Syntactic Currying:

yet another approach to

Partial Evaluation

N. Carsten Kehler Holst

July 17, 1989

Holst, Carsten Kehler and Danvy, Olivier. Partial Evaluation without a Partial Evaluator. 1989. in preparation. Follows

Abstract

Syntactic currying of a program is the transformation of one phase program into two phase programs. One phase programs takes all the input and produce the result, while a two phase program takes part of it input and produce a residual program as an intermediate result, when the residual program is applied to the rest of the input it computes the final result.

Syntactic currying is very close to partial evaluation. Just like there are two ways of running programs: interpreting, and compiling; there are two ways of performing partial evaluation. Until now people have concentrated solely on the interpretive style (which corresponds to the implementation of the S_n^m-theorem). This report introduce the compile style (which corresponds to the implementation of syntactic curry). It is shown that the compile style (or model) have some advantages for the interpretive style both considering giving a pedagogic explanation of partial evaluation, and when it comes to efficient implementation of partial evaluation.

The report gives a stepwise development of syntactic currying. Starting with with an implementation of syntactic currying based on evaluation over a domain of expressions. It is then realized that the inefficiency of this implementation, comes form the fact that it is an implementation of a dynamically typed language, with the binding time types static (early binding time), and dynamic (late binding time), where all operations are overloaded. Developing a type inference for binding time types and transforming the programs into statically typed versions, leads to an implementation which is statically typed and much more efficient. The type inference is very inspired by Nielson and Nielson [Nielson 86], and corresponds to the binding time analysis in [Bondorf 88].

The syntactic currying is implemented in Scheme using `extend-syntax` (macros). In the end the syntactic currying is compared with partial evaluation and our implementation is compared with other partial evaluation systems.

Keywords: Syntactic currying, partial evaluation, binding time analyses, type analysis, compilation, Scheme, Syntactic extensions.

Contents

Chapter 1

Introduction

This project was written during summer 1989 by Carsten Kehler Holst. It was supervised by Olivier Danvy, and carried out at DIKU (Department of Computer Science, University of Copenhagen). The project describes syntactic currying, and implements a syntactic curryer in two different ways. Syntactic currying is inspired by partial evaluation, in fact a syntactic curryer is a syntactically curried version of a partial evaluator.

This report contains a stepwise development of an efficient syntactic curryer. The syntactic curryer defined in this report is not fully correct with respect to termination properties, but when it terminates it gives the correct result. Syntactic currying can be seen as an approach to partial evaluation, and syntactic currying and partial evaluation have much in common. But the emphasize in thesis report is put on syntactic currying, and the description of, and comparison with, partial evaluation are postponed until chapter 6.

The main contributions of this project are the development of a syntactic curryer, and the development of a binding time type system, in which early binding times are subtypes of late binding times. Intuitively speaking one can always bind things later but not visa versa.

The introduction starts with an section that motivates syntactic currying, and give the intuition behind the concept. Then the prerequisites are stated and some of the notation used in the rest of the report are explained. Finally an outline of the report is given.

1.1 Motivation

Normally a program is applied to some data and yields a result. This is a one phase operation, all the data has to be supplied before any computation can take place. Some times it is beneficial to split a program into several phases, so part of the computation can be done in a prephase, *e.g.*, a general LR parser would generate the parse tables in the first phase and parse the text in the second phase. This is advantageous if the parser is applied to several input strings for each grammar. Even if the parser is only applied to one string it may turn out, that it is better to split the parsing in two phases, because the recognition algorithm that use parse tables is preferable to the one that use backtracking.

Syntactic currying is a technique that splits programs into phases. Let p be a program taking a pair of input data (s, d), such that $p(s, d) = \mathit{result}$. A syntactic curried version of p, p', is a program that takes the first part of input s (the static part of the input) and produce a residual program p_s as result, $(p\ s = p_s)$, the residual program will then compute the result given the rest of the input, d (the dynamic part of the input), $(p_s\ d = \mathit{result})$.

The main interest is not just syntactic currying, which is trivial, but efficient syntactic currying. Efficient in the sense that some of the original computation is done when the syntactically curried program is applied to the static input, which again means that the residual program p_s, applied to d have less computation to do, than p applied to both static and dynamic input, (s, d).

It is of course important that the splitting of the program into two phases can be described by a splitting of the input into two parts, since a syntactic curryer splits the program into two phases given a splitting of the input into a static and a dynamic part. A general LR parser has this property. The grammar is the part of input that in a natural way describes the first phase and the rest of the input, the input string, determines the second phase. On the other hand it would properly not make any sense to specialize the LR parser with respect to the input string, expecting as a result a program that recognize the grammars that accepts that particular input. We refer to [Dybkjaer 85], which describes the specialization of Earley's general context free parser with respect to various grammars.

If we assume that it is possible to implement syntactic currying efficient, it would be reasonable to make programs very general, and then specialize them with respect to certain applications. A typical area is database applications which would have to be specialized with respect to the configuration of the specific database.

In general the motivation is to split program into two phased programs, in order to achieve better performance in the second phase.

1.2 Prerequisites and notation

The LISP dialect Scheme [Rees 86] is used as example language throughout this report. The implementation of the syntactic curryer is done using the macro facility of Chez Scheme, `extend-syntax`. The reader is supposed to know Scheme or LISP, but it is not necessary to know about the macro facility `extend-syntax`. Neither is it necessary to know anything about partial evaluation, although such knowledge certainly would help in the understanding of the fine points in the project.

1.2.1 The Scheme subset used

This project does not handle full Scheme, only a first order subset containing: constants, basic operations, conditions, function calls, and recursive function definitions. We have chosen this subset, because it is the minimal subset of Scheme that demonstrates the ideas in syntactic currying. Handling the higher-order case would be beyond the scope of this report, likewise would global states, and side effects. Both these have been handled in [Bondorf 89b,Gomard 88, Jones 89b]. There have been made an extended version, of the system described in this report, that also works for higher-order programs.

Programming Languages A programming language is a function that assigns meaning to a text. A program in it self is just a sequence of characters, but when considered as a program in a particular language, it gets a meaning, typically some input-output function.

Definition A programming language is a partial function from the domain of program texts into the domain of computable functions. Let L be a programming language then L has the following type.

$$L : P \dot{\rightarrow} I \dot{\rightarrow} O$$

P is the set of L-programs and $(I \dot{\rightarrow} O)$ is the set of computable functions. We use the notation $\dot{\rightarrow}$ for computable partial functions.

The Type of a Program Text In computer science we work with representations of functions (programs), instead of functions. Some times this is very apparent, and this is one of these times. In this report we juggle around with programs all the time, so it will be nice with a notation for the type of a program text.

Definition Let p be an L-program. We denote the type of p as $\overline{t}$ where t is the type of the input-output function of p. $(L\ p : t \equiv p : \overline{t})$. With this notation the type of an interpreter, i, becomes $(i : \overline{(I \dot{\rightarrow} O)} \times I \dot{\rightarrow} O)$.

Expressions building Expressions It is often the case in syntactic currying that we constructs a expression that evaluates to an expression, e.g., an expression that evaluates to an if-expression could be `(list 'if` $e_0\ e_1\ e_2$`)`. In the rest of this report we use an underline to express that an expression is supposed, not to evaluate, but to build an expression. The building if-expression from above becomes $\underline{\texttt{(if}}\ e_0\ e_1\ e_2\underline{)}$

The semantics of the underline construction is the same as for the backquote in LISP. so all underlined lists, $\underline{(}\ e_1 \ldots e_n\underline{)}$, should be read as an expression that constructs a list, `(list` $e_1 \ldots e_n$`)`, and all underlined atoms, $\underline{\texttt{i-am-atom}}$, should be read as an expression that constructs atoms, *i.e.*, a quote, `(quote i-am-atom)`.

1.2.2 A Syntactic Curryer

Until now we have been talking about a syntactic curryer without stating precisely what it is. A syntactic curryer is a program, *mcx*[1], Which fulfills the following requirements.

A Syntactic Curryer *mcx*

$$L\ mcx \ :\ \overline{(S \times D \dot{\rightarrow} R)} \dot{\rightarrow} \overline{(S \dot{\rightarrow} \overline{(D \dot{\rightarrow} R)})}$$

$$L\ (L\ (L\ mcx\ p)\ s)\ d = L\ p\ (s, d)$$

The type variables S, D, and R, stands for Static input, Dynamic input, and Result. They are different names for some universal data domain (*e.g.*, lists, or strings). Looking at the type expression for *mcx* we find four function arrows "$\dot{\rightarrow}$", we use the following terms when we talk about these function applications. The first function arrow from the left represents application of the original function. The second represents syntactic currying (some times called translation). The third represents application of the syntactically curried program to the static data, it is called the static application. The result of the static application is a residual program. Finally, the last application is called a dynamic application. We shall often us the terms static time and dynamic time, instead of static application and dynamic application.

[1]The name *mcx* was suggested by Thomas P. Jensen along with the name *comix*. Both names are inspired by *mix* [Jones 85a], and the fact that, if *mix* is viewed as an interpreter, *mcx* should be seen as a compiler

Example The following construction shows how simple syntactic currying can be expressed using the newly defined notation, and the standard notation from other translation schemes. Syntactic variables are written using *italic*. The *s*, and *d* below belongs to the syntactic class of variables, and the *e* belongs to the syntactic class of Scheme expressions. We take the liberty to be fairly imprecise with respect to the definition of concepts like Variables and Expressions, because their precise definition is of less relevance in this project. The function that performs syntactic currying is called $\mathcal{C}$.

The simplest definition of Syntactic Currying

$s, d \in$ Variables
$e \in$ Expressions

$$\mathcal{C} \;:\; \overline{(S \times D \dot{\to} R)} \dot{\to} \overline{(S \dot{\to} \overline{(D \dot{\to} R)})}$$

$$\mathcal{C}[\![\, \texttt{(lambda (}s\ d\texttt{) }e\texttt{)} \,]\!] =$$
$$\texttt{(lambda (}s\texttt{) }\underline{\texttt{(lambda (}d\texttt{) (let ([}s\texttt{ (quote}}\ s\ \underline{\texttt{)]) }e\texttt{))}}\texttt{)}$$

Example:

$\mathcal{C}[\![$
```
(lambda (a b) (+ a b))
```
$]\!] =$
```
(lambda (a)
   (list 'lambda (list 'b)
            (list 'let (list (list 'a (list 'quote a)))
                     (list '+ 'a 'b))))
```

Notice how useful the underline notation is, in the translation of `(lambda (a b) (+ a b))`. The translation is a little longer that it need to be, but we have followed our definition of underline truthfully.

1.2.3 Symbolic Evaluation

When we talk about symbolic evaluation in this report we think of a particular evaluation strategy, that is developed in chapter 2. By symbolic evaluation we do not mean non numeric evaluation, but evaluation over a domain of expressions. Symbolic evaluation corresponds to various reduction strategies, since the result of symbolically evaluating an expression is an expression with the same meaning (value) as the original expression.

1.3 Outline

In this report we develop syntactic currying, using the techniques known from partial evaluation.

In chapter 2 it is shown how syntactic currying can be implemented using evaluation with free variables, or evaluation over a domain of expressions. We call that kind of evaluation, symbolic evaluation. That chapter develops an almost complete implementation of symbolic evaluation. The rest of the report can then be seen as an attempt to optimize that implementation.

The first optimization stems from the observation that symbolic evaluation can be seen as evaluation over a domain with two types, Static which represents early binding times, and Dynamic which represents late binding times, together these two types are called binding time types. In chapter 3 a type system is developed, and it is shown how symbolic evaluation of programs can be optimized by transforming the programs into programs that are statically typed, with respect to binding time types, before the symbolic evaluation. The optimization stems from the fact that all constructions in the programming language can be seen as overloaded, with respect to these types, and that the overloading can be solved if a type analysis is used.

The optimized version of symbolic evaluation is developed in chapter 4, where the implementation developed in chaptersymbolic-evaluation, is rephrased with the added assumption that the overloading of the language constructs have been solved.

In chapter 5 a concrete implementation of the statically typed symbolic evaluation is developed. This is done by annotating the program with type tags, and giving the annotated constructs a new meaning. The implementation of this new meaning is done with `extend-syntax`.

In chapter 6 *mcx* is compared with partial evaluation in general, and the MIX system in particular. Chapter 7 shows some experiments done with *mcx* and the efficiency of *mcx* is compared to that of *mix*. It is shown how syntactic currying of an interpreter yields a compiler. Chapter 8 concludes on the project as a whole.

Chapter 2

Symbolic Evaluation

This chapter contains a stepwise development of symbolic evaluation, which is one approach to syntactic currying. We use the term symbolic evaluation for an evaluation strategy where the value domain is expressions, and do not refer to non numeric computation. Syntactic currying is transformation of a program, p, into a program, p', such that, $(L\ p\ (s,d) = L\ (L\ p'\ s)\ d) \forall s,d \in S \times D$, where S, and D range over some reasonable domain.

Symbolic evaluation is not supposed to be the final implementation of syntactic currying in this project, only a step on the way. Therefore the aim in this chapter is not on efficiency, but on approachability. In later chapters we shall show how the principles in symbolic evaluation can be implemented efficiently.

At static time, the syntactically curried program is applied to the static data. At that time some of the variables will be bound and others will remain unbound (or free). In the introduction we saw how a function definition could be curried in a very simple way. The main objection to that solution was that no computation was done at static time, all the computation is left until dynamic time. It was our aim, that as much computation as possible should be performed at static time – leaving as little as possible to dynamic time.

In this chapter we shall develop an evaluation strategy, symbolic evaluation, that evaluates "as much as possible". The evaluation strategy corresponds in some sense to beta and delta reductions of lambda terms with free variables, or evaluation of an expression with unbound variables. The intuition behind the evaluation strategy is that subexpressions without unbound variables (*i.e.*, closed terms) are evaluated while subexpressions with unbound variables are left as they are. This effect can be achieved by letting the value domain be residual expressions. This means that the value of an expression when it is symbolically evaluated is a residual expression, with the same value as the original expression. This way an operation can be evaluated if all its subexpressions evaluates to constant expressions, otherwise it must be left as it is.

Definition of Symbolic Evaluation In symbolic evaluation all values are expressions. This means that symbolic evaluation of an expression yields an expression with the same value as the original expression. This can be illustrated as follows.

$$\mathcal{E}(\ \mathcal{SE}[\![\ e\ ]\!]\ \rho_{partial}\ )\ \rho = \mathcal{E}[\![\ e\ ]\!]\ \rho$$

Symbolic evaluation in a partial environment, $\rho_{partial}$, results in a residual expression. The partial environment, $\rho_{partial}$, is an environment where some of the variables are unbound. When the residual expression is evaluated it gives the same result as the original expression if evaluated in the complete environment, ρ.

In the next section we shall see how translation of an expression into an expression that performs symbolic evaluation can be achieved. Using the notation from above, an expression, e, is translated from "$\mathcal{SE}$" to "$\mathcal{E}$", *i.e.*, an expression, e, is translated into an expression, e', such that the meaning of e' under the normal semantic is the same as the meaning of e under the symbolic evaluation semantic. Call the translation function $\mathcal{SC}$ for syntactic currying. ρ_S is the environment that binds the static arguments, and ρ_D binds the dynamic arguments.

$$\mathcal{SC}[\![\ e\ ]\!] = e'$$
$$\mathcal{E}(\ \mathcal{E}[\![\ e'\ ]\!]\,\rho_S\ )\,\rho_D = \mathcal{E}[\![\ e\ ]\!]\,\rho$$

Thus $\mathcal{SC}$ performs syntactic currying, given that the environment, ρ, is split into a static part, ρ_S, and a dynamic part, ρ_D. The rest of this chapter develops $\mathcal{SC}$. This is done in a fairly operational style with much emphasize on how symbolic evaluation should be implemented.

2.1 The Simple Constructs

This section describes how to make syntactic currying of simple expressions like constants, variables, basic operations, and conditionals. The symbolic evaluation of these expressions are quite simple, and the syntactic currying is therefore also simple. The next section handle function definitions, and function calls.

Constants & Variables To make life simpler we demand that all constants are quoted. This is not normally the case in Scheme, where numbers and strings are self-evaluating. In this case all constants have the form `(quote` *v*`)`.

Variables are translated into variables using the identity function. It is then left over to the constructions that bind variables to values to make sure that they are bound to something sensible, *i.e.*, an expression.

A constant must evaluate to a constant. So the translation of a constant becomes a constant expression which value is a constant expression.

Symbolic Evaluation of Constants

$$\mathcal{SC}[\![\, x\,]\!] = x$$
$$\mathcal{SC}[\![\,\texttt{(quote } v\texttt{)}\,]\!] = \underline{\texttt{(quote } v\texttt{)}} \equiv \texttt{(quote (quote } v\texttt{))}$$

Basic Operations basic operations are a bit more involved. If all the arguments to the operation are constants (*i.e.*, static) the operation should be evaluated with the value of the arguments a arguments, otherwise we cannot do any better than building a residual operation. Of course it is some times possible to reduce the operations, e.g., `(car (cons` *a b*`))`, but in general this will not be the case.

Symbolic Evaluation of Basic Operations

```
$\mathcal{SC}[\![$ (O $e_1 \ldots e_n$) $]\!]$ = (let ([*arg$_1$* $\mathcal{SC}[\![ e_1 ]\!]$]...[*arg$_n$* $\mathcal{SC}[\![ e_n ]\!]$)])
                          (if (and (static? *arg$_1$*)...(static? *arg$_n$*))
                              (quote (O (de-ref *arg$_1$*)...(de-ref *arg$_n$*)))
                              (O *arg$_1$* ... *arg$_n$*)
```

In order to avoid repeated evaluations of the arguments a let-expression is inserted. Unfortunately this leads to some problems, the variable names *arg$_1$* to *arg$_n$* have to be invented, and we have to make sure that no name-clash appears with the names in the expressions e_1 to e_n, if the names generated equals names used in the source expression the semantics of the translation falls to the ground. We adopt the convention that in the equations variable names with stars around, like *name*, are unique. Inside the same equation *name*, and *name* are the same name, but besides that *name* is a unique name. Normally such a name will be generated using the Scheme built in function gensym, which builds a unique name, or by using hygienic macro expansion [etal 86].

The function static? checks if the argument is a static value, *i.e.*, a constant expression, and de-ref, de-reference an constant expression and returns the value of the constant expression. It is supposed that the standard function are not redefined by the source program. *I.e.*, and, if, *e.t.c.* have their usual meaning.

Conditionals An obvious choice would be to implement conditionals just like basic operations, but actually the meaning of the if-expression only depends on the condition. If the condition is static the conditional expression can be resolved, otherwise it must be left as it is.

Symbolic Evaluation of Conditionals

```
$\mathcal{SC}[\![$ (if c t e) $]\!]$ =
    (let ([*condition* $\mathcal{SC}[\![ c ]\!]$][*then* $\mathcal{SC}[\![ e ]\!]$][*else* $\mathcal{SC}[\![ e ]\!]$])
        (if (static? *condition*)
            (if (de-ref *condition*) *then* *else*)
            (if *condition* *then* *else*)))
```

2.2 Function Definitions, and Function Calls

Symbolic evaluation of function calls can be implemented in the same way as we implemented symbolic evaluation of basic operations. If one of the arguments are dynamic the call is left as it is, otherwise it is evaluated.

This is not the solution we want, since it would mean that recursive function that are only partly static would not be specialized at all. Instead we would like to specialize the function with respect to the static arguments and then make a residual function call where the specialized function is called with the dynamic arguments.

The function definition should be transformed into a definition of a specializing function. The specializing function should return a specialized function definition when applied to the static arguments.

A function call should should apply the translated function to the static arguments and build a residual application that apply the result of specializing the function with respect to the static arguments to the dynamic arguments.

We chose that a function call binds both the static and the dynamic arguments. The static arguments are bound to their value while the dynamic arguments are bound to variables. This way the definition of the specializing function have to build a function definition which have the dynamic arguments as formal parameters, and the residual expression obtained by symbolically evaluating the original body, as residual body.

If the arguments are static we chose to evaluate the function normally, just like with the basic operations. Below a solution that fulfills these requirements is shown.

Symbolic Evaluation of Function Calls, and Definitions (first)

```
$\mathcal{SC}[\![$ (letrec ([$n_1$ $d_1$]...[$n_n$ $d_n$]) $e$) $]\!]$ =
    (letrec ([$n_1$ $\mathcal{SC}[\![ d_1 ]\!]$]...[$n_n$ $\mathcal{SC}[\![ d_n ]\!]$]) $\mathcal{SC}[\![ e ]\!]$)

$\mathcal{SC}[\![$ (lambda ($v_1 \ldots v_n$) $e$) $]\!]$ =
    (lambda ($v_1 \ldots v_n$)
       (let ([*rbody* $\mathcal{SC}[\![ e ]\!]$])
       (if(and (static? *arg$_1$*)...(static? *arg$_n$*))
          *rbody*
          (lambda (extract-dynamic ($v_1 \ldots v_n$) *rbody*)))))

$\mathcal{SC}[\![$ ($f$ $e_1 \ldots e_n$) $]\!]$ =
    (let ([*arg$_1$* $\mathcal{SC}[\![ e_1 ]\!]$]...[*arg$_n$* $\mathcal{SC}[\![ e_n ]\!]$])
       (if(and (static? *arg$_1$*)...(static? *arg$_n$*))
          ($f$ *arg$_1$*...*arg$_n$*)
          (cons ($f$ (dynamic-to-var *arg$_1$*) ... (dynamic-to-var *arg$_n$*))
                (extract-dynamic (*arg$_1$*...*arg$_n$*)))))

where
    (define (extract-dynamic l)
       (append (map (lambda (d?) (if (static? d?) () d?)) l)))
    (define (dynamic-to-var s?)
       (if (static? s?) s? *new-var*))
```

Translating a letrec-expression yields a letrec, with translated versions of the definitions, and the body. The function call is translated into a "curried" function call. The translation of the function f is applied to the static arguments and place holders (variables) for the dynamic arguments. The translated version of f returns a lambda expression (a residual function definition), and a residual application of the residual function definition to the dynamic arguments, are build. The auxiliary function `extract-dynamic` takes a list of expressions and returns the sublist of dynamic expressions, in this case it returns the actual dynamic parameters.

When the function is called all dynamic expressions is replaced with variables, so when `(extract-dynamic ...)` is applied to the arguments of the function it returns a list of variables, which is used as the list of formal parameters in the residual function definition. A function definition (a lambda-expression) is translated into a function definition, which yields a residual function definition, when it is applied to the static arguments and variables for the dynamic arguments. The residual function definition takes only the dynamic arguments as formal parameters.

The solution above is correct, when it terminates, unfortunately there is a very broad class of function definitions for which it goes into an infinite loop at static time. The example below shows what happens. The append function is repeatedly called with static second argument equal to `'(3 4)` and dynamic first argument. The condition which control the recursion is dynamic, so both branches are symbolically evaluated and the symbolic evaluation goes into an infinite loop.

Repeated specializations with respect to the same values

```
Let p = (letrec ([append (lambda (x y)
                           (if (null? x) y
                               (cons (car x) (append (cdr x) y))))])
          (append v w))
```

then evaluation of $\mathcal{SC}[\![p]\!]$ in an environment where `v` is bound to the variable `z` and `w` is bound to the static values `'(3 4)` "yields" the following infinite result

```
((lambda (*var1*)
    (if (null *var1*) '(3 4)
        ((lambda (*var2)
            (if (null? *var2*) infinite unfolding))
         (cdr *var1*))))
 z)
```

A solution to this problem would be to record the specialization of a function, and then refer to the residual function definition by name instead of inserting the function definition in the residual function call. This way it would only be necessary to specialize a function once with respect to each set of static arguments, and the infinite specialization exhibited in the example would not occur.

We need an accumulator in which the specialized functions (that are the residual function definitions) are collected. A function call should check if there have already been a call to the same function with the same set of static arguments, and if that is the case just leave a call to the specialized version of that function, otherwise it should specialize the function and record the specialized version of the function in the accumulator, and generate a call to it. Of course the residual function definitions have to be put somewhere. The natural place would in a residual letrec generated by the letrec expression that defined the function originally. When the scope of that letrec-expression is left no more calls to the functions defined there can occur, and thus no more specialized functions.

In the example above the result would be a letrec with one specialized version of append.

2.2.1 Implementation of an Accumulator

There are three natural possibilities: an accumulator for each function, an accumulator for each letrec, or a global accumulator for the whole program. Global means complex, local means simple, in the sense that a global accumulator would have to record: which letrec, which function, and which set of static arguments, that gave rise to a particular residual function definition, while the accumulator that is local to the function, only would have to record the value of the static arguments, for each residual function definition.

Where to define the accumulator(s) is mostly a matter of taste. We chose to have one accumulator for each letrec, as this seems to be the easiest solution.

The accumulator could either be implemented as a state (variable), that is destructively updated using Schemes assignments, or be passed around using continuation passing style. We chose the solution with a state, that is destructively updated. Again it is very much a matter of taste. It should be noticed that the solution is independent of the order of evaluation (assuming call-by-value). A syntactic curryer that pass the accumulator around using continuation passing style is described in [Holst 89b].

When a function is called it should check if it has been specialized with respect to the current set of static arguments. If that is the case it should build a residual function call using the residual function name recorded in the accumulator. Otherwise it must record in the accumulator, that the function is being specialized with respect to the current set of static arguments, specialize the function, and build the residual function call. It is necessary to record that the function is being specialized, because we specialize functions using a depth-first strategy. This means, that when we during specialization reach a function call, we suspend the actual specialization and start to specialize the new function. This may lead into an infinite loop just like in the example before unless we have recorded that the function are being specialized with respect to that set of static arguments and what name the residual function will get.

When a function have been specialized the residual function definition should be recorded in the accumulator. Finally when we leave a letrec the collected residual function definitions should be put in a residual letrec.

Following the idea with the accumulator the function definitions and calls are translated as described below.

Symbolic Evaluation of Function Calls, and Definitions (second)

```
    $\mathcal{SC}$⟦(letrec ($d_1 \ldots d_n$) $e$)⟧ =
        (let ([*acc* (empty-acc)])
           (letrec ($\mathcal{SC}$⟦$d_1$⟧...$\mathcal{SC}$⟦$d_n$⟧)
              (let ([*rexp* $\mathcal{SC}$⟦$e$⟧]) (extract *acc* *rexp*))))

    $\mathcal{SC}$⟦[$n$ (lambda ($v_1 \ldots v_n$) $e$)]⟧ =
        [$n$ (lambda ($v_1 \ldots v_n$)
              (if(and (static? $v_1$)...(static? $v_n$))
                 $\mathcal{SC}$⟦$e$⟧
                 (let ([*done* (called? *acc* $\underline{n}$ $\underline{(}v_1 \ldots v_n\underline{)}$)])
                    (if (null? *done*)
                        (let ([*rdef* $\underline{\text{(lambda}}$ (extract-dynamic $\underline{(}v_1 \ldots v_n\underline{)}$) $\mathcal{SC}$⟦$e$⟧$\underline{)}$])
                           (specialize! *acc* $\underline{n}$ $\underline{(}v_1 \ldots v_n\underline{)}$ *rdef*))
                        *done*))))]

    $\mathcal{SC}$⟦($f$ $e_1 \ldots e_n$)⟧ =
        (let ([*arg$_1$* $\mathcal{SC}$⟦$e_1$⟧]...[*arg$_n$* $\mathcal{SC}$⟦$e_n$⟧])
           (if(and (static? *arg$_1$*)...(static? *arg$_n$*))
              ($f$ *arg$_1$*...*arg$_n$*)
              (cons ($f$ (dynamic-to-var *arg$_1$*) ... (dynamic-to-var *arg$_n$*))
                    (extract-dynamic $\underline{(}$*arg$_1$*...*arg$_n$*$\underline{)}$))))

    where
        (define (extract-dynamic l)
           (append (map (lambda (d?) (if (static? d?) () d?)) l)))
        (define (dynamic-to-var s?)
           (if (static? s?) s? *new-var*))
```

The precise definition of the functions `empty-acc`, `called?`, `specialize!`, and `extract!`, is given in appendix A. Here we shall only state the intuition behind them.

To make the definition of letrec more readable it has been split into two parts: the body part, and the definition part. Unfortunately this means that we diverge from one of our rules, namely that variables surrounded by "*" are unique to the particular application of the rule. The `*acc*` that occurs in the function definition is the same as the one that occurs in the translation of the letrec in which the function definition occurs. Each letrec have its own `*acc*` which is only referred to by the functions defined by that letrec.

The letrec construction is translated into a let-expression that introduce the accumulator `*acc*`. The recursive functions are then defined in the scope of `*acc*`. The sole reason for the innermost let-expression, is that we want the body evaluated before we generate the residual letrec-expression. The reason for this, is that all functions that are specialized during the evaluation of the body have to be put into the accumulator before the value of the accumulator is used to build the residual letrec. The function `extract!` builds a letrec expression if the accumulator is nonempty, otherwise it just return the residual body.

A function call is translated like before. With the difference that a call to a function, f, returns a residual function name instead of a residual function definition.

A function definition is more involved than before. First it is checked if the function n has been called with this set of static arguments before. If that is the case `*done*` is set to the name of the residual function. Otherwise `*done*` is set to `nil`, and it is noted in the accumulator that the function have been called with this set of static arguments now. The function **`called?`** takes care of this. Then **`*done*`** is checked. If it is a function name, different from **`nil`**, that function name is returned. If **`*done*`** is **`nil`**, we specialize the function, this may give rise to other specializations, and put the specialized function into the accumulator using **`specialize!`**.

Chapter 3

Type System

The last section developed symbolic evaluation, as an approach to syntactic currying. In this chapter various optimizations to symbolic evaluation are proposed. Most important, a types inference which solves the overloading of the operators is developed.

Symbolic evaluation goes on in a world with two types, static, and dynamic. Symbolic evaluation is dynamically typed (the types can first be determined at static time), and all constructs are overloaded. This chapter develops a type inference that makes it possible to transform dynamically typed programs into statically typed programs, thus makes it possible to solve the overloading at translation time (when the syntactic currying occurs)

This type analysis corresponds almost precisely to the binding time analysis from [Jones 85a, Bondorf 88]. The idea of using type inference to describe binding times analysis was inspired by [Nielson 86], and first described in [Holst 89a].

3.1 Analysis of Symbolic Evaluation of a car Operation

Consider the symbolic evaluation of a car-expression, below we have shown what a car-expression is translated into in order to work as an symbolic evaluating car-expression.

When car is applied to the expression e it evaluates e using symbolic evaluation. The result of that evaluation is an expression, maybe a constant. Then it checks if the result of evaluating the argument, $*arg_1*$, is a constant. If $*arg_1*$ is a constant, it de-ref the constant (gets the value of the constant), takes car of that value and lift the result of the car operation back into a constant expression. Otherwise, if $*arg_1$ is not a constant, it builds a car operation, which on dynamic time will take car of the argument.

A symbolic evaluating car-expression

```
𝒯⟦ (car e) ⟧ = (let ([*arg₁* 𝒯⟦ e ⟧])
                    (if (static? *arg₁*)
                        (quote (car (de-ref *arg₁*)))
                        (car *arg₁*)))
```

The amount of work is overwhelming compared to the work normally executed by an car-expression. The reason for this is that the car expression is overloaded, its meaning depend on the "type" of its argument. If the argument is static it evaluates, otherwise it build a residual expression. Even worse when the argument is static it has to `de-ref` the constant expression take the car of it and then lift it back into a constant expression. Both these problems could be solved if it were possible to determine the "type" of its argument before the translation and then use the type information to solve the overloading.

Let T be a type system with the two types static (abbreviated S), and dynamic (abbreviated D). Let the type static denote constants or (constant expressions), and dynamic all expressions. Dynamic have deliberately been chosen to describes all expressions, because then static becomes a subtype of dynamic, just like Int is a subtype of Real.

Assume for a moment that we use ordinary values for static values (without a quote), and we for each operator knows if its arguments will be static or dynamic. Then a car-expression can be translated as follows.

Definitions of static and dynamic car operations

$\mathcal{T}[\![$ `(car`$_{S\rightarrow S}$ e`)` $]\!] \Rightarrow$ `(car` $\mathcal{S}[\![e]\!]$`)`

$\mathcal{T}[\![$ `(car`$_{D\rightarrow D}$ e`)` $]\!] \Rightarrow$ $\underline{\texttt{(car}}$ $\mathcal{D}[\![e]\!]\underline{\texttt{)}}$

This code is much preferable compared to the first solution. A static car operation is translated into a car operation and will evaluate as a car operation; if the arguments to an operation are know at static time that operation can be executed normally. On the other hand if the arguments are dynamic it is translated into an operation which when executed builds a car expression; if the value of the arguments are unknown we must build an a piece of code which performs the operation at dynamic time.

The other constructs in Scheme can be given types in the same way. For example the meaning of an if-expression with type $S \times D \times D \rightarrow D$ would be translated into the following .

Translation of if-expression with type $S \times D \times D \rightarrow D$

$\mathcal{T}[\![$ `(if`$_{S\times D\times D\rightarrow D}$ s $d1$ $d2$`)` $]\!]$ = `(if` s $d1$ $d2$`)`

3.2 The type System T

The type system T has two types static (S), and dynamic (D), where static is a subtype of dynamic. In general these types must be determined dynamically, *i.e.*, at run time. A typical example is a conditional where the then branch have type S and the else branch have type D. Such a conditional might have results of type S in some cases and of type D in other.

The type system T

$T = \{S, D\}$, and $S \lhd D$

As we already saw in the definition of symbolic evaluation of car the meaning of the operators and constructs depend on their types. Take as example cons, which can have all the following four types.

$$S \times S \;\rightarrow\; S$$

$$\begin{aligned} S \times D &\rightarrow D \\ D \times S &\rightarrow D \\ D \times D &\rightarrow D \end{aligned}$$

If cons has type $S \times S \rightarrow S$ it have the ordinary meaning. It takes two ordinary values and produce the pair of them. Likewise if cons had type $D \times D \rightarrow D$ the arguments values are residual expressions and cons should simply build a residual cons operation with the values of its arguments as arguments. The meaning of the cons operation for the last two types is roughly the same. If a cons operation has type $D \times S \rightarrow D$ it takes the value of its second argument and lift it to a constant expression, then it proceeds as if it had type $D \times D \rightarrow D$.

The type of the cons operation is not a type in our type system T, because we only consider a first order language, and values therefore only have types in the range S, D.

The problem with static typing can be solved using the sum domain $(S + D)$, but as we saw in the previous example this is rather inefficient. An other solution would be to make the program statically typed. As each occurrence of an operator or construction in a statically typed language have one and only one type the overloading can be solved statically.

One should not be confused over the term statically typed. It means that the types can be determined before the program is syntactically curried. So do not confuse statically typing with static time. If the types were determined at static time it would be dynamic typing.

3.3 A Static Type System

We go through all the constructions in the Scheme subset one by one. Finally we put the whole thing together and strengthen our type system a bit.

The type environment Let τ be a type environment, mapping variable names to either static or dynamic, and mapping function names to descriptions of the types of their arguments, and the type of their result.

τ : Names $\rightarrow$ (Value-Type + Function-Type)

where

Value-Type = S or D and
Function-Type = Value-Type* $\times$ Value-Type

3.4 Type conversion

It will often be the case that we have an expression of type S in a place where we would like an expression of type D. S is a subtype of D so we can safely convert (lift) the type of the expression from S to D. Assume that we have an operator which sole mission is to lift the type of the argument from static to dynamic. The type scheme for such an operator is as follow.

$$\frac{\tau \vdash e : S}{\tau \vdash \texttt{(lift } e\texttt{)} : D}$$

3.5 Constants & Basic operations

A constant is always static.

$$\tau \vdash \texttt{(quote } v\texttt{)} : S$$

If one of the arguments to a basic function is of type D then the result of the operation should be of type D. On the other hand if all arguments are of type S the result of the operation should be of type S.

$$\frac{\tau \vdash e_1 : D \vee \ldots \vee \tau \vdash e_n : D}{\tau \vdash \texttt{(}O\ e_1 \ldots e_n\texttt{)} : D} \qquad \frac{\tau \vdash e_1 : S \wedge \ldots \wedge \tau \vdash e_n : S}{\tau \vdash \texttt{(}O\ e_1 \ldots e_n\texttt{)} : S}$$

3.6 Conditionals

Conditionals do not differ much from the basic operations. Only if both the condition and the two branches are of type S will the result be of type S. The conditional should have the same type every time independent of the values of the condition so even if the condition is static and one of the branches is of type S the conditional should be of type d if the other branch is of type D.

$$\frac{\tau \vdash e_1 : D \vee \ldots \vee \tau \vdash e_3 : D}{\tau \vdash \texttt{(if } e_1\ e_2\ e_3\texttt{)} : D} \qquad \frac{\tau \vdash e_1 : S \wedge \ldots \wedge \tau \vdash e_3 : S}{\tau \vdash \texttt{(if } e_1\ e_2\ e_3\texttt{)} : S}$$

3.7 Letrec, Function calls, and Variables

Function calls differ from basic operations, because we want each occurrence of a function to have the same type. The reason for this is that we will translate the function according to its type, and if the function is used with several different types we would have to make several versions of the function, one for each type.

$$\frac{\tau(v) = t}{\tau \vdash v : t}$$

$$\frac{\begin{array}{c}\tau(f) = t_1 \times \ldots \times t_n \rightarrow t \\ \tau \vdash e_1 : t_1, \ldots, \tau \vdash e_n : t_n\end{array}}{\tau \vdash \texttt{(}f\ e_1 \ldots e_n\texttt{)} : t}$$

$$\frac{\begin{array}{c}\tau' = \tau[f_1 \mapsto t_{11} \times \ldots \times t_{1n_1} \rightarrow t_1, \ldots, f_k \mapsto t_{k1} \times \ldots \times t_{kn_k} \rightarrow t_k] \\ \tau'[x_{11} \mapsto t_{11}, \ldots, x_{1n_1} \mapsto t_{1n_1}] \vdash e_1 : t_1, \ldots \\ \tau'[x_{k1} \mapsto t_{k1}, \ldots, x_{kn_k} \mapsto t_{kn_k}] \vdash e_k : t_k \\ \tau' \vdash e : t\end{array}}{\begin{array}{ll}\tau \vdash \texttt{(letrec (} & \texttt{[}f_1\ \texttt{(lambda (}x_{11}, \ldots, x_{1n_1}\texttt{)}\ e_1\texttt{)]}, \ldots \\ & \texttt{[}f_k\ \texttt{(lambda (}x_{k1}, \ldots, x_{kn_k}\texttt{)}\ e_k\texttt{)])}\ e\texttt{)} : t\end{array}}$$

3.8 Some Optimizations to the Type Scheme

As we saw earlier a basic operation with n arguments have 2^n different types. These 2^n types can be collapsed into two principal types. The type where all arguments are static and the result is static, and the type where all arguments are dynamic and the result is dynamic. The reason for this is, that when the result has to be dynamic the static arguments have to be lifted to dynamic. This lifting can either be a part of semantic of the basic operation (like $\texttt{cons}_{S\times D\rightarrow D}$ which would have to lift its first argument to dynamic and then create an cons operation, as illustrated by the following semantic expression $\underline{\texttt{(cons (quote}}\ \underline{\textit{arg-1})}\ \textit{arg-2}\underline{)})$. So we shall restrict our type scheme in such a way that all the arguments of a basic operation have the same type.

With the same argumentation as for the basic operations we restrict our type system such that the two branches of a conditional have the same type. It will not be reasonable to restrict it further, because even when the two branches are dynamic the if-expression could be evaluated if the condition is static.

With these two optimizations we can state the type inference rules. The first rule is a little awkward having the sole mission to introduce the type environment τ. The rest of the rules describes the type restrictions added by our annotations.

Type Inference Rules for Scheme

$$\frac{\tau = [x_1 \mapsto S, \ldots, x_n \mapsto S, y_1 \mapsto D, \ldots, y_n \mapsto D] \qquad \tau \vdash e : D}{\texttt{(lambda-c}\ (x_1 \ldots x_n)\ (y_1 \ldots y_m)\ e\texttt{)}\ \text{is well typed}}$$

$$\frac{\tau(v) = t}{\tau \vdash v : t} \qquad \tau \vdash \texttt{(quote}\ v\texttt{)} : S \qquad \frac{\tau \vdash e : S}{\tau \vdash \texttt{(lift}\ e\texttt{)} : D}$$

$$\frac{\tau \vdash e_1 : D, \ldots, \tau \vdash e_n : D}{\tau \vdash (O\ e_1 \ldots e_n) : D} \qquad \frac{\tau \vdash e_1 : S, \ldots, \tau \vdash e_n : S}{\tau \vdash (O\ e_1 \ldots e_n) : S}$$

$$\frac{\tau \vdash e_1 : D, \ldots, \tau \vdash e_3 : D}{\tau \vdash \texttt{(if}\ e_1\ e_2\ e_3\texttt{)} : D} \qquad \frac{\tau \vdash e_1 : S, \ldots, \tau \vdash e_3 : S}{\tau \vdash \texttt{(if}\ e_1\ e_2\ e_3\texttt{)} : S} \qquad \frac{\tau \vdash e_1 : S, \tau \vdash e_2 : D, \tau \vdash e_3 : D}{\tau \vdash \texttt{(if}\ e_1\ e_2\ e_3\texttt{)} : D}$$

$$\frac{\tau(f) = (t_1 \times \ldots \times t_n) \times t \qquad \tau \vdash e_1 : t_1, \ldots, \tau \vdash e_n : t_n}{\tau \vdash (f\ e_1 \ldots e_n) : t}$$

$$\frac{\begin{array}{c} \tau' = \tau[f_1 \mapsto (t_{11} \times \ldots \times t_{1n_1}) \times t_1, \ldots, f_k \mapsto (t_{k1} \times \ldots \times t_{kn_k}) \times t_k] \\ \tau'[x_{11} \mapsto t_{11}, \ldots, x_{1n_1} \mapsto t_{1n_1}] \vdash e_1 : t_1, \ldots \\ \tau'[x_{k1} \mapsto t_{k1}, \ldots, x_{kn_k} \mapsto t_{kn_k}] \vdash e_k : t_k \\ \tau' \vdash e : t \end{array}}{\begin{array}{l} \tau \vdash \texttt{(letrec (}\ \ \texttt{[}f_1\ \texttt{(lambda}\ (x_{11}, \ldots, x_{1n_1})\ e_1\texttt{)]}, \ldots \\ \qquad\qquad\qquad\ \texttt{[}f_k\ \texttt{(lambda}\ (x_{k1}, \ldots, x_{kn_k})\ e_k\texttt{)])}\ e\texttt{)} : t \end{array}}$$

This ends our introduction of types. It can be seen how all function can be transformed into well typed functions by insertion of lift operators at various places in the program. It should be noticed that a program can be well typed in several different ways. The most extreme typing of a program makes everything dynamic, the only exception are the constants which will have to be lifted to dynamic.

Taking this viewpoint binding time analysis as described in [Jones 85a] can be described as a type analysis.

Chapter 4

Symbolic Evaluation with Static Types: An efficient approach to Syntactic Currying

In chapter 2 we developed symbolic evaluation. In chapter 3 we noticed that symbolic evaluation was dynamically typed with the binding time types: static, and dynamic. Furthermore we developed a type inference such that we could transform dynamically typed program into statically typed programs.

This chapter rephrase the implementation of symbolic evaluation developed in chapter 2 under the assumption that the types are static and the overloading have been solved. In addition to that, an optimization: call unfolding, which avoids the expensive specializing function call, is described.

A warning: do not confuse statically typing with static time. Static typing occurs when the program is syntactically curried, and means that the type of the operators are independent of the actual value of the input. Dynamic typing occurs at static time, which corresponds to runtime.

4.1 The Completely Static Constructs

Constructs with type $(\ldots \rightarrow S)$ have only completely static subparts. This is intuitively clear and can be proven, using induction on the type scheme. The meaning of static constructions at static time is the ordinary meaning of these constructions. So the translation scheme for static constructions is the identity.

Translation Scheme for Completely Static Constructions

$$\mathcal{SC}_S[\![\,\texttt{(quote } v\texttt{)}\,]\!] = \texttt{(quote } v\texttt{)}$$
$$\mathcal{SC}_S[\![\,\texttt{(}O_{\ldots\to S}\ e_1\ldots e_n\texttt{)}\,]\!] = \texttt{(}O\ \mathcal{SC}_S[\![\,e_1\,]\!]\ldots\mathcal{SC}_S[\![\,e_n\,]\!]\texttt{)}$$
$$\mathcal{SC}_S[\![\,\texttt{(if}_{\ldots\to S}\ e_0\ e_1\ e_2\texttt{)}\,]\!] = \texttt{(if } \mathcal{SC}_S[\![\,e_0\,]\!]\ \mathcal{SC}_S[\![\,e_1\,]\!]\ \mathcal{SC}_S[\![\,e_2\,]\!]\texttt{)}$$
$$\mathcal{SC}_S[\![\,\texttt{(}f\ e_1\ldots e_n\texttt{)}\,]\!] = \texttt{(}f\ \mathcal{SC}_S[\![\,e_1\,]\!]\ldots\mathcal{SC}_S[\![\,e_n\,]\!]\texttt{)}$$
$$\mathcal{SC}_S[\![\,\texttt{(letrec (}d_1\ldots d_n\texttt{) } e\texttt{)}\,]\!] = \\ \quad \texttt{(letrec (}\mathcal{SC}_S[\![\,d_1\,]\!]\ldots\mathcal{SC}_S[\![\,d_n\,]\!]\texttt{) } \mathcal{SC}_S[\![\,e\,]\!]\texttt{)}$$
$$\mathcal{SC}_S[\![\,\texttt{[}n\ \texttt{(lambda}_{\ldots\to S}\ \texttt{(}v_1\ldots v_n\texttt{) } e\texttt{)]}\,]\!] = \\ \quad \texttt{[}n\ \texttt{(lambda (}v_1\ldots v_n\texttt{) } \mathcal{SC}_S[\![\,e\,]\!]\texttt{)]}$$

In fact the recursive application of the translation scheme $\mathcal{SC}_S$ is not necessary, as it is the identity. But we have included it here to point out that this technique could also be used if the source and target language were different.

4.2 The Dynamic Constructs

The dynamic constructs are almost as easy as the static constructs. All and all, it is just a transformation of the symbolic evaluation scheme. There is one new construct: the type conversion function `lift`. Type conversion boils down to lifting a value to a constant expression. This is easy in the first-order case, just add a quote. In the higher-order case, which we are not handling, there would be a problem: how do one lift a functional value into an expression. Notice that there are two versions of the if-expression in the translation scheme below: one where the condition is static, and one where the condition is dynamic. The case with the static condition and the completely static case were handled in the same way in the symbolic evaluation semantics, here we have chosen to split the things in two, in order to emphasize that the result of the two expressions have different types. Note also that there are no constant-expressions — constant-expressions cannot be of type dynamic.

Translation Scheme for Dynamic Constructions

$$\mathcal{SC}_{\mathcal{D}}[\![\,\texttt{(lift } e\texttt{)}\,]\!] = \underline{\texttt{(quote}}\ \mathcal{SC}_S[\![\,e\,]\!]\underline{\texttt{)}}$$
$$\mathcal{SC}_{\mathcal{D}}[\![\,\texttt{(}O_{\ldots\to D}\ e_1\ldots e_n\texttt{)}\,]\!] = \underline{\texttt{(}O}\ \mathcal{SC}_{\mathcal{D}}[\![\,e_1\,]\!]\ldots\mathcal{SC}_{\mathcal{D}}[\![\,e_n\,]\!]\underline{\texttt{)}}$$
$$\mathcal{SC}_{\mathcal{D}}[\![\,\texttt{(if}_{D\times D\times D\to D}\ e_0\ e_1\ e_2\texttt{)}\,]\!] = \underline{\texttt{(if}}\ \mathcal{SC}_{\mathcal{D}}[\![\,e_0\,]\!]\ \mathcal{SC}_{\mathcal{D}}[\![\,e_1\,]\!]\ \mathcal{SC}_{\mathcal{D}}[\![\,e_2\,]\!]\underline{\texttt{)}}$$
$$\mathcal{SC}_{\mathcal{D}}[\![\,\texttt{(if}_{S\times D\times D\to D}\ e_0\ e_1\ e_2\texttt{)}\,]\!] = \texttt{(if } \mathcal{SC}_S[\![\,e_0\,]\!]\ \mathcal{SC}_{\mathcal{D}}[\![\,e_1\,]\!]\ \mathcal{SC}_{\mathcal{D}}[\![\,e_2\,]\!]\underline{\texttt{)}}$$

4.3 Function Calls and Function Definitions

The function calls differ from the basic operations in that they have mixed argument types. This means that the compile time actions are a bit more involved than in the translation of basic operations. This again means that more work can be moved from the static time to compile time, which is good, as the program only will be compiled once, but specialized with respect to static data several times. On the other hand the treatment of the accumulator will still be at static time, so all in all, function calls are rather expensive at static time.

Dynamic Function Calls and Definitions

```
$\mathcal{SC}_\mathcal{D}[\![$ (letrec ($d_1 \ldots d_n$) $e$) $]\!]$ =
    (let ([*acc* (empty-acc)])
        (letrec ($\mathcal{SC}_\mathcal{D}[\![ d_1 ]\!] \ldots \mathcal{SC}_\mathcal{D}[\![ d_n ]\!]$)
            (let ([*rexp* $\mathcal{SC}_\mathcal{D}[\![ e ]\!]$]) (extract *acc* *rexp*))))

$\mathcal{SC}_\mathcal{D}[\![$ ($f_{t_1 \times \ldots \times t_n \to t}$ $e_1 \ldots e_n$) $]\!]$ =
    $\underline{(}$($f$ $x_1 \ldots x_n$) $\mathcal{SC}_\mathcal{D}[\![ e_{d_1} ]\!] \ldots \mathcal{SC}_\mathcal{D}[\![ e_{d_m} ]\!]\underline{)}$
where
    $x_i =$ (make-sym), if $t_i = D$
    $x_i = \mathcal{SC}_\mathcal{S}[\![ e_i ]\!]$,    if $t_i = D$
    $t_{d_1}, \ldots, t_{d_m} = D$ so $e_{d_1}, \ldots, e_{d_m}$ is the dynamic arguments

$\mathcal{SC}_\mathcal{D}[\![$ [$n$ (lambda$_{t_1 \times \ldots \times t_n \to t}$ ($v_1 \ldots v_n$) $e$)] $]\!]$ =
    [$n$ (lambda ($v_1 \ldots v_n$)
            (let ([*done* (called? *acc* $\underline{n}$ $\underline{(}v_{s_1} \ldots v_{s_m}\underline{)}$)])
                (if (null? *done*)
                    (let ([*rdef* $\underline{\texttt{(lambda (}}$ $v_{d_1} \ldots v_{d_k}\underline{)}$ $\mathcal{SC}[\![ e ]\!]\underline{)}$])
                        (specialize! *acc* $\underline{n}$ $\underline{(}v_{s_1} \ldots v_{s_m}\underline{)}$ *rdef*))
                    *done*))))]
where
    $t_{s_1}, \ldots, t_{s_m} = S$ which means that $v_{s_1}, \ldots, v_{s_m}$ is the static variables, and
    $t_{d_1}, \ldots, t_{d_k} = D$ which means that $v_{s_1}, \ldots, v_{s_m}$ is the dynamic variables
```

This description is a little bit simpler that the one given in the chapter about symbolic evaluation, because there is no test on whether the arguments are static of dynamic. there is only one new function, `make-sym`, which corresponds to, `gensym`, and generates a unique variable.

4.4 Call Unfolding: An Optimization that removes the Expensive Function Specialization

Unfolding a function call corresponds to beta reduction. The call is replaced by the body of the called function in which the formal parameters are substituted for the actual parameters.

In the following we shall motivate why call unfolding is such a great improvement. Besides being a great improvement call unfolding has the drawback that it introduce the possibility of infinite unfolding. Since we leave it to the user to annotate which calls she want to unfold it will be her responsibility to avoid infinite unfolding.

There are two sides of inefficiency of function specialization. At static time the function specialization is time consuming, and at dynamic time the many function calls which comes from the many specialized function is often not what we want. Consider the following example where the function append is specialized with respect to the known first argument `'(1 2)`.

Specialization of append

Let p =
```
(letrec ([append (lambda (x y)
                    (if (null? x) y
                        (cons (car x) (append (cdr x) y))))])
        (append v w))
```

evaluating $\mathcal{SC}[\![p]\!]$ in an environment where v is bound to (1 2) and w is bound to the variable z gives the following result.

```
(letrec
   ([app1 (lambda (y) (cons '1 (app2 y)))]
    [app2 (lambda (y) (cons '2 (app3 y)))]
    [app3 (lambda (y) y)])
   (app1 z))
```

Instead we would like the following result, in which all function calls have been unfolded.

```
(cons '1 (cons '2 z))
```

Specializing a function call is time consuming at static time (or specialization time), because we have to make one or two lookups in the accumulator to see if we have specialized the function previously, further more it is expensive to build a function definition and a call. At dynamic time the specialized function will have to be called, this is expensive, not because function calls are specially expensive, but they are more expensive than if they were not there.

Unfolding the function calls helps on both. At static time unfolding a function call corresponds to evaluation the body of the function in the symbolic evaluation semantics (with, or without types) with the arguments bound to the actual parameters. This is not so much different from how it was done before. Instead of binding the dynamic variable to a fresh variable we bind it to the actual parameter. This way we avoid the lookup in the accumulator, and we need not build function declarations, and function applications. At dynamic time there are no function calls, so the resulting residual programs will also be faster.

In general it is not safe to unfold all calls. This would lead into the same trap as the first implementation of function calls we made chapter 2 – infinite unfolding. This is not surprising since the the collecting function specialization was invented to avoid infinite specialization. Another danger is, that the termination properties of the program may change. Call unfolding corresponds to call-by-name, thus there is a risk that the resulting program terminates more often than the original. This risk is not considered to be very dangerous.

The following rules describe how unfolding calls are to be compiled. It can almost be deduced from the simpler implementation that the unfolding calls must be more effective. We shall not consider how it is determined which calls to unfold, but leave it for the user to annotate the program with information that solves this problem. Interested should read [Sestoft 88, Bondorf 89b].

Definition of Unfolding Calls

$$\mathcal{SC}_{\mathcal{D}}[\![\, (f^{u}_{t_1 \times \ldots \times t_n \to t}\ e_1 \ldots e_n)\,]\!] = (f\ x_1 \ldots x_n)$$

where

$x_i = \mathcal{SC}_{\mathcal{D}}[\![\, e_i \,]\!]$, if $t_i = D$
$x_i = \mathcal{SC}_{\mathcal{S}}[\![\, e_i \,]\!]$, if $t_i = S$

$$\mathcal{SC}_{\mathcal{D}}[\![\, [n\ (\text{lambda}^{u}_{t_1 \times \ldots \times t_n \to t}\ (v_1 \ldots v_n)\ e)]\,]\!] = [n\ (\text{lambda}\ (v_1 \ldots v_n)\ \mathcal{SC}[\![\, e \,]\!])]$$

Chapter 5

Implementation

In this chapter we develop a concrete implementation of syntactic currying. The preceding chapters have developed syntactic currying in details, so this chapter will concentrate on how to implement it in Scheme.

We have chosen to implement syntactic currying using syntactic extensions. The reason for this is, that the translation scheme developed in the last chapter easily transforms into syntactic extensions. The transformation is possible, because of the absence of compile time environments; the translation is independent of its context.

Using syntactic extensions to implement the syntactic currying free us from the work involved in writing a compiler, and emphasize that syntactic currying is a transformation of programs from SE (symbolic evaluation semantics) into statically typed programs SSE, the syntactic extensions gives meaning to the constructions in SSE. Another solution would be to translate the programs from SSE into ordinary Scheme or some other implementation language.

This chapter are divided into three parts. A very superficial overview of the syntactic extensions in Chez Scheme, `extend-syntax`, [Dybvig 87,Kohlbecker 86,etal 86,Kohlbecker 87]. The development of a program annotation, so programs have a form that is suitable when the syntactic extensions have to recognize the various constructions in the programs. Finally the translation scheme from the last chapter is transformed into syntactic extensions.

5.1 Syntactic Extensions

Syntactic extensions are compile time macros. When the program is compiled the macros (or syntactic extensions) are expanded. Syntactic extensions are extremely powerful in languages like Scheme and LISP, because the source program is represented with the abstract datatype, lists. This makes it possible to inspect and rebuild the source expression without any restrictions. In fact the power at hand are so big that the construction `extend-syntax` [Kohlbecker 86] were constructed, such that the user were encouraged not to use the full potential of macros.

Syntactic extensions should not be confused with the run time macros of LISP. A run time macro is first applied at run time, and has therefore access both to the expression that are to be macro expanded, and to the value of the arguments. It is therefore nearly impossible to give a reasonable semantics for run time macros.

This section gives a short introduction to the macro facility, `extend-syntax`, in Chez Scheme [Dybvig 87]. It should not be thought of as a manual, only as an introduction. The aim is to make the definitions later in this chapter understandable, not to give them precise operational meanings.

Macro definition of list

```
(extend-syntax (list)
   [(list) '()]
   [(list e r ...) (cons e (list r ...))])
```

Expanding `(list 1 2 3)` yields

```
(cons 1 (cons 2 (cons 3 '())))
```

The first argument to `extend-syntax` is a list of keywords, of which the first is the macro name, the second argument is a set of rewrite rules. In this case there is only one keyword, the macro name, `list`, and two rules. Each rule consist of a pattern with keywords, and variables, and an expression with constants and variables. The syntactical extension works as follows: if the expression match the pattern (the pattern must be a list with the macro name as head) the expression is translated into the righthandside. The text matched by the variables introduced in the pattern, is inserted for the variables in the expression.

The first rule have only the keyword and no variables, and is expanded into nil. The second rule have two variables, `e`, and `r`, where `r` is followed by an ellipsis .... The ellipsis means, zero or more occurrences of the preceding variable, or pattern, in this case variable. This pattern matches lists with the macro name `list` as head, and (length > 1). The variable `e` matches the first argument to list, and `r` matches the rest of the arguments. Notice that this macro definition is recursive.

If it is necessary to compute anything at macro expansion time it must be done using the special `with`-clause as righthandside of a rule. The following example shows how the `with`-clause can be used to generate a new symbol at macro expansion time.

Macro definition of `or`

```
(extend-syntax (or)
   [(or) #f]
   [(or e r ...) (with ([n (gensym)]) (let ([n e]) (if n n (or r ...))))])
```

Expanding `(or a b)` yields

```
(let ([g16 a]) (if g16 g16 (let ([g17 b]) (if g17 g17 #f))))
```

This ends our very superficial introduction to `extend-syntax`. More information on the subject can be found in [Dybvig 87,Kohlbecker 86,etal 86,Kohlbecker 87].

5.2 Annotations

In order to make the translation using macro expansion, the macros must be able to recognize the various constructs. The various constructs are made recognizable by adding a tag like `base`, for all basic operations and `call`, for all function calls. Constructions like `letrec`, and `if`, are tagged already. The types of the various constructions are made explicit by annotating the tags. A dynamic basic operation is tagged, `base-d`. This annotation, and tagging forms an abstract syntax.

All constructions have either result type S or D. A straight forward annotation would be to put an "-s" on the tag if the construction had result type S, and an "-d" if it had result type D, e.g., `(base-d car` e`)`. To minimize the tags, and because constructions with result type S have the "ordinary" semantics, we shall not put "-s" on static constructions, but only put "-d" on the tags of dynamic constructions. In fact we shall not put any superficial tags on the static construct since their translation is the identity.

The arguments to a function may have different types. In order to resolve the types of the arguments from the call, or the function definition alone without looking to the arguments we split the arguments into to argument lists: a list of static arguments, and a list of dynamic arguments. Although it would be most logical to split both the formal, and the actual parameter list, only the formal parameter list is split into two. Because of this arrangement with two parameter lists, the arguments to the user defined functions have to be rearranged, such that the static arguments comes first, followed by the dynamic arguments. Another solution would be to add the type of the function explicitly, both to calls, and definitions, but that gives a less elegant translation scheme. In both case the annotation are supposed to be done by machine so the difference in notation is of little importance.

A few more annotations have to be introduced. These annotations are not type annotations but annotations that helps the partial evaluation phase. One such annotation is the "-e" on the if expression. If the condition in an if expression has type S, the if expression can be eliminated at static time, even if the branches have type D. Such an if expression are annotated as `(if-d-e` s t e`)`. As a matte of fact `if-d-e` is translated into an ordinary if, so we could leave it without the annotation. Another annotation is the annotation of calls as unfoldable. An unfoldable call is annotated as `call-d-u`, such a call have the same type as a `call-d`, the only difference is that it will be unfolded at static time.

The Abstract Source Syntax

$e, e_1, \ldots, e_n \in$ Exp
$l, l_1, \ldots, l_n \in$ Lambda
$o \in$ Operators
$p, p_1, \ldots, p_n \in$ ProcedureNames
$i, i_1, \ldots, i_n \in$ Identifiers
$t, t_1, \ldots, t_n \in$ Types

Exp ::= `I | C |` $(O\ e_1 \ldots e_n)$ `|` `(if` $e_0\ e_1\ e_2$`)` `|` $(p\ e_1 \ldots e_n)$ `|`
`(letrec ([`$p_1\ l_1$`]...[`$p_n\ l_n$`])` e`)` `|`
`(base-d` $o\ e_1 \ldots e_n$`)` `|` `(if-d` $e_0\ e_1\ e_2$`)` `|` `(call-d` $p\ e_1 \ldots e_n$`)` `|`
`(letrec-d ([`$p_1\ l_1$`]...[`$p_n\ l_n$`])` e`)` `|`
`(if-d-e` $e_0\ e_1\ e_2$`)` `|` `(call-d-u` $p\ e_1 \ldots e_n$`)`

Lambda ::= `(lambda (`$i_1 \ldots i_n$`)` e`)` `|` `(lambda-d (`$i_1 \ldots i_{s_j}$`) (`$i_1 \ldots i_{d_k}$`)` e`)`

The source program should follow the above syntax and obey the following type rules.

Type Inference Rules for Annotated Scheme

$$\frac{\begin{array}{c}\tau = [x_1 \mapsto S, \ldots, x_n \mapsto S, y_1 \mapsto D, \ldots, y_n \mapsto D] \\ \tau \vdash e : D\end{array}}{\texttt{(lambda-c } (x_1 \ldots x_n)\ (y_1 \ldots y_m)\ e\texttt{)} \text{ is well typed}}$$

$$\frac{\tau(v) = t}{\tau \vdash v : t} \qquad \tau \vdash \texttt{(quote } v\texttt{)} : S \qquad \frac{\tau \vdash e : S}{\tau \vdash \texttt{(lift } e\texttt{)} : D}$$

$$\frac{\tau \vdash e_1 : D, \ldots, \tau \vdash e_n : D}{\tau \vdash \texttt{(base-d } O\ e_1 \ldots e_n\texttt{)} : D} \qquad \frac{\tau \vdash e_1 : S, \ldots, \tau \vdash e_n : S}{\tau \vdash \texttt{(} O\ e_1 \ldots e_n\texttt{)} : S}$$

$$\frac{\tau \vdash e_1 : D, \ldots, \tau \vdash e_3 : D}{\tau \vdash \texttt{(if-d } e_1\ e_2\ e_3\texttt{)} : D} \qquad \frac{\tau \vdash e_1 : S, \ldots, \tau \vdash e_3 : S}{\tau \vdash \texttt{(if } e_1\ e_2\ e_3\texttt{)} : S} \qquad \frac{\tau \vdash e_1 : S, \tau \vdash e_2 : D, \tau \vdash e_3 : D}{\tau \vdash \texttt{(if-d-e } e_1\ e_2\ e_3\texttt{)} : D}$$

$$\frac{\begin{array}{c}\tau(f) = (S_1 \times \ldots \times S_n) \times S \\ \tau \vdash e_1 : S, \ldots, \tau \vdash e_n : S\end{array}}{\tau \vdash \texttt{(} f\ e_1 \ldots e_n\texttt{)} : S} \qquad \frac{\begin{array}{c}\tau(f) = (t_1 \times \ldots \times t_n) \times D \\ \tau \vdash e_1 : t_1, \ldots, \tau \vdash e_n : t_n\end{array}}{\tau \vdash \texttt{(call-d } f\ e_1 \ldots e_n\texttt{)} : D}$$

$$\frac{\begin{array}{c}\tau(f) = (t_1 \times \ldots \times t_n) \times D \\ \tau \vdash e_1 : t_1, \ldots, \tau \vdash e_n : t_n\end{array}}{\tau \vdash \texttt{(call-d-u } f\ e_1 \ldots e_n\texttt{)} : t}$$

$$\frac{\begin{array}{c}\tau' = \tau[\ \ f_1 \mapsto (S_{11} \times \ldots \times S_{1n_1}) \times S_1, \ldots, \\ \qquad f_k \mapsto (S_{k1} \times \ldots \times S_{kn_k}) \times S_k] \\ \tau'[x_{11} \mapsto S, \ldots, x_{1n_1} \mapsto S] \vdash e_1 : S, \ldots \\ \tau'[x_{k1} \mapsto S, \ldots, x_{kn_k} \mapsto S] \vdash e_k : S \\ \tau' \vdash e : S\end{array}}{\begin{array}{l}\tau \vdash \texttt{(letrec (}\ \ \texttt{[}f_1\ \texttt{(lambda } (x_{11}, \ldots, x_{1n_1})\ e_1\texttt{)]}, \ldots \\ \qquad\qquad\qquad \texttt{[}f_k\ \texttt{(lambda } (x_{k1}, \ldots, x_{kn_k})\ e_k\texttt{)])}\ e\texttt{)} : S\end{array}}$$

$$\frac{\begin{array}{c}\tau' = \tau[\ \ f_1 \mapsto (S_{11} \times \ldots \times S_{1n_1} \times D_{11} \times \ldots \times D_{1m_1}) \times D_1, \ldots, \\ \qquad f_k \mapsto (S_{k1} \times \ldots \times S_{kn_k} \times D_{11} \times \ldots \times D_{1m_1}) \times D_k] \\ \tau'[x_{11} \mapsto S, \ldots, x_{1n_1} \mapsto S, y_{11} \mapsto D, \ldots, y_{1m_1} \mapsto D] \vdash e_1 : D, \ldots \\ \tau'[x_{k1} \mapsto S, \ldots, x_{kn_1} \mapsto S, y_{k1} \mapsto D, \ldots, y_{km_1} \mapsto D] \vdash e_k : D \\ \tau' \vdash e : D\end{array}}{\begin{array}{l}\tau \vdash \texttt{(letrec-d (}\ \ \texttt{[}f_1\ \texttt{(lambda } (x_{11}, \ldots, x_{1n_1})\ (y_{11}, \ldots, y_{1m_1})\ e_1\texttt{)]}, \ldots \\ \qquad\qquad\qquad\quad \texttt{[}f_k\ \texttt{(lambda } (x_{k1}, \ldots, x_{kn_k})\ (y_{11}, \ldots, y_{1m_1})\ e_k\texttt{)])}\ e\texttt{)} : D\end{array}}$$

5.3 Implementation details

The implementation of the translation scheme, defined in the previous chapter, using **extend-syntax** is straight forward. This section describes the few changes that have been made. The full implementation can be found in appendix A.

5.3.1 Static Constructs

In chapter 4, it was realized that the static constructions should be translated using the identity function. *I.e.*, a static if-expression is implemented as an if-expression. Taking the consequence of this we do not translate the static construction at all, but leave them as they are.

5.3.2 Dynamic Constructs

Underlining a construction meant, this construction should be build and not evaluated. In Scheme this can be expressed using the backquote notation. Informally backquote "‘" means underline the following construction and comma "," means, remove the underline from the following expression. So "‘(a ,b)" means "$\underline{\texttt{(a}}\ \texttt{b}\underline{\texttt{)}}$". Except from that the constructions are implemented directly.

5.3.3 Function Calls and Definitions

In order to have both specializing calls, and unfolding calls, a function is defined as a pair of functions. The first performs a specializing call, and the second performs an unfolding call. It is then determined by the call which of the two functions, that are to be called, `call-d` calls the first, and `call-d-u` calls the second. The new variables are introduced by the function **make-sym**, that corresponds to gensym, but make more readable names.

To avoid an explicit notion of the types of the arguments the full handling of function calls is done by the definition of the function instead of the call. Likewise are the definition of dynamic letrec not split into two, because this would give problems with the introduced ***acc***. If we had a separate rule to translate a definition that rule would not know which accumulator the function should use. We use a with construction, in order to invent unique names for the accumulators at macro expansion time. This would not be necessary if we used hygienic macro expansion [etal 86] which makes variables that are introduced in the expansion unique.

5.3.4 The Syntactic Currying of Lambda

There is a problem which has not been mentioned in the previous chapters. There must be a way to introduce the static and dynamic variables. Until now this report have mostly been about symbolic evaluation as a way to achieve syntactic currying. Introducing a curried lambda, `lambda-c`, solves these problems. The curried lambda curry a non recursive function. To define a recursive function, one should use letrec. The reason for this is that the accumulator have to be introduced somewhere, in this case in the letrec.

Curried Lambda

$$\begin{aligned}
&\mathcal{T}[\![\,\texttt{(lambda-c } (v_{s_1}\ldots v_{s_j})\ (v_{d_1}\ldots v_{d_k})\ e\ ]\!] = \\
&\quad \texttt{(lambda } (v_{s_1}\ldots v_{s_j}) \\
&\qquad \texttt{(let ([}v_{d_1}\ \texttt{'*arg}_{d_1}\texttt{*]}\ldots\texttt{[}v_{d_k}\ \texttt{*arg}_{d_k}\texttt{*])} \\
&\qquad\quad \underline{\texttt{(lambda } (v_{d_1}\ldots v_{d_k})\ e\texttt{)}}\texttt{))}
\end{aligned}$$

The following little example show how the system works. But first a warning to "users". Never, never, make a syntactic curried version of a system function, never! *E.g.*, if one redefine the append function as a currying append all the system functions that depend on append will be highly confused.

Syntactic Currying of Append

```
> (define append-curried
    (lambda-c (x) (y)
      (letrec-d
        ([app (lambda-d (x) (y)
                (if-d-e (base null? x) y
                  (base-d (lift (base car x))
                          (call-d-u app (base cdr x) y))))])
        (call-d app x y))))
append-curried
> (append-curried '(1 2))
(lambda (y)
  (letrec
     ((app-1 lambda (y-1) (cons '1 (cons '2 y-1))))
    (app-1 y)))
```

Let append-12 be the syntactically curried version of append applied to the list `'(1 2)`. Applying append-12 to `'(3 4)` yields the same as applying the original append function to `'(1 2)` and `'(3 4)`[1].

```
> (append-12 '(3 4))
(1 2 3 4)
```

[1]There is a small problem here. The result of applying a syntactically curried function to its static arguments is a lambda expression (not a function) so we cannot run this as `((append-curry '(1 2)) '(3 4))`. One solution is to write the lambda expression on a file and then load the definition. `((store-and-load (append-curry '(1 2))) '(3 4))`. Another way is to use the system function eval (this is safe since it is applied to a closed term) `((eval (append-curry '(1 2))) '(3 4))`.

Chapter 6

Related Work

This chapter compares syntactic currying with related work, mainly partial evaluation. Syntactic currying, as it is described in this report, was developed, as a method to achieve partial evaluation. So it is not surprising, that the MIX-system and the system developed here, have a lot in common.

In the following we shall see, that not only do the functionality of some of the programs in the MIX system equal the functionality of programs in our system, the programs also have a common structure.

The chapter ends with a rephrasing of the three Futamura projections, that describes specialization of an interpreter, and with an explanation of the importance of the static type analysis.

6.1 Relating *mix* to *mcx*

Both *mix*, and *mcx*, although in different ways, achieve partial evaluation. In this section we shall relate the two programs, offering a more complete picture of partial evaluation. The equations below states the functionality of *mix*, and *mcx* respectively. For notational convenience we chose that all the languages in the definitions are equal. Through this chapter we shall call the program, *p*, and the input: *s*, and *d*, standing for static and dynamic.

It should be clear that *mix* is an implementation of the S^m_n function from Kleene's S^m_n-theorem [Kleene 52], while *mcx* is a syntactic curryer.

Definition of *mix*, and *mcx*

$$(L\ mix) : \overline{(S \times D \dot{\to} R)} \times S \dot{\to} \overline{(D \dot{\to} R)}$$
$$L\ (L\ mix\ (p,s))\ d = L\ p\ (s,d)$$

$$L\ mcx\ : \overline{(S \times D \to R)} \to \overline{S \to \overline{(D \to R)}}$$
$$L\ (L\ (L\ mcx\ p)\ s)\ d = L\ p\ (s,d)$$

From the types it is clear that *mcx* is a syntactically curried version of *mix*, and that that *mcx* performs syntactic currying. So running *mcx* on *mix* gives a residual program with the same functionality (or type) as *mcx*. The calculations below confirms this.

$$\begin{aligned}
mcx_{mix} &\overset{def}{=} L\ mcx\ mix \\
L\ (L\ (L\ mcx_{mix}\ p)\ s)\ d &= L\ (L\ (L\ (L\ mcx\ mix)\ p)\ s)\ d \\
&= L\ (L\ mix\ (p,s))\ d \\
&= L\ p\ (s,d)
\end{aligned}$$

Substituting equals for equals in the definition of *mix*, and *mcx* we get the following

$$L\ (L\ mix\ (p,s))\ d = L\ (L\ (L\ mcx\ p)\ s)\ d$$

which tells us the same as the types; the functionality (or types) of the residual programs are the same. The residual program resulting from partially evaluating p with respect to s using *mix*, have the same functionality as the residual program obtained using *mcx*. Not only do the two residual programs have the same functionality, they also share the same structure. The claim that two different programs are structurally equivalent calls for an explanation. There is of course no formal means by which we may conclude that program A is structurally equivalent to program B. So when we state that the programs are structurally equivalent we mean that they have some structure in common.

$$p_s \overset{def}{=} L\ mix\ (p,s) \overset{structure}{\equiv} L\ (L\ mcx\ p)\ s \overset{def}{=} p'_s$$

Examining the residual programs obtained using our system with the residual programs obtained using the mix system, we find that the structure of the residual programs are remarkably equal. One of the reasons for this, is that *mix*, and *mcx* are related programs; constructed using the same principles; *mcx* is built using our knowledge about how *mix* works.

6.2 Self-Application of *mix*

An interesting property of *mix* is that it can be self-applied. We can thus partially evaluate *mix* with respect to the program p, thereby obtaining a residual program mix_p which applied to s results in p_s. This program has the same functionality as the syntactically curried version of p, resulting from running ($L\ mcx\ p = mcx_p$).

$$mix_p \overset{def}{=} L\ mix\ (mix,p) \qquad \text{and} \qquad mcx_p \overset{def}{=} L\ mcx\ p$$

$$L\ (L\ mix_p\ s)\ d = L\ (L\ mcx_p\ s)\ d$$

The intuition behind the two programs, mix_p, and mcx_p, are the same. This is also described by their common type.

$$\begin{aligned}
&L\ m?x_p\ :\ S \rightarrow \overline{(D \rightarrow R)} \\
&\text{where}\quad L\ p\ :\ S \times D \rightarrow R
\end{aligned}$$

They take some static input, s, produce a residual program which, when applied to the dynamic input, d, gives the final result. This suggest that mix_p, and mcx_p share some structure.

$$mix_p \overset{structure}{\equiv} mcx_p$$

In the system developed in this paper, the program mcx_p is the result of macro expansion. Comparing mix_p with mcx_p we find that the two programs have roughly the same structure. This statement should be read: we are able to recognize common structures.

Going one step further, examining mix_{mix} which is the result of partially evaluating *mix* with respect to *mix*. ($mix_{mix} = L\ mix\ (mix, mix)$), we find that it should be compared to *mcx* itself.

$$L\ (L\ (L\ mix_{mix}\ p)\ s)\ d = L\ (L\ (L\ mcx\ p)\ s)\ d$$

Comparing *mcx* with mix_{mix} is a bit difficult, because *mcx* is a set of syntactical extensions (macros), each defining the translation of a language construction into its curried version. While mix_{mix} is a generated program. Nevertheless, examining mix_{mix}, we find pieces of code that are almost identical to our syntactical extensions.This suggest the following view on *mcx*: *mcx* is a hand written version of mix_{mix}.

6.3 The Futamura Projections

There are three special instances of partial evaluation, that have drawn much attention since 1971, where they were introduced by Futamura in [Futamura 71]; the three Futamura projections. The Futamura projections considers the special case where the program being partially evaluated is an interpreter, *int*. In that case the static data is a source program, *src*, and the dynamic data is the input to the source program, *data*. The specialized version of the interpreter with respect to the source program, int_{src}, is a target program, *mix* specialized with respect to *int* is a compiler, *comp*, and *mix* specialized with respect *mix* itself is a compiler generator, *cogen*.

The three Futamura projections

$$\begin{aligned} L\ mix\ (int, src) &\overset{def}{=} target \\ L\ mix\ (mix, int) &\overset{def}{=} comp \\ L\ mix\ (mix, mix) &\overset{def}{=} cogen \end{aligned}$$

$$\begin{aligned} &L\ target\ data = L\ (L\ mix\ (int, src))\ data = result \\ &L\ comp\ src = L\ (L\ mix\ (mix, int))\ src = target \\ &L\ cogen\ int = L\ (L\ mix\ (mix, mix))\ int = comp \end{aligned}$$

What is the role of *mcx* in this case? We have already seen that *mcx* corresponds to *cogen* (or mix_{mix}), so

$$\begin{aligned} L\ mcx\ int &= comp \\ L\ comp\ src &= target \\ L\ target\ data &= result \end{aligned}$$

6.4 Language Triplets

The theory about language triplets developed in [Holst 88], gives a description of the relation between *mix*, and *mcx*. The main theorem in the theory of language triplets states that there exists a language, L^S, such that programs that are M-interpreters (interpreters for the language M) in L are compilers from M to S written in L^S. Let *int* be a M-interpreter, ad *src* a M-program. then the theorem states that there exists a language L^S that fulfills the following requirements.

$$L\ int\ (src, data) = S\ (L^S\ int\ src)\ data$$

It is one of the observations in [Jones 85b,Holst 88] that *mix* is an interpreter for a language like L^S. Likewise *mcx* can be seen as a compiler from that language into, say, T.

$$S\ (L\ mix\ (int, src)\ data = S\ (L^S\ int\ src)\ data = S\ target\ data$$

$$S\ (T\ (L\ mcx\ int)\ src)\ data = S\ (L^S\ int\ src)\ data = S\ target\ data$$

The conclusion is that *mcx* is an compiler for the language that *mix* is an interpreter for.

Let L, S, and T be the same language. Then L^L is $Curry(S^1_1)$. Where S^1_1 is the S^m_n function for $m = n = 1$. S^1_1 is the function computed by *mix*. This corresponds beautifully with the the following view on interpreters: an interpreter for language L is an uncurry'ed version of L; so an L-interpreter is an implementation of $UnCurry(L)$. As *mix* is an L^L-interpreter, *mix* is an implementation of $UnCurry(Curry(S^1_1)) = S^1_1$.

6.5 The Importance of Binding Time Typing

The type analysis, as we prefer to call, it corresponds to the binding time analysis in the MIX-system [Bondorf 88]. Both in the MIX-system, and in our implementation the type analysis is performed in a separate prephase which annotates the program with type information. The type analysis need not be performed in a separate prephase, but it is crucial that it can be performed at translation time. Like in other languages static typing removes the need for run time type checks, and thus leads to more efficient implementations. Type checking at run time in the language L_{pe} is fairly expensive; experiments shows that it introduce an overhead between 20, and 50 [Holst 89a], not to mention the space overhead.

In the table below we have listed the overhead introduced if we use a run time type system. The only case in which the overhead grows to unbearable sizes is when self-applying *mix*. This is because the overhead, o, grows to o^2 when *mix* is self-applied.

$mcx_p = L\ mcx\ p$	none
$p_s = L\ mcx_p\ s$	20 to 50
$result = L\ p_s\ d$	none
$p_s = L\ mix\ (p, s)$	20 to 50
$result = L\ p_s\ d$	none
$mix_p = L\ mix\ (mix, p)$	400 to 2500

6.6 Conclusion

Summarizing this chapter: A syntactic curryer is a syntactic curried version of a partial evaluator, and opposite a partial evaluator is a syntactic uncurried version of a syntactic curryer. The program *mcx* is a hand written version of *cogen*.

Chapter 7

Experiments

The experiments in this chapter falls into two categories. Some experiments aim at showing what syntactic currying can be used to. Others compare the results obtained by this system with the results obtained with partial evaluators, like the MIX system.

There are no systematic tests, that try to prove the system correct, since this is an experimental system. But the sequence of experiments carried out in this chapter, makes us believe that the number of errors in the system is small.

This chapter reports the following experiments. Specialization of the power function, which lifts y to the power x. Then two different interpreters are curried, a Turing machine interpreter, and a interpreter for recursive equations. The curried versions of the interpreters are compilers and they are used to compiler some small example programs. An example that goes into an infinite loop is shown, and it is shown how to rewrite the program, so the curried version terminates. Finally some run time statistics are listed.

7.1 Power, or y^x

This is a classical example. Consider the function, power, defined below, that lift y to the x'th power, and suppose the first argument, x, is static and the second, y, is dynamic.

Power, A Classical Example

```
(define power
  (lambda-c (x) (y)
    (letrec-d '([p (lambda-d (x) (y)
                    (if-d-e (zero? x)
                            (lift '1)
                    (if-d-e (even? x)
                            (call-d-u p (quotient x '2) (base-d sqr y))
                            (base-d * y (call-d-u p (- x '1) y)))))])
      (call-d-u p x y))))
power
> (power '5)
(lambda (y) (* y (* (sqr (sqr y)) '1)))
```

This example is interesting, because all the control has been solved. The result could have been slightly better if the system had been able to reduce the multiplication with the unity 1. But such optimizations are not worth the effort, except in case where arithmetical computation plays a central role; an example can be found in [Mogensen 86], where a raytracer is specialized. There is a infinite class of such optimizations, and the number of times they can be applied is very small compared to the number of times where they do not apply. Therefore they will make the program less efficient at static time, and only give a small speedup at static time.

This example also shows some of the dangers in unfolding calls. Suppose that the square function, was a user defined function, like the one below, instead of a basic operation. If such a user defined square function had been called with unfoldable calls. The result would be as seen below, *i.e.*, $O(n)$ operations instead of $O(\log n)$ operations.

```
([sqr (lambda-d () (y) (base-d * y y))])
(lambda (y) (* y (* (* (* y y) (* y y)) '1)))
```

It is easy to think of examples where the run time goes from linear time to exponential time. So unfolding of function calls is a delicate matter. In [Sestoft 88] it is described how call duplication, as described above, can be detected automatically. Another solution to this problem is to insert let-expressions at the point of each function call.

Annotating the Program with let-expressions to avoid Call Duplication It is possible to avoid call duplication completely using the following principle which removes the need for runtime checks. Put let-expressions around unfoldable calls, such that all the arguments to unfoldable calls are variables.

Unfortunately our system do not handle let-expressions, but if it did, this solution could be used. By the way, let-expressions are not difficult to handle.

7.2 A Turing Machine Interpreter

In the last chapter we saw that syntactic currying of an interpreter yields a compiler. In this section we examine the result of using a syntactic curried Turing machine interpreter to compile small example programs.

We use Post's variant of the Turing machine, inspired by [Gomard 88]. A Turing program is a list of instructions $(I_0, \ldots, I_n)$. There are the following instructions: (left), (right), (jmp l), (if t l), (write t). The tape of the Turing machine is infinite in both directions. Initially the head is placed over the tape at the beginning of the input string. The instructions are computed in sequence, with the following exceptions: (jmp l) continues with instruction I_l, and (if t l) continues with I_l if the tape symbol under the head is t, otherwise it continues with the next instruction. (write t) write t on the tape at the position under the head. (left), and (right) moves the head left and right on the tape.

The Turing machine interpreter below use head, and tail when the tape is moved. The infinite tape is implemented by defining that: head of the empty list is blank, tail of the empty list is the empty list. The complete definition of the definition of the Turing machine interpreter can be found in the appendix B, in which all the auxiliary functions are defined.

A Turing Machine Interpreter

```
(define tint-r (lambda-c (p) (t)
  (letrec-d
    ([ex (lambda-d (pc) (l r)
           (if-d-e (null? pc)
                   r
           (if-d-e (left? pc)
                   (call-d-u ex (next pc)
                                (base-d tail l)
                                (base-d cons (base-d head l) r))
           (if-d-e (right? pc)
                   (call-d-u ex (next pc)
                                (base-d cons (base-d head r) l)
                                (base-d tail r))
           (if-d-e (write? pc)
                   (call-d-u ex (next pc)
                                l
                                (base-d cons (lift (get1 pc))
                                             (base-d tail r)))
           (if-d-e (if? pc)
                   (if-d (base-d eq? (lift (get1 pc)) (base-d head r))
                         (call-d ex (jmp-to (get2 pc) p) l r)
                         (call-d ex (next pc) l r))
           (if-d-e (jmp? pc)
                   (call-d-u ex (jmp-to (get1 pc) p) l r)
                   (lift 'error))))))))])
  (call-d ex p (lift '()) t))))
```

Scheme is not the best suited language for implementation of the Turing machine with its assignments, so the resulting code tends to be rather large.

Therefore we just show a small example namely the example which can be found in [Gomard 88]. This Turing program change the first occurrence of 0 with a 1 and returns the rest of the tape. A larger example can be found in appendix B

```
(
   (if 0 3)   ;   0
   (right)    ;   1
   (jmp 0)    ;   2
   (write 1)  ;   3
)
```

The compiled version of this program is the following Scheme code.

```
(lambda (t)
   (letrec
       ((ex-3 (lambda (l-3 r-3)
                 (if (eq? '0 (head (tail r-3)))
                     (ex-2 (cons (head r-3) l-3) (tail r-3))
                     (ex-3 (cons (head r-3) l-3) (tail r-3)))))
        (ex-2 (lambda (l-2 r-2) (cons '1 (tail r-2))))
        (ex-1 (lambda (l-1 r-1)
                (if (eq? '0 (head r-1)) (ex-2 l-1 r-1) (ex-3 l-1 r-1)))))
     (ex-1 '() t)))
```

As already said Scheme is not the best suited language for implementing the Turing machine with its assignments. Nevertheless the code seems quite reasonable.

7.3 An Interpreter for Recursive Equations

The annotated interpreter for recursive equations is found in appendix C. We have chosen to translate the normal Fibonacci program by applying the syntactically curried interpreter to it. Below we have shown the definition of the Fibonacci function in recursive equation(s) and the translated version in Scheme. the main difference is the part that builds the argument list. Using some kind of variable splitting like the ones described in [Jones 89a,Mogensen 88a, Romanenko 88] the variable list would have been split into separate variables. In this case there is only one variable.

```
(
 (fib (n) (if (< n '2) '1
              (+ (fib (- n '1)) (fib (- n '2)))))
)

(lambda (input)
   (letrec
        ((exec-5
            (lambda (values-5)
               (if (< (car values-5) '2)
                   '1
                   (+ (exec-5 (cons (- (car values-5) '1) '()))
                      (exec-5 (cons (- (car values-5) '2) '()))))))))
      (exec-5 input)))
```

The two examples shows that this implementation is comparable to the implementations in [Jones 85a,Gomard 88].

7.4 Infinite Specialization

In chapter 2 we began to share specialized function in order to avoid infinite specialization. Nevertheless, the risk of infinite specialization is still present. In this section an example that produce infinitely many specialized functions are exhibited, and it is shown how the program can be rewritten, such that only finitely many specialized functions are generated.

The example program is the power function from the first section. This time we swap the arguments, so it is curried with respect to y being static.

Power(2), A Classical Example

```
> (define power2
    (lambda-c (y) (x)
      (letrec-d
        ([p (lambda-d (y) (x)
              (if-d (base-d zero? x)
                    (lift y)
              (if-d (base-d even? x)
                    (call-d p (sqr y) (base-d quotient x (lift '2)))
                    (base-d * (lift y) (call-d p y (base-d - x (lift '1)))))))])
        (call-d p y x))))
power2
> (power2 '5)
```
goes into an infinite loop

Examining the program we can see that we first makes a specialized version of p with $y = 5$, that version gives rise to a call to p with $y = 5^2 = 25$, which is then specialized. This continues with $y = 5, 25, 625, \ldots$, the problem we arrive at is a general one. Static accumulating arguments gives rise to infinite unfolding. This seems a little awkward at first, since we normally try to make our programs tail recursive. The solution to the problem is to rewrite power, such that it no longer have an accumulating argument.

Power(3), A Classical Example

```
> (define power3
    (lambda-c (y) (x)
      (letrec-d
        ([p (lambda-d (y) (x)
              (if-d (base-d zero? x)
                    (lift y)
              (if-d (base-d even? x)
                    (base-d sqr (call-d p y (base-d quotient x (lift '2))))
                    (base-d * (lift y) (call-d p y (base-d - x (lift '1)))))))])
        (call-d p y x))))
power3
> (power3 '5)
(lambda (x)
   (letrec
        ((p-3 (lambda (x-3)
                 (if (zero? x-3)
                     '5
                     (if (even? x-3)
                         (sqr (p-3 (quotient x-3 '2)))
                         (* '5 (p-3 (- x-3 '1))))))))
      (p-3 x)))
```

Another solution to the problem, which always works would be to lift the static arguments to dynamic. This is the general solution, but one gets lousy results by just throwing information away.

7.5 Comparison with other Systems

In contrast to other works in this area this system have no mix program. It is therefore not possible to make experiments with self-application. Compiler generation happens at macro expansion time and is taken care of by the Scheme system. Experiments shows that macro expansion take some time; defining a normal interpreter takes 40 ms. while definition of an annotated interpreter, *i.e.*, a compiler, takes 60 ms. But it is difficult to make any clear conclusions from these figures. The run times are cpu milliseconds and are compiled using the time function of Scheme. Compilation is very fast. All the programs we have compiled have been compiled in 20 ms. or less. The code generated by our compilers are comparable to the code generated by other partial evaluators [Jones 85a,Jones 89a,Holst 88].

Our results can not compete with the results obtained by [Mogensen 88b,Bondorf 89a], because our system do not use variable splitting. We believe that our system without large difficulties could be extended to do variable splitting.

We report on two experiments: an interpreter for a Turing machine which together with a closer description can be found in appendix B, and an interpreter for recursive equations. The interpreter for recursive equations is written in a style close to the self-interpreters in [Sestoft 86,Jones 89a,Holst 88]. The interpreter for recursive equations can be found in appendix C.

	Int(p,d)	Int_p(d)	ratio
Turing-int	412	80	5.26
Self-int	4268	224	19.05

The figures in the table below is the time for interpreting fib with input 15 and for running a compiled version of fib on 15. The experiment with the Turing machine reported in the table above can be found in appendix B.

Chapter 8

Conclusion

This conclusion naturally divides into two independent parts. A technical summary, stating the achievements of the project. A historical summary, stating where the ideas and inspiration to this work came from and how the project developed.

8.1 Achievements

This report describes syntactic currying; a technique that in many respects is very close to partial evaluation.

Syntactic currying is the transformation of a program, p, that works in one phase into a program, p', that works in two phases. The splitting of the program reflects a splitting of the input into two parts: the static part, and the dynamic part. The program, p', works in two phases. The first phase produce a residual function when applied to the static data. The residual function then performs the second phase when applied to the static data.

We implemented syntactic currying in several steps. First we defined a evaluation strategy, which we called symbolic evaluation, symbolic evaluation was evaluation over a domain of expressions. We recognized that symbolic evaluation was dynamically typed, with the types: static, and dynamic (representing value, and expression), and that all language constructs was overloaded with respect to these types. A type inference was developed, and it was shown how the dynamically typed programs could be transformed into statically typed program by insertion of type conversion operations.

This view of binding times as types, binding type analysis as a type inference, and annotation as transformation of the program into a form that is statically typed, and where the overloading of the constructions is solved, is a contribution of this project. Nielson & Nielson uses types in their two level meta language, in a way very similar to the approach taken in this project [Nielson 86].

Given that the programs was statically typed, and the overloading was solved, we gave a much more efficient implementation of symbolic evaluation, and thus syntactic currying.

This implementation of syntactic currying corresponds closely to the normal implementation of partial evaluation. One of the differences is that we use a depth-first strategy in our function specialization while the MIX-system uses a breadth-first strategy. This had the advantage that recursive declarations could be treated locally, since the need for a global loop was removed.

Given the statically typed version of the source syntactic currying turned out to be straight forward. The syntactic curryer was implemented using Chez Schemes syntactic extensions, `extend-syntax`. The system was compared with the MIX-system. It was realized that the syntactic curryer, *mcx*, corresponded to the result of the third Futamura projection, *cogen*. This way the structure of the compilers and compiler generator produced by the MIX-system could be described.

Lines were drawn to language triplets [Holst 88], and it was shown that *mcx* was a compiler for the language, L_{pe}, that *mix* was an interpreter for. *mcx* is a syntactically curried version of *mix*, and *mix* is an syntactically uncurried version of *mcx*, which is the same as an implementation of the S^m_n-function.

Concluding on this report, one might say that the report has not so much new to say, but it offers an enlightening view on syntactic currying, and partial evaluation. We also believe that it close some of the gaps between partial evaluation and other compiler generator systems. For example: the compile structure of *mcx* is closer to denotational like systems like CERES, still *mcx* is a syntactically curried version of *mix*. This shows that the interpretive structure of *mix* is not needed to implement to implement partial evaluation.

8.2 The Story About a Project

This project started with the idea, that the annotated version of programs in the mix system, already was the compiler, and that it only was a question about the language in which one considered them as programs. This was also the idea in the article [Holst 88], and in the thesis [Holst 89a]. It also started with the idea that partial evaluation could be achieved using the macro facility in Chez Scheme, `extend-syntax`, [Dybvig 87,Kohlbecker 86]. In the beginning the project was inspired by work done on partial evaluating the lambda-calculus, by Carsten Gomard, and Neil D. Jones [Jones 89b]. Later, in cooperation with Olivier Danvy, it became obvious that this was actually syntactic currying, and this approach had some advantages compared to the normal implementation of partial evaluation. Currently an article giving an extract of these ideas is being written together with Olivier Danvy [Holst 89b].

8.3 Acknowledgements

I would very much like to thank Olivier Danvy who was the first that encourage me to write about syntactic currying, and later supervised this report. Thanks also goes to Carsten Gomard, and Helene Garde. Carsten, because he forced me to make this (my last project), and kept telling me that it only was supposed to be a 75 hours written project. Helene because she kept telling me what was going on in the outside world. Last, but not least I would like to thank the Copenhagen MIX group, lead by Professor Neil D. Jones, which have been a very stimulating environment for the last four years[1].

POPL 87 and LISP conference 86 E. Kohlbecker hygenic extensions

[1]As an aside I would also like to thank the computer department at DIKU, although they weren't paid for it, they would reboot the machine every time it failed, even on sundays, and at the very small hours. And they didn't throw my files away.

Bibliography

[Bondorf 88] Bondorf, A., Jones, N.D., Mogensen, T., and Sestoft, P. *Binding Time Analysis and the Taming of Self-Application.* Draft, 18 pages, DIKU, University of Copenhagen, Denmark, August 1988.

[Bondorf 89a] Bondorf, A. A Self-Applicable Partial Evaluator for Term Rewriting Systems. In Diaz, J. and Orejas, F. (editors), *TAPSOFT '89. Proc. Int. Conf. Theory and Practice of Software Development, Barcelona, Spain, March 1989. (Lecture Notes in Computer Science, Vol. 352)*, pages 81–95. Springer-Verlag, 1989.

[Bondorf 89b] Bondorf, Anders. Partial Evaluation of Higher-Order Scheme. 1989. in preparation.

[Dybkjaer 85] Dybkjær, H. *Parsers and Partial Evaluation: An Experiment.* Student report no. 85-7-15, 128 pp., DIKU, University of Copenhagen, Denmark, July 1985.

[Dybvig 87] Dybvig, Kent. *The Scheme Programming Language.* Prentice-Hall,inc., 1987.

[etal 86] et. al., Eugene E. Kohlbecker. Hygienic Macro Expansion. In *Conference on LISP and Functional Programming*, ACM, pages 151–161. 1986.

[Futamura 71] Futamura, Y. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls* 2(5):45–50, 1971.

[Gomard 88] Gomard, Carsten K. and Jones, Neil D. *Compiler Generation by Partial Evaluation: a Case Study.* DIKU tech. report 88/24, DIKU, University of Copenhagen, Denmark, December 1988. to be published.

[Holst 88] Holst, N.C.K. Language Triplets: The AMIX Approach. In Bjørner, D., Ershov, A.P., and Jones, N.D. (editors), *Partial Evaluation and Mixed Computation*, pages 167–185. North-Holland, 1988.

[Holst 89a] Holst, Carsten K. Program Specialization for Compiler Generation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1989.

[Holst 89b] Holst, Carsten Kehler and Danvy, Olivier. Partial Evaluation without a Partial Evaluator. 1989. in preparation.

[Jones 85a] Jones, N.D., Sestoft, P., and Søndergaard, H. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In Jouannaud, J.-P. (editor), *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, Vol. 202)*, pages 124–140. Springer-Verlag, 1985.

[Jones 85b] Jones, Neil D. and Tofte, Mads. Towards a Theory of Compiler Generation. In Bjøner, D. (editor), *Workshop on Formal Software Development Methods.* North-Holland, 1985.

[Jones 89a] Jones, N.D., Sestoft, P., and Søndergaard, H. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation* 2(1):9–50, 1989.

[Jones 89b] Jones, Neil D. and Gomard, Carsten K. Partial Evaluation of the λ-calculus. 1989. in preparation.

[Kleene 52] Kleene, S.C. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.

[Kohlbecker 86] Kohlbecker, Eugene E. *Syntactic Extensions in the Programming Language LISP.* PhD thesis, Indiana University, 1986.

[Kohlbecker 87] Kohlbecker, Eugene E. and Wand, Mitchell. Macro-by-example: Deriving syntatic transformations from their specifications. In *Symposium on Principles of Programming Languages*, ACM, pages 77–84. Munich, West Germany, January 1987.

[Mogensen 86] Mogensen, T. The Application of Partial Evaluation to Ray-Tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[Mogensen 88a] Mogensen, T. Partially Static Structures in a Self-Applicable Partial Evaluator. In Bjørner, D., Ershov, A.P., and Jones, N.D. (editors), *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[Mogensen 88b] Mogensen, Torben and Holst, Carsten Kehler. Terminology. In *Partial Evaluation and Mixed Computation*, IFIP World Congress Proceedings. North-Holland, 1988.

[Nielson 86] Nielson, Flemming. Code Generation for a two level denotational meta Language. In Ganzinger, H. and Jones, N.D. (editors), *Programs as Data Objects, Copenhagen, Denmark, 1985. (Lecture Notes in Computer Science, Vol. 217)*, page ??? Springer-Verlag, 1986.

[Rees 86] Rees, Jonathan A. and Clinger, William. The Revised^3 Report on the Algorithmic Language Scheme. *Sigplan Notices* 21(12), December 1986.

[Romanenko 88] Romanenko, S.A. A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure. In Bjørner, D., Ershov, A.P., and Jones, N.D. (editors), *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.

[Sestoft 86] Sestoft, P. The Structure of a Self-Applicable Partial Evaluator. In Ganzinger, H. and Jones, N.D. (editors), *Programs as Data Objects, Copenhagen, Denmark, 1985. (Lecture Notes in Computer Science, Vol. 217)*, pages 236–256. Springer-Verlag, 1986.

[Sestoft 88] Sestoft, P. Automatic Call Unfolding in a Partial Evaluator. In Bjørner, D., Ershov, A.P., and Jones, N.D. (editors), *Partial Evaluation and Mixed Computation*, pages 485–506. North-Holland, 1988.

Appendix A

The whole definition of Syntactic Currying

A.1 The definition of syntactic currying of definitions

```
(extend-syntax (lambda-c)
 [(lambda-c (s ...) (d ...) body)
  (lambda (s ...)
     '(lambda (d ...) ,(let ([d 'd] ...) body)))])
```

A.2 The Static part

The static part is out-commented because it is the identity function. These syntactic extensions would not function if defined.

```
;; The Static part

;(extend-syntax (base)
;  [(base o e ...) (o e ...)])

;(extend-syntax (if)
;  [(if e1 e2 e3) (if e1 e2 e3)])

;(extend-syntax (letrec)
;  [(letrec d e) (letrec d e)])

;(extend-syntax (call)
;  [(call f e ...) (f e ...)])
```

A.3 The Dynamic part

The variables `*acc*`,`*rdef*`, `*done*`, and `*rbody*` are invented at macro expansion time.

```
(extend-syntax (lift)
  [(lift e) '(quote ,e)])

(extend-syntax (base-d)
  [(base-d o e ...) `(o ,e ...)])

(extend-syntax (if-d)
  [(if-d e1 e2 e3) `(if ,e1 ,e2 ,e3)])

(extend-syntax (if-d-e)
  [(if-d-e e1 e2 e3) (if e1 e2 e3)])

(extend-syntax (letrec-d lambda-d)
  [(letrec-d ([n (lambda-d (s ...) (d ...) le)] ...) e)
   (with ([*acc* (make-sym '*acc*)]
          [*rdef* (make-sym '*rdef*)]
          [*done* (make-sym '*done*)]
          [*rbody (make-sym 'rbody)])
     (let ([*acc* (empty-acc)])
       (letrec
         ([n (cons
               (lambda (s ... d ...)
                 (let ([*done* (called? *acc* 'n `(,s ...))])
                   (if (null? *done*)
       (let ([*rdef* (let ([d (make-sym 'd)] ...)
   `(lambda (,d ...) ,le))])
                          `(,(specialize! *acc* 'n `(,s ...) *rdef*) ,d ...))
                        `(,*done* ,d ...))))
               (lambda (s ... d ...) le))] ...)
         (let ([*rbody* e])
           (extract! *acc* *rbody*)))))])

(extend-syntax (call-d)
  [(call-d f e ...) ((car f) e ...)])

(extend-syntax (call-d-u)
  [(call-d-u f e ...) ((cdr f) e ...)])
```

A.4 The auxiliary functions, and macros

The accumulator is a list of pairs, the head the function anme and the value of the static arguments are recorded, in the tail the residual funciton definition is stored.

```
(extend-syntax (empty-acc)
  [(empty-acc) ()])

(extend-syntax (called?)
  [(called? a n s)
   (let ([*d* (assoc (cons n s) a)])
      (if (null? *d*)
          (begin (set! a (cons (cons (cons n s) (list (make-sym n))) a)) '())
          (cadr *d*)))])

(extend-syntax (extract!)
  [(extract! a b)
   (if (null? a) b `(letrec ,(map cdr a) ,b))])
```

```
(extend-syntax (specialize!)
  [(specialize! a n s d)
   (let ([*d* (assoc (cons n s) a)])
      (if (null? *d*) (error 'specialize! "Couldn't find it")
      (begin (set-cdr! (cdr *d*) (list d)) (cadr *d*))))])
```

The following is our "improved" definition of gensym.

```
(define (make-cif n)
  (if (and (number? n)(<= 0 n)(<= n 9))
    (list-ref '("0" "1" "2" "3" "4" "5" "6" "7" "8" "9") n)
    (error 'make-cif "~s is not a ciffer" n)))

(define (make-number n)
    (if (number? n)
       (if (= n 0) "0" (let f ([x n])
          (if (= x 0) "" (string-append (f (quotient x 10))(make-cif (mod x 10))))))
       (error 'make-number "~s is not a number" n)))

(define make-sym
     (let ([*symbols* '()])
        (lambda (s)
           (let ([*s* (assoc s *symbols*)])
              (if (null? *s*)
                  (begin (set! *symbols* (cons (cons s 0) *symbols*))
                         (make-sym s))
                  (begin (set-cdr! *s* (+ 1 (cdr *s*)))
                         (string->symbol
                           (string-append
                              (symbol->string s) "-"
                              (make-number (cdr *s*))))))))))
```

Appendix B

A Turing Machine Interpreter

We use Post's variant of the Turing machine, which was also used in [Gomard 88]. A Turing program is a list of instructions $(I_0, \ldots, I_n)$. There are the following instructions: (left), (right), (jmp n), (if t n), (write t). The instruction are computed in sequence unless the instruction is (jmp n), or (if t n). (jmp n) means continue with instruction I_n. (if t n) means if the symbol on the tape is t continue with I_n otherwise continue with the next instruction. (write t) write t on the tape. (left), and (right) moves the head left and right on the tape. The tape is infinite in both directions and there are blanks on the tape unless there is specified something else.

The Turing machine interpreter below use head, and tail when the tape is moved. The infinite tape is implemented by defining that: head of the empty list is blank, tail of the empty list is the empty list

```
(define tint-r (lambda-c (p) (t)
  (letrec-d
    ([ex (lambda-d (pc) (l r)
          (if-d-e (null? pc)
                  r
          (if-d-e (left? pc)
                  (call-d-u ex (next pc)
                               (base-d tail l)
                               (base-d cons (base-d head l) r))
          (if-d-e (right? pc)
                  (call-d-u ex (next pc)
                               (base-d cons (base-d head r) l)
                               (base-d tail r))
          (if-d-e (write? pc)
                  (call-d-u ex (next pc)
                               l
                               (base-d cons (lift (get1 pc))
                                            (base-d tail r)))
          (if-d-e (if? pc)
                  (if-d (base-d eq? (lift (get1 pc)) (base-d head r))
                        (call-d ex (jmp-to (get2 pc) p) l r)
                        (call-d ex (next pc) l r))
          (if-d-e (jmp? pc)
                  (call-d-u ex (jmp-to (get1 pc) p) l r)
                  (lift 'error))))))))])
  (call-d ex p (lift '()) t))))
```

B.1 The Auxiliary Functions

```
(define (left? c) (eq? 'left (caar c)))
(define (right? c) (eq? 'right (caar c)))
(define (write? c) (eq? 'write (caar c)))
(define (if? c) (eq? 'if (caar c)))
(define (jmp? c) (eq? 'jmp (caar c)))
(define (head l) (if (null? l) 'b (car l)))
(define (tail l) (if (null? l) '() (cdr l)))
(define (next p) (cdr p))
(define (get1 i) (cadar i))
(define (get2 i) (caddar i))
(define (jmp-to n p)
  (let nth ([n n][p p])
    (if (= 0 n) p
(nth (- n 1) (next p)))))
```

B.2 A Turing Program that Multiply by 2

To test the Turing machine interpreter we made a program that multiplied by two. The program takes a sequence of 1's and returns a sequence with twice as many zeros.

```
(define tp '(
  (if b 15)   ;  0     if n=0 end
  (if 0 15)   ;  1     if finished end
  (write b)   ;  2     delete a 1
  (right)     ;  3
  (if b 7)    ;  4     move to end
  (right)     ;  5
  (jmp 4)     ;  6
  (write 0)   ;  7     write two 0
  (right)     ;  8
  (write 0)   ;  9
  (if b 13)   ; 10     move to start
  (left)      ; 11
  (jmp 10)    ; 12
  (right)     ; 13
  (jmp 1)     ; 14     repeat
)
```

The result of compiling this Turing program is the following amount of scheme code. Purely functional Scheme is not the best suited programming language when it comes to implementation of the Turing machine with its assignments.

```
(lambda (t)
   (letrec
        ((ex-10
            (lambda (l-10 r-10)
               (if (eq? 'b (head (tail r-10)))
                   (ex-7 (cons (head r-10) l-10) (tail r-10))
                   (ex-10 (cons (head r-10) l-10) (tail r-10)))))
         (ex-9 (lambda (l-9 r-9)
          (if (eq? 'b (head (cons (head l-9) r-9)))
              (ex-8 (tail l-9) (cons (head l-9) r-9))
              (ex-9 (tail l-9) (cons (head l-9) r-9)))))
   (ex-8 (lambda (l-8 r-8)
          (if (eq? '0 (head (tail r-8)))
              (ex-5 (cons (head r-8) l-8) (tail r-8))
              (ex-6 (cons (head r-8) l-8) (tail r-8)))))
   (ex-7 (lambda (l-7 r-7)
          (if (eq? 'b
                   (head (cons '0
                               (tail (tail (cons '0
                                                 (tail r-7)))))))
              (ex-8 (cons (head (cons '0 (tail r-7))) l-7)
                    (cons '0 (tail (tail (cons '0 (tail r-7))))))
              (ex-9 (cons (head (cons '0 (tail r-7))) l-7)
                    (cons '0
                          (tail (tail (cons '0 (tail r-7)))))))))
   (ex-6 (lambda (l-6 r-6)
          (if (eq? 'b (head (tail (cons 'b (tail r-6)))))
              (ex-7 (cons (head (cons 'b (tail r-6))) l-6)
                    (tail (cons 'b (tail r-6))))
              (ex-10
                 (cons (head (cons 'b (tail r-6))) l-6)
                 (tail (cons 'b (tail r-6)))))))
   (ex-5 (lambda (l-5 r-5) r-5))
   (ex-4 (lambda (l-4 r-4)
          (if (eq? '0 (head r-4)) (ex-5 l-4 r-4) (ex-6 l-4 r-4)))))
(ex-4 '() t)))
```

Appendix C

An Interpreter for a Recursive Equations

The interpreter shown in this appendix is very limited. It only supports the instructions needed to interpret the fibonacci program. A larger interpreter or an interpreter which used an external function "base-eval" to implement the basic operation could easily be made.

```
(define int-2
  (lambda-c (program) (input)
    (letrec-d
      ([exec (lambda-d (exp names) (values)
          (letrec-d
            ([ex (lambda-d (exp) ()
           (if-d-e (symbol? exp)
                       (call-d-u get exp)
                   (if-d-e (constant? exp)
                       (lift (arg1 exp))
           (if-d-e (null?? exp)
                       (base-d null? (call-d-u ex (arg1 exp)))
                   (if-d-e (-? exp)
                       (base-d - (call-d-u ex (arg1 exp))
                                 (call-d-u ex (arg2 exp)))
           (if-d-e (+? exp)
                       (base-d + (call-d-u ex (arg1 exp))
                                 (call-d-u ex (arg2 exp)))
                   (if-d-e (<? exp)
                       (base-d < (call-d-u ex (arg1 exp))
                                 (call-d-u ex (arg2 exp)))
                   (if-d-e (if? exp)
                       (if-d (call-d-u ex (arg1 exp))
                             (call-d-u ex (arg2 exp))
                             (call-d-u ex (arg3 exp)))
                       (let ([fun (assoc (car exp) program)])
                    (call-d exec (caddr fun)
                                        (cadr fun)
                                        (call-d-u ex* (cdr exp))))
                                                       ))))))))]
        [ex* (lambda-d (exp) ()
           (if (null? exp) (lift '())
                (base-d cons (call-d-u ex (car exp)) (call-d-u ex* (cdr exp)))))]
           [get (lambda-d (name) () (call-d-u lookup name names values))]
           [lookup (lambda-d (name names) (values)
               (if (null? names) (error 'lookup "can't find ~s" name)
               (if (eq? name (car names)) (base-d car values)
                   (call-d-u lookup name (cdr names) (base-d cdr values)))))])
           (call-d-u ex exp)))])
     (call-d exec (caddar program) (cadar program) input))))

;------------------------------------------------------------------------
; Auxiliary functions

(define (constant? e) (and (pair? e) (eq? (car e) 'quote)))
(define (base? e) (and (pair? e) (eq? (car e) 'base)))
(define (null?? e) (and (pair? e) (eq? (car e) 'null?)))
(define (car? e) (and (pair? e) (eq? (car e) 'car)))
(define (cdr? e) (and (pair? e) (eq? (car e) 'cdr)))
(define (cons? e) (and (pair? e) (eq? (car e) 'cons)))
(define (if? e) (and (pair? e) (eq? (car e) 'if)))
(define (call? e) (and (pair? e) (eq? (car e) 'call)))
(define (-? e) (and (pair? e) (eq? (car e) '-)))
(define (+? e) (and (pair? e) (eq? (car e) '+)))
(define (<? e) (and (pair? e) (eq? (car e) '<)))
(define arg1 cadr)
(define arg2 caddr)
(define arg3 cadddr)
```

The result of compiling the fibonacci progam can be seen below.

```
(
   (fib (n) (if (< n '2) '1
                (+ (fib (- n '1)) (fib (- n '2)))))
)

(lambda (input)
   (letrec
        ((exec-5
             (lambda (values-5)
                (if (< (car values-5) '2)
                    '1
                    (+ (exec-5 (cons (- (car values-5) '1) '()))
                       (exec-5 (cons (- (car values-5) '2) '())))))))
      (exec-5 input)))
```