X ) ) ) ), hd( tl( hd( tl( X ) ) ) ) ),
             tl( X )
            );
      X := cons( <N.N>, X );
      while( hd( X ) ) do
        if( hd( hd( tl( X ) ) ) ) then
          if( tl( hd( tl( X ) ) ) ) then
            if( hd( hd( hd( tl( X ) ) ) ) ) then
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           cons(
                             hd( hd( hd( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( hd( hd( tl( X ) ) ) ) ),
                               tl( hd( hd( tl( X ) ) ) )
                              )
                            ),
                           cons(
                             hd( hd( tl( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( tl( hd( tl( X ) ) ) ) ),
                               tl( tl( hd( tl( X ) ) ) )
                              )
```

```
                      )
                    ),
                   tl( tl( X ) )
                  )
                )
        else
          X := cons( N, cons( N, tl( tl( X ) ) ) )

      else
        if( hd( tl( hd( tl( X ) ) ) ) ) then
          X := cons( N, cons( N, tl( tl( X ) ) ) )
        else
          X := cons(
                 <N.N>,
                 cons(
                   cons(
                     tl( hd( hd( tl( X ) ) ) ),
                     tl( tl( hd( tl( X ) ) ) )
                    ),
                   tl( tl( X ) )
                  )
                )


    else
      X := cons( N, cons( N, tl( tl( X ) ) ) )

  else
    if( tl( hd( tl( X ) ) ) ) then
      X := cons( N, cons( N, tl( tl( X ) ) ) )
    else
      X := cons( N, cons( <N.N>, tl( tl( X ) ) ) )

  ;
X := tl( X );
if( hd( X ) ) then
  X := tl( X );
  X := cons(
         <N.N>,
         cons(
           cons( tl( hd( hd( X ) ) ), tl( tl( hd( X ) ) ) ),
           tl( X )
          )
        )
else
  X := tl( X );
  X := cons(
         <N.N>,
         cons(
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) ),
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) )
                )
              )

    else
      X := cons( N, N )

  else
    X := cons( N, <N.N> )
  ;
X := tl( X )
```

data

```
static   <<N.N>.N>
dynamic  <<N.N>.<<<N.N>.N>.N>>
expected <N.N>
```

B.2.24 KMP large true

program

```
X := cons( <N.N>, cons( X, X ) );
while( hd( X ) ) do
  if( hd( hd( tl( X ) ) ) ) then
    if( tl( hd( tl( X ) ) ) ) then
      X := cons(
             cons( hd( hd( hd( tl( X ) ) ) ), hd( tl( hd( tl( X ) ) ) ) ),
             tl( X )
            );
      X := cons( <N.N>, X );
      while( hd( X ) ) do
        if( hd( hd( tl( X ) ) ) ) then
          if( tl( hd( tl( X ) ) ) ) then
            if( hd( hd( hd( tl( X ) ) ) ) ) then
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           cons(
                             hd( hd( hd( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( hd( hd( tl( X ) ) ) ) ),
                               tl( hd( hd( tl( X ) ) ) )
                              )
                            ),
                           cons(
                             hd( hd( tl( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( tl( hd( tl( X ) ) ) ) ),
                               tl( tl( hd( tl( X ) ) ) )
                              )
                            )
                          ),
                         tl( tl( X ) )
                        )
                      )
              else
                X := cons( N, cons( N, tl( tl( X ) ) ) )

            else
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons( N, cons( N, tl( tl( X ) ) ) )
              else
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           tl( hd( hd( tl( X ) ) ) ),
                           tl( tl( hd( tl( X ) ) ) )
                          ),
                         tl( tl( X ) )
                        )
                      )


          else
            X := cons( N, cons( N, tl( tl( X ) ) ) )

        else
          if( tl( hd( tl( X ) ) ) ) then
            X := cons( N, cons( N, tl( tl( X ) ) ) )
          else
            X := cons( N, cons( <N.N>, tl( tl( X ) ) ) )

        ;
      X := tl( X );
      if( hd( X ) ) then
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( tl( hd( hd( X ) ) ), tl( tl( hd( X ) ) ) ),
                 tl( X )
                )
              )
      else
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) ),
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) )
                )
              )

    else
      X := cons( N, N )

  else
    X := cons( N, <N.N> )
  ;
X := tl( X )
```

data

```
static   <<<N.N>.N>.<<N.N>.N>>
dynamic  <<N.N>.<<<N.N>.N>.<<N.N>.<<<N.N>.N>.N>>>>
expected <N.N>
```

B.2.25 KMP large false

program

```
X := cons( <N.N>, cons( X, X ) );
while( hd( X ) ) do
  if( hd( hd( tl( X ) ) ) ) then
    if( tl( hd( tl( X ) ) ) ) then
      X := cons(
             cons( hd( hd( hd( tl( X ) ) ) ), hd( tl( hd( tl( X ) ) ) ) ),
             tl( X )
            );
      X := cons( <N.N>, X );
      while( hd( X ) ) do
        if( hd( hd( tl( X ) ) ) ) then
          if( tl( hd( tl( X ) ) ) ) then
            if( hd( hd( hd( tl( X ) ) ) ) ) then
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           cons(
                             hd( hd( hd( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( hd( hd( tl( X ) ) ) ) ),
                               tl( hd( hd( tl( X ) ) ) )
                              )
                            ),
                           cons(
                             hd( hd( tl( hd( tl( X ) ) ) ) ),
                             cons(
                               tl( hd( tl( hd( tl( X ) ) ) ) ),
                               tl( tl( hd( tl( X ) ) ) )
                              )
                            )
                          ),
                         tl( tl( X ) )
                        )
                      )
              else
                X := cons( N, cons( N, tl( tl( X ) ) ) )

            else
              if( hd( tl( hd( tl( X ) ) ) ) ) then
                X := cons( N, cons( N, tl( tl( X ) ) ) )
              else
                X := cons(
                       <N.N>,
                       cons(
                         cons(
                           tl( hd( hd( tl( X ) ) ) ),
                           tl( tl( hd( tl( X ) ) ) )
                          ),
                         tl( tl( X ) )
                        )
                      )


          else
            X := cons( N, cons( N, tl( tl( X ) ) ) )

        else
          if( tl( hd( tl( X ) ) ) ) then
            X := cons( N, cons( N, tl( tl( X ) ) ) )
          else
            X := cons( N, cons( <N.N>, tl( tl( X ) ) ) )

        ;
      X := tl( X );
      if( hd( X ) ) then
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( tl( hd( hd( X ) ) ), tl( tl( hd( X ) ) ) ),
                 tl( X )
                )
              )
      else
        X := tl( X );
        X := cons(
               <N.N>,
               cons(
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) ),
                 cons( hd( tl( X ) ), tl( tl( tl( X ) ) ) )
                )
              )

    else
      X := cons( N, N )

  else
    X := cons( N, <N.N> )
  ;
X := tl( X )
```

data

```
static   <<<N.N>.N>.<<N.N>.N>>
dynamic  <<N.N>.<<<N.N>.N>.<N.<<<N.N>.N>.N>>>>
expected N
```

References

Beckman, L., Haraldson, A., Oskarson, O., & Sandewall, E. 1976. A Partial Evaluator and its Use as a Programming Tool,. *Artificial Intelligence*, **7**, 319–357.

Futamura, Yoshihiko. 1971. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Systems. Computers. Controls.*, **2**(5), 45–50.

Jones, Neil D. 1996. What Not to Do When Writing an Interpreter for Specialisation. *Pages 216–237 of:* Danvy, Olivier, Glück, Robert, & Thiemann, Peter (eds), *Partial Evaluation*, vol. 1110. Springer-Verlag.

Jones, Neil D. 1997. *Computability and Complexity from a Programming Perspective.* MIT Press.

Jones, Neil D., Sestoft, Peter, & Søndergaard, Harald. 1985. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. *Pages 124–140 of: RTA '85: Proceedings of the First International Conference on Rewriting Techniques and Applications.* London, UK: Springer-Verlag.

Jones, Neil D., Gomard, Carsten K., & Sestoft, Peter. 1993. *Partial Evaluation.* Prentice Hall International.

Kleene, S. C. 1952. Introduction to Metamathematics.

Lombardi, L. A., & Raphael, B. 1964. Lisp as the language for an incremental computer. *Pages 204–219 of:* Berkeley, E. C., & Bobrow, D. G. (eds), *The Programming Language Lisp.* MIT Press.

Rose, Eva. 1996 (July). *Characterizing Computation Models with a Constant Factor Time Hierachy.* Electronic.