Type Specialisation of a Subset of Haskell

Per Sjörs
URL: http://www.mdstud.chalmers.se/∼md3perra.

June 16, 1997

Abstract

John Hughes presents a new method for performing partial evaluation in [Hug96b]. The method is called type specialisation and functions much like a type checker. It infers its results. The static content of every sub-expression of the source program is derived. By letting this static content be the type of the expression, new specialised types are produced. The types are then propagated through the specialisation process independently of how code is residualised. This enables strong specialisations.

This paper describes an implementation with a subset of Haskell as both the source and the residual (target) language. It is capable of handling Haskell's data types, including specialising constructors.

One problem with Hughes' specialiser was that it could not handle static tuples and projections on them properly. Here, a solution is presented; a post-phase called projection unfolding. The method is capable of removing all static tuples provided neither the type of the program nor a sub-type of the type of the program is a static tuple.

The specialiser uses a monad with backtracking capabilities for the handling of polyvariance. It has been an open question whether this is an unreasonably costy solution. This paper describes a general method for computing the ratio between the effort spent in the succeeding branch and the total effort, including that spent in the failing branches, when finding the solution in a backtracking computation. This is actually a kind of state with limited capabilities which is global to the entire computation. Examples in this paper show that little effort is spent in the failing branches.

Contents

1 Introduction

Consider the function

$$\mathit{power} \ = \lambda x\ n \rightarrow \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ x \times \mathit{power}\ x\ (n-1)$$

This is one definition of the power function calculating x^n. We can use it to calculate the cube of a number by supplying the value 3 as argument for n. We can also use it for calculating the square by supplying 2. To calculate the cube we could also use the more efficient definition

$$\mathit{power}_3 \ = \lambda x \rightarrow x \times x \times x$$

In this second definition the fact that x should be multiplied 3 times is built into the code rather than supplied as an argument. We could not , however, use this second definition for calculating the square. To calculate the square in the more efficient manner would call for another definition.

What we have here is a contradiction between generality and efficiency. The solution to the problem is *partial evaluation* [JGS93] [BEJ88], which is also called *program specialisation.* This is a method which automatically *specialises* a *general* program, by exploiting known information, to a less general and hopefully more efficient one.

By specialising the function *power* with respect to the information that n is 3, we would automatically obtain the definition of the function power_3. This enables us to obtain efficient functions for calculating x^n for any n, but still only having to write one definition. We will see how this is done using our specialiser in section 4.1.

If we on the other hand know the value of x to be 4 while n is unknown, it is not possible to make the same simplification. The only rewriting we can do is to[1]

$$\mathit{power}_4 = \lambda n \rightarrow \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ 4 \times \mathit{power}_4\ (n-1)$$

This definition is only slightly more efficient.

If some of the input data to a program is known, we can use this information to produce a new program. The difference in efficiency between the new program and the original is determined by how this information is used within the program. In the above example, n determines the depth of the recursion, i.e. how many times *power* should call itself, while x is just an operand to the multiplication and not involved in controlling the program flow.

In our terminology the program, or function, *power* is said to be the *source code* while the specialised function power_3 is said to be the *residual code.*

1.1 Binding-Times

In the first example, n is said to be *static*, which means that it is known at specialisation time, while x, being unknown, is said to be *dynamic.* This is called their respective *binding-times.* In fact every syntactic element of the source code has a binding-time. This includes function application, functions, multiplication etc. The specialiser must be aware of the binding-times to be able to know what to do. In this

[1] This simplification corresponds to constant folding.

paper we only consider *off-line* partial evaluation, which means that the binding-times are determined in advance. This can be done automatically in a *binding-time analysis* or manually through *annotations*. We only consider the latter. Dynamic objects are annotated by means of underlining.

1.2 The *mix*-Equations

In the examples above the source and residual code are in the same language. This is not necessary. They might just as well be in different.

Call the input language, the *source language*, S, and the output, the *target language*, T. Call the language that the partial evaluator itself is written in the *object language*, L. Let $[\![\cdot]\!]_L$ be the denotational function for an L-program and *mix* be a partial evaluator.

We are now able to define the behaviour of *mix*.

$$[\![mix]\!]_L \; p \; s = p_s$$
$$[\![p_s]\!]_T \; d = [\![p]\!]_S \; s \; d$$

Specialising a program p with respect to static input s results in a program p_s which when run on the remaining input d should have the same behaviour as the original program run on s and d.

We can define an interpreter, *int*, for the language U written in L by

$$[\![int]\!]_S \; p \; d = [\![p]\!]_U \; d$$

Interpreting the program p with input data d should have the same behaviour as the meaning of p with d as input data. The interpreter is a function which takes two arguments, the program to interpret and the input data to it. Specialise the interpreter with respect to the program

$$[\![mix]\!]_L \; int \, p = int_p$$
$$[\![int_p]\!]_T \; d = [\![int]\!]_S \; p \; d$$

According to the definition of the interpreter it must follow that

$$[\![p]\!]_U = [\![int_p]\!]_T$$

which means that p and the specialised interpreter has the same behaviour but are written in different languages. This is recognised as compilation. In this case from the language U to L. The equation

$$[\![mix]\!]_L \; int \, p = int_p$$

is called *the first Futamura projection*[2][Fut71].

In addition to making more efficient specialised programs, this is one motivation for partial evaluation.

[2] There is also a second and a third Futamura projection but they will not be discussed here.

1.3 Traditional Partial Evaluation

Most efforts in the field of partial evaluation have so far concerned a method which hereinafter will be referred to as the *interpretive style*. Its principle resembles much that of an interpreter when dealing with static data. In fact, its behaviour could informally be described as just copying dynamic constructions and evaluating static ones in the same fashion as an interpreter. Recall the definition of the *power* function. It is now annotated.

$$power = \lambda x\ n \rightarrow \textbf{if}\ n = 0\ \textbf{then}\ \underline{1}\ \textbf{else}\ x \underline{\times} (power\ (n-1)\ x)$$

To begin with, as n is static, the result of the conditional depends solely on known data and is therefore also static along with the whole **if**-expression. We always know which branch to choose. As the depth of the recursion solely depends on n, the recursive call to *power* can be unfolded, i.e. it is considered static. It is only the multiplication that cannot be evaluated because its left hand operand is not known. The 1 will eventually be the right hand argument to a dynamic multiplication and should therefore also be dynamic. The function *power* and the application of it is static and are thus unfolded in much the same way as an interpreter evaluates a call. The function body is evaluated in an environment which is extended with bindings for the λ-abstracted variables. The $x\times$ expression will be copied and so will the 1. This results in the residual program

$$power' = \lambda x \rightarrow x \times x \times \ldots \times x \times 1 \qquad \textit{(with x appearing n times)}$$

1.4 Our Specialiser

John Hughes adopted a different approach to the problem of partial evaluation. The principles for this method were first presented in [Hug96b].[3] An implementation for an extended λ-calculus was also described there. The specialiser described in this paper is based on that principle.

One of the obstacles, which was mentioned as left for future work, with Hughes specialiser was the handling of static tuples and projections on them. They where simply left in the code. Here we present a solution to that problem; a post-phase called *projection unfolding*.

2 Theory

One big drawback with the traditional method is that it does not handle types. Thus it handles typed programs as though they were untyped. The method used in this paper *does* handle types.

The method is based on inference and functions to a great extent the same way as a type-checker. The type-checker infers the type of an expression. Our method infers the type of the specialised expression, the *residual type*, and it also infers the residual code itself. The type of the source code is called the *source type*. As the source and target language does not need to be the same, of course that accounts for their type systems as well. Actually, we will see that the types too are specialised.

[3] For an introduction see [Hug96a].

The method will be referred to as the *inference style*.

2.1 Program Structure

The partial evaluator constructed can be viewed as built up by a number of functions applied in turn. These functions will be discussed in the same order as they are applied.

The *parser*[4] [Hut92] builds a two level syntax tree which is passed to the *type-checker* [Mil78] [PJ87]. If it succeeds, the same syntax tree is passed to the *specialiser*. The type specialisation gives rise to trivial values in the code, they are removed by the *void erasure*. In addition static tuples are left in the code, these are handled by the *projection unfolding*. A *pretty-printer* [Hug95] is finally applied to make the representation into text again.

2.2 Grammar

The language chosen for our partial evaluator is Haskell [Hud92], although not the entire language. The target (residual) language is a true subset of Haskell. This subset is extended to form the source language. The extensions are the following. As a binding-time analysis is not provided, the user must annotate the binding-times. Therefore annotations are added. A Haskell compiler performs an analysis which divides the declarations in a **let**-expression into strongly connected components, transforming it into potentially many nested **let**/**letrec**-expressions. This analysis is not provided in our partial evaluator and must be done by the user. The keyword **letrec** is added. **lift** and **lifts** are two operators that turn static integer values and static string values into their dynamic correspondences. As will be seen below, we need an explicit way of indicating polyvariance, which is done by using **poly** and **spec**.

The grammar for the source language is shown in Figure 1. The underlining means that the construction is dynamic. Dynamic tuples are written in the usual way, $(e_1, \ldots, e_n)$ while $<e_1, \ldots, e_n>$ stands for a static tuple. The notation $[x_i]_{i=1}^{n}$ is an abbreviation for $[\ x_1 \ldots\ x_n]$. The source types, shown in figure 2, follows the constructions in the grammar. As all code constructs have two forms, types should too. Again underlining means dynamic. For example **int** stands for static integer while $\underline{\textbf{int}}$ stands for dynamic integer. D stands for a data type. If a function of type τ is prefixed by the operator **poly**, the type will become **poly** τ . If the 'super constructor' *In* is applied to an expression of type τ , the type of the entire expression is **sum** τ.

2.3 Type Checker

We are concerned with typed programs and thus the input to the specialiser has a source type. Type errors in the source type can make the specialiser 'go wrong', the source code must therefore be type checked. This is an ordinary type check, similar to that performed in a compiler. The one difference is that the source code here is

[4] The parser was built using a library of parser combinators constructed by Rogardt Heldal. The parser is mainly written by Lars Pareto.

$$
\begin{array}{rcl}
e & ::= & e\ e \mid \lambda x \to e \\
& \mid & \mathbf{let}\ [x = e]_1^n\ \mathbf{in}\ e \mid \mathbf{letrec}\ [x = e]_1^n\ \mathbf{in}\ e \\
& \mid & \mathbf{case}\ e\ \mathbf{of}\ [p \to e] \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \\
& \mid & e \oplus e \\
& \mid & C\ e \ldots e \mid {<}e, \ldots, e{>} \\
& \mid & x \mid n \mid s \\
& \mid & e\,\underline{@}\,e \mid \underline{\lambda} x \to e \\
& \mid & \underline{\mathbf{let}}\ [x = e]_1^n\ \underline{\mathbf{in}}\ e \mid \underline{\mathbf{letrec}}\ [x = e]_1^n\ \underline{\mathbf{in}}\ e \\
& \mid & \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ [p \to e]_1^n \mid \underline{\mathbf{if}}\ e\ \underline{\mathbf{then}}\ e\ \underline{\mathbf{else}}\ e \\
& \mid & e\underline{\oplus}e \\
& \mid & \underline{C}\ e \ldots e \mid \underline{In}\ e \mid (e, \ldots, e) \\
& \mid & \mathbf{lift}\ e \mid \mathbf{lifts}\ e \mid \mathbf{poly}\ e \mid \mathbf{spec}\ e \\
\\
p & ::= & C\ p \ldots p \mid {<}p, \ldots, p{>} \\
& \mid & x \mid n \mid s \mid \_ \\
& \mid & \underline{C}\ p \ldots p \mid (p, \ldots, p) \\
& \mid & \underline{In}\ p
\end{array}
$$

Figure 1: Grammar of source language.

$$
\begin{array}{rcl}
\tau & ::= & \tau \to \tau \\
& \mid & D\ \tau \ldots \tau \mid {<}\tau, \ldots, \tau{>} \\
& \mid & \mathbf{int} \mid \mathbf{string} \\
& \mid & \tau\ \underline{\to}\ \tau \\
& \mid & \underline{D}\ \tau \ldots \tau \mid (\tau, \ldots, \tau) \\
& \mid & \underline{\mathbf{int}} \mid \underline{\mathbf{string}} \\
& \mid & \mathbf{poly}\ \tau \mid \mathbf{sum}\ \tau
\end{array}
$$

Figure 2: Source type system.

annotated and thus represented by a two level language. The type checker must be able to handle this. One elegant thing appears here, we obtain a binding time checker for free! For example, we should provide a type inference rule for static addition of two static integers and another for dynamic addition of two dynamic integers. When it comes to integer additions, no additional rules should be provided. The result is that when mixing binding times in an inappropriate way, the type checker will report an error.

Rules are stated for expressions but we must also handle the patterns in **case**-expressions. The patterns should be given the same type as the expression which we case upon. Therefore define $p \Leftarrow \tau$ to have the meaning to bind the type τ to the pattern p and construct the *least* Γ such that $\Gamma \vdash p : \tau$. Note that it is possible that no such *Gamma* exists. The rule is then considered not applicable.

Rules are stated for expressions and also for patterns. When checking a **case**-expression we must of course derive the types of the patterns which should be the same as for the expression upon which we case. The rules for expressions and patterns are basically the same, though much fewer for patterns.

The rules are presented in figures 3, 5 and 4 without any further presentations. They should almost be self explanatory.

Up to now we have presented a monomorphic type-checker where τ have denoted types but to allow declarations to have polymorphic types we need *type schemes*. Call these σ.

$$\sigma ::= \tau \mid \forall \alpha . \sigma$$

We must also allow type variables in types and extend the syntax for them.

$$\tau ::= \ldots \mid \alpha$$

Furthermore we need to add the following rules

$$\textit{(INST)} \quad \frac{\Gamma \vdash e : \forall \alpha . \sigma}{\Gamma \vdash e : \sigma[\tau / \alpha]}$$

$$\textit{(GEN)} \quad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha . \sigma} \quad \alpha \; not \; free \; in \; \Gamma$$

The generalisation rule, *(GEN)*, has the meaning that if a type scheme is derived for an expression, for example $\Gamma \models \lambda x \rightarrow x : \alpha \rightarrow \alpha$, it is valid for all different instantiations of a certain type variable, which means for all α in the example. The example is generalised to $\Gamma \models \lambda x \rightarrow x : \forall \alpha . \alpha \rightarrow \alpha$. There is one restriction, there must not be any assumption on the type variable that is quantified in Γ. It must not occur free in Γ.

The instantiation rule states that a type variable which is universally quantified can be instantiated to any type. This means that the entire type expression becomes less general. For example

$$\begin{array}{ll} \textbf{let} & f = \lambda x \rightarrow x \\ \textbf{in} & (f \; 1, f \; \mathit{True}) \end{array}$$

$$\textit{(DAPP)}\qquad \frac{\Gamma \vdash e : \sigma \underline{\to} \tau \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash e \,\underline{@}\, e' : \tau}$$

$$\textit{(DLAM)}\qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \underline{\lambda} x \to e : \sigma \underline{\to} \tau}$$

$$\textit{(DCASE)}\qquad \frac{\Gamma \vdash e_0 : \tau_0 \qquad [\Gamma, p_i \Leftarrow \tau_0 \vdash e_i : \tau]_{i=1}^{n}}{\Gamma \vdash \underline{\mathbf{case}}\ e_0\ \underline{\mathbf{of}}\ [p_i \to e_i]_{i=1}^{n} : \tau}$$

$$\textit{(DTUP)}\qquad \frac{[\Gamma \vdash e_i : \tau_i]_{i=1}^{n}}{\Gamma \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)}$$

$$\textit{(DCON)}\qquad \frac{[\Gamma, (C : \ldots) \vdash e_i : \tau_i]_{i=1}^{n}}{\Gamma, C : \tau_1 \to \cdots \to \tau_n \to \underline{D}\ \tau_{k_1} \ldots \tau_{k_m} \vdash \underline{C}\ e_1 \ldots e_n : \underline{D}\ \tau_{k_1} \ldots \tau_{k_m}}$$

$$\textit{(DIF)}\qquad \frac{\Gamma \vdash e_0 : \underline{\mathbf{bool}} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \underline{\mathbf{if}}\ e_0\ \underline{\mathbf{then}}\ e_1\ \underline{\mathbf{else}}\ e_2 : \tau}$$

$$\textit{(DPRIM)}\qquad \frac{\Gamma \vdash e_1 : \underline{\mathbf{int}} \qquad \Gamma \vdash e_2 : \underline{\mathbf{int}}}{\Gamma \vdash e_1 \underline{\oplus} e_2 : \underline{\mathbf{int}}} \qquad \textit{where } \underline{\oplus} \in \{\underline{+},\ \underline{\times},\ \underline{-},\ \underline{/}\}$$

$$\textit{(LIFT Int)}\qquad \frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash \mathbf{lift}\ e : \underline{\mathbf{int}}}$$

$$\textit{(LIFT String)}\qquad \frac{\Gamma \vdash e : \mathbf{string}}{\Gamma \vdash \mathbf{lifts}\ e : \underline{\mathbf{string}}}$$

Figure 3: Type checking rules for dynamic constructs.

$$\textit{(POLY)}\qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau}$$

$$\textit{(SPEC)}\qquad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau}{\Gamma \vdash \mathbf{spec}\ e : \tau}$$

$$\textit{(IN)}\qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathit{In}\ e : \mathbf{sum}\ \tau}$$

$$\textit{(INCASE)}\qquad \frac{\Gamma \vdash e : \mathbf{sum}\ \tau \qquad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \mathit{In}\ x \to e' : \tau'}$$

Figure 4: Type checking rules for polyvariant constructs.

$$\textit{(SAPP)} \quad \frac{\Gamma \vdash e : \sigma \to \tau \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash e e' : \tau}$$

$$\textit{(SLAM)} \quad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x \to e : \sigma \to \tau}$$

$$\textit{(SCASE)} \quad \frac{\Gamma \vdash e_0 : \tau_0 \qquad [\Gamma, p_i \Leftarrow \tau_0 \vdash e_i : \tau]_{i=1}^{n}}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ [p_i \to e_i]_{i=1}^{n} : \tau}$$

$$\textit{(STUP)} \quad \frac{[\Gamma \vdash e_i : \tau_i]_{i=1}^{n}}{\Gamma \vdash <e_1, \ldots, e_n> : <\tau_1, \ldots, \tau_n>}$$

$$\textit{(SCON)} \quad \frac{[\Gamma, (C : \ldots) \vdash e_i : \tau_i]_{i=1}^{n}}{\Gamma, C : \tau_1 \to \cdots \to \tau_n \to D\ \tau_{k_1} \tau_{k_m} \vdash C\ e_1 \ldots e_n : D\ \tau_{k_1} \ldots \tau_{k_m}}$$

$$\textit{(SIF)} \quad \frac{\Gamma \vdash e_0 : \mathbf{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}$$

$$\textit{(SPRIM)} \quad \frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 \oplus e_2 : \mathbf{int}} \qquad \textit{where} \oplus \in \{+,\ \times,\ -,\ /\}$$

$$\textit{(VAR)} \quad \Gamma, x : \tau \vdash x : \tau$$

$$\textit{(INT)} \quad \Gamma \vdash n : \mathbf{int}$$

$$\textit{(STR)} \quad \Gamma \vdash s : \mathbf{string}$$

Figure 5: Type checking rules for static constructs.

$$\textit{(DLET)} \quad \frac{[\Gamma \vdash e_i : \sigma_i]_{i=1}^{n} \qquad \Gamma, [x_i : \sigma_i]_{i=1}^{n} \vdash e_0 : \tau}{\Gamma \vdash \underline{\textbf{let}}\ [x_i = e_i]_{i=1}^{n}\ \underline{\textbf{in}}\ e_0 : \tau}$$

$$\textit{(DLETREC)} \quad \frac{[\Gamma, [x_i : \tau_i]_{i=1}^{n} \vdash e_j : \tau_j]_{j=1}^{n} \qquad \Gamma, [x_i : \sigma_i]_{i=1}^{n} \vdash e_0 : \tau}{\Gamma \vdash \underline{\textbf{letrec}}\ [x_i = e_i]_{i=1}^{n}\ \underline{\textbf{in}}\ e_0 : \tau}$$

where σ_i is a generalization of the type scheme τ_i

$$\textit{(SLET)} \quad \frac{[\Gamma \vdash e_i : \sigma_i]_{i=1}^{n} \qquad \Gamma, [x_i : \sigma_i]_{i=1}^{n} \vdash e_0 : \tau}{\Gamma \vdash \textbf{let}\ [x_i = e_i]_{i=1}^{n}\ \textbf{in}\ e_0 : \tau}$$

$$\textit{(SLETREC)} \quad \frac{[\Gamma, [x_i : \sigma_i]_{i=1}^{n} \vdash e_j : \sigma_j]_{j=1}^{n} \qquad \Gamma, [x_i : \sigma_i]_{i=1}^{n} \vdash e_0 : \tau}{\Gamma \vdash \textbf{letrec}\ [x_i = e_i]_{i=1}^{n}\ \textbf{in}\ e_0 : \tau}$$

Figure 6: Type checking rules for **let**- and **letrec**-expressions.

The type scheme $\alpha \to \alpha$ is derived for $\lambda x \to x$. It is generalised to $\forall \alpha . \alpha \to \alpha$, which will be the type scheme of f. At the left application of f α is instantiated to **int**, resulting in the type **int** → **int**, and in the right application, a is instantiated to **bool**.

We are now ready to give rules for **let**/ **letrec**-expressions. This is done in figure 6. Note that the variables of the declarations have polymorphic types as they are given type schemes.

2.4 Specialiser

The role of the specialiser is to derive the residual code and residual type of the source expressions. We consider these two concepts as representing the dynamic respectively the static content of the source code. Dynamic constructs are supposed to appear in the code while static are not. On the other hand, static information should not disappear from the specialisation process, because if it does, important information is lost which could affect the process later on. Static information is therefore propagated through the residual types.

The specialiser is specified by a set of inference rules. The judgements are of the form

$$\Gamma \mid\!\vdash e \hookrightarrow e' : \tau$$

which means that the expression e is specialised to the residual expression e' having residual type τ. This is done in the environment Γ which holds bindings for the free variables of e. The bindings have the form $x \hookrightarrow e : \tau$.

Let us look at a simple example just to grasp the idea.

$$\begin{array}{lll} \vdash & \underline{\textbf{let}} \quad f = \underline{\lambda} x \to \textbf{lift}\ x & \hookrightarrow \quad \textbf{let} \quad f = \lambda x \to 1 \quad : \textbf{int} \\ & \underline{\textbf{in}} \quad\ f\ 1 & \qquad\ \ \textbf{in} \quad\ f\ \bullet \end{array}$$

The program is a dynamic **let**-expression. To specialise it, first specialise the declaration ($\underline{\lambda} x \rightarrow \textbf{lift}\ x$) and bind the result ($\lambda x \rightarrow 1 : \mathbf{1} \rightarrow \textbf{int}$) to its original name ($f$). This binding will function as an environment ($[f \hookrightarrow f : \mathbf{1} \rightarrow \textbf{int}]$) when specialising the top level expression ($f\ 1$) to its residual correspondence ($f\ \bullet : \textbf{int}$). The **let**-expression is rebuilt with the new specialised expression.

What is this notion of a residual type? It should be able to express the static content of an expression. This amounts to entirely static expressions as well as entirely dynamic. It must therefore possess the capability of representing every value of the language respectively every type. The static content of a static integer is its value, the integer value itself. The static content of a dynamic integer is its type, the fact that it is an integer. For these reasons it seems natural that every value of all base types and every source type also should be a residual type.

So what should the result be of specialising the static expression 1? In terms of static and dynamic content, the static content is of course 1 and as the expression is entirely static the dynamic content is null. Every specialisation must however have residual code, as well as residual type, as a result. Therefore, as a kind of placeholder, let the code be $\bullet$. We call this value *void*.

This can be explained in another way. For every static value having a base type (integers and strings), construct a new type with exactly one element. Name the type after the value and let its only element be void. The element of a one point domain is $\perp$ but by assuming lazy semantics this causes no problem.

What about the residual type when considering complex (tuples, functions etc.) and partially static expressions?

Now that we can find the residual type of both static and dynamic expressions it is easy to see that partially static expressions are typed by mixing the different types we have come up with. For example, the expression $(1, \textbf{lift}\ 2)$ is a dynamic pair with the elements a static 1 and a dynamic 2. The content is a pair of the value $\bullet$, with residual type $\mathbf{1}$, and the value 2, with residual type **int**. Resulting in the specialisation

$$\vdash (1, \textbf{lift}\ 2) \hookrightarrow (\bullet, 2) : (\mathbf{1}, \textbf{int})$$

We can now give the grammar and the type system for the residual (target) language. These are shown in figures 7 and 8. Note that there is a new construct in the grammar, π_i. This is static projection. It was mentioned above that the specialiser is not capable of removing static tuples and projections on them and there must therefore be syntax for these. The constructor *In* has recieved an index. This is, as we will see, because it is specialised and one occurence of it might be responsible of many occurrence in the residual code.

2.4.1 Base Types and Variables

We can now state the first inference rules, the ones for static integers and static strings.

$$\textit{(SINT)} \quad \Gamma \models n \hookrightarrow \bullet : \mathbf{n}$$

$$\textit{(SSTR)} \quad \Gamma \models s \hookrightarrow \bullet : \mathbf{s}$$

$$
\begin{array}{rcl}
e & ::= & e\ e \mid \lambda x \rightarrow e \\
 & \mid & \textbf{let}\ [x = e]_1^n\ \textbf{in}\ e \\
 & \mid & \textbf{case}\ e\ \textbf{of}\ [p \rightarrow e] \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
 & \mid & e \oplus e \\
 & \mid & C\ e \ldots e \mid In_i\ e \ldots\ e \mid <e, \ldots, e> \mid \pi_i\ e \mid (e, \ldots, e) \\
 & \mid & x \mid n \mid s \mid \bullet
\end{array}
$$

$$
\begin{array}{rcl}
p & ::= & C\ p \ldots p \mid In_i\ e \ldots\ e \mid <p, \ldots, p> \mid (p, \ldots, p) \\
 & \mid & x \mid n \mid s \mid\ \bullet\ \mid \_
\end{array}
$$

Figure 7: Grammar of the residual language.

$$
\begin{array}{rcl}
\tau & ::= & \tau \rightarrow \tau \mid \sum\ In\ \tau \ldots \tau \\
 & \mid & D\ \tau \ldots \tau \mid C\ \tau \ldots \tau \mid <\tau, \ldots, \tau> \mid (\tau, \ldots, \tau) \\
 & \mid & \textbf{int} \mid \textbf{string} \mid \mathbf{n} \mid \mathbf{s} \mid \alpha \mid \bullet \\
 & \mid & \textbf{clos}\langle <x, \ldots, x>, <\tau, \ldots, \tau>, x, e\rangle \\
 & \mid & \textbf{rec}\langle x, <x, \ldots, x>, <\tau, \ldots, \tau>, [(x, x, e)]_1^n\rangle
\end{array}
$$

Figure 8: Type system for residual language.

There is no way of expressing dynamic integers or strings directly, instead we use the operators **lift** and **lifts**. They turn a static object into its dynamic correspondence. There is one rule for each operator, one for integers and one for strings.

$$
\textit{(LIFT}_{\textit{Int}}\textit{)} \qquad \frac{\Gamma \models e \hookrightarrow e' : n}{\Gamma \models \textbf{lift}\ e \hookrightarrow n : \textbf{int}}
$$

$$
\textit{(LIFT}_{\textit{String}}\textit{)} \qquad \frac{\Gamma \models e \hookrightarrow e' : s}{\Gamma \models \textbf{lifts}\ e \hookrightarrow s : \textbf{string}}
$$

Static and dynamic primitive operations can only be performed on static and dynamic values of base types respectively. In the static case the actual operation is performed while in the dynamic the two arguments are specialised and the operator is residualised along with these two specialisations.

$$
\textit{(S}\oplus\textit{)} \qquad \frac{\Gamma \models e_1 : \tau \hookrightarrow e_1' : v_1 \quad \Gamma \models e_2 : \tau \hookrightarrow e_2' : v_2 \quad v_1 \oplus v_2 = v}{\Gamma \models e_1 \oplus e_2 : \sigma \hookrightarrow \bullet : v}
$$

$$
\textit{(D}\oplus\textit{)} \qquad \frac{\Gamma \models e_1 : \underline{\tau} \hookrightarrow e_1' : \tau \quad \Gamma \models e_2 : \underline{\tau} \hookrightarrow e_2' : \tau}{\Gamma \models e_1 \underline{\oplus} e_2 : \underline{\sigma} \hookrightarrow e_1' \oplus e_2' : \sigma}
$$

$$
\begin{array}{rl}
\textit{where} & \tau = \sigma = \textbf{int},\ \oplus \in \{+,\ \times,\ -,\ /\} \\
\textit{or} & \tau \in \{\textbf{int},\ \textbf{string}\},\ \sigma = \textbf{bool},\ \oplus \in \{==,\ /=,\ <,\ <=,\ >,\ >=\} \\
\textit{or} & \tau = \sigma = \textbf{bool},\ \oplus \in \{\&\&,\ ||\}
\end{array}
$$

Γ is the environment holding bindings for the free variables of the expression to be specialised. Specialising a variable amounts to looking up the variable in the environment.

$$\textit{(VAR)} \quad \Gamma, x \hookrightarrow e : \tau \vdash x \hookrightarrow e : \tau$$

2.4.2 Tuples

The rule for dynamic tuples is quite straightforward. Simply specialise the elements and reinstall the residual expressions into the tuple. Note that we are free to mix dynamic and static expressions as elements of tuples.

$$\textit{(DTUP)} \quad \frac{[\Gamma \vdash e_i \hookrightarrow e_i' : \tau_i']_{i=1}^{n}}{\Gamma \vdash (e_1, \ldots, e_n) \hookrightarrow (e_1', \ldots, e_n') : (\tau_1', \ldots, \tau_n')}$$

A static construct is supposed to disappear from the code. The same goes for static tuples. For not obvious reasons, the handling of static tuples is so complicated that it is postponed to a separate phase called projection unfolding. (See section 2.6 below.) A rule for handling static tuples must however be provided. It works by residualising the tuple, leaving it to the post-phase.

$$\textit{(STUP)} \quad \frac{[\Gamma \vdash e_i \hookrightarrow e_i' : \tau_i']_{i=1}^{n}}{\Gamma \vdash <e_1, \ldots, e_n> \hookrightarrow <e_1', \ldots, e_n'> : <\tau_1', \ldots, \tau_n'>}$$

2.4.3 Functions and Function Applications

Specialising a dynamic λ-expression amounts to specialising its body in an environment extended with the variable bound by the λ. There is a risk for name capture. It is possible that a dynamic λ-expression will give rise to two nested λ-expressions. A fresh variable name is therefore used.

$$\textit{(DLAM)} \quad \frac{\Gamma, x \hookrightarrow x' : \sigma' \vdash e \hookrightarrow e' : \tau'}{\Gamma \vdash \underline{\lambda} x \rightarrow e \hookrightarrow \lambda x' \rightarrow e' : \sigma' \rightarrow \tau'} \quad x' \textit{ fresh}$$

The rule for dynamic application is the simple

$$\textit{(DAPP)} \quad \frac{\Gamma \vdash e_1 \hookrightarrow e_1' : \sigma' \rightarrow \tau' \qquad \Gamma \vdash e_2 \hookrightarrow e_2' : \sigma'}{\Gamma \vdash e_1 \underline{@} e_2 \hookrightarrow e_1' \; e_2' : \tau'}$$

When specialising a static function, we know that the application of the function will eventually be performed but not where or when. Thus the function must be specialised into such a form that when the application is actually to be performed, it is possible to recall the source definition of the function in its whole and specialise it in place with its lambda bound variable bound to the argument of the application. The environment in scope where the function is defined is not necessarily the same as that in scope where the application is defined. The specialisation should take place in the former. Therefore, the environment must also be retrievable along with the function definition. The rule for static functions just saves the function in a form known as its *closure*.

The result of the specialisation must still be represented in the form of residual code and type. The static and dynamic parts of the closure must be separated. Recalling the form of the environment, one entry has the form

$$x \hookrightarrow e : \tau$$

Here, dynamic and static data are intermingled. The residual expression e is of course dynamic, while the residual type τ is static. x is also considered static, it is after all not supposed to appear in the residual program. The same goes for the function itself. The rule then becomes

$$\begin{array}{ll} \textit{(SLAM)} & [x_i \hookrightarrow e_i : \tau_i]_{i=1}^{n} \ \vdash \\ & \quad \lambda x \to e \ \hookrightarrow \\ & \qquad <e_{k_1}, \ldots, e_{k_m}> : \textbf{clos}\langle <x_{k_1}, \ldots, x_{k_m}>, <\tau_{k_1}, \ldots, \tau_{k_m}>, x, e \rangle \end{array}$$

$$\textit{where } x_{k_i} \textit{ occurs free in } \lambda x \to e$$

Only the elements of the environment with xs that occurs free in the λ-expression needs to be stored. When the environment is rebuilt, only those could possibly be looked up in it. The residual expressions from the environment, the es, are statically tupled to form the residual code, while the variable names, the xs, and the types, the τs, receive the same treatment but are put into the residual type instead. The function definition is just copied into the type.

The rule for static application is

$$\textit{(SAPP)} \quad \frac{\begin{array}{c} \Gamma \vdash e_1 \hookrightarrow e_1' : \textbf{clos}\langle <x_1, \ldots, x_n>, <\tau_1, \ldots, \tau_n>, x, e \rangle \\ \Gamma \vdash e_2 \hookrightarrow e_2' : \sigma' \\ [x_i \hookrightarrow \pi_i\ e_1' : \tau_i]_{i=1}^{n}, x \hookrightarrow e_2' : \sigma' \vdash e \hookrightarrow e' : \tau' \end{array}}{\Gamma \vdash e_1\ e_2 \hookrightarrow e' : \tau'}$$

where the environment is rebuilt from the residual code and type of the specialised function. The body of the function is retrieved from the residual type of the function and specialised in the rebuilt environment appended with the lambda bound variable of the function bound to the specialisation of the argument in the application. Note the usage of static tuples, which is the residual type of the function. Projections are used in the reconstruction of the environment. An example might elucidate.

$$\begin{array}{llll}
\vdash & \textbf{let} & a & = \textbf{lift}\ 1 \\
& & b & = \textbf{lift}\ 2 \\
& & c & = \textbf{lift}\ 3 \\
& \textbf{in} & \underline{\textbf{let}} & f = \lambda x \to x \underline{+} a \underline{+} b \\
& & \underline{\textbf{in}} & \quad \underline{\textbf{let}} \quad a = \textbf{lift}\ 4 \\
& & & \quad \underline{\textbf{in}} \quad\ \ f\ a
\end{array}
\qquad
\begin{array}{lllll}
\hookrightarrow & \textbf{let} & f = <1, 2> & & \qquad : \textbf{int} \\
& \textbf{in} & \textbf{let} & a = 4 & \\
& & \textbf{in} & a + \pi_1 f + \pi_2 f &
\end{array}$$

where f is specialised to

$$f \hookrightarrow <1, 2> : \textbf{clos}\langle <a, b>, <\textbf{int}, \textbf{int}>, x, x + a + b \rangle$$

The environment in scope when the function is defined consists of the three variables a, b and c with specialisations $1 : \textbf{int}$, $2 : \textbf{int}$ and $3 : \textbf{int}$. c does not occur free in the definition of f and is therefore not included in the closure. When f is applied, there is another environment in scope with another binding for a. So when the first a is referenced in the function body, it has to be done through the closure by a projection on f while the second a was bound to x and replaced it in the function body.

2.4.4 Dynamic and Static let-Expressions

The rule for dynamic **let**-expressions specialises the expressions in the declarations first. As there is a risk for name capture, the variables declared must be replaced by fresh ones when specialising the top level expression.

$$\textit{(DLET)}\quad \frac{[\Gamma \vdash e_i \hookrightarrow e'_i : \tau'_i]_{i=1}^{n} \qquad \Gamma, [x_i \hookrightarrow x'_i : \tau'_i]_{i=1}^{n} \vdash e_0 \hookrightarrow e'_0 : \tau'_0}{\Gamma \vdash \underline{\textbf{let}}\ [x_i = e_i]_{i=1}^{n}\ \underline{\textbf{in}}\ e_0 \hookrightarrow \textbf{let}\ [x'_i = e'_i]_{i=1}^{n}\ \textbf{in}\ e'_0 : \tau'_0}$$

$$x'_i\ \textit{fresh}$$

When specialising static **let**-expressions there is no risk of name capture as the declared variables are bound to their specialised expressions when specialising the top level expression.

$$\textit{(SLET)}\quad \frac{[\Gamma \vdash e_i \hookrightarrow e'_i : \tau'_i]_{i=1}^{n} \qquad \Gamma, [x_i \hookrightarrow e'_i : \tau'_i]_{i=1}^{n} \vdash e_0 \hookrightarrow e'_0 : \tau'_0}{\Gamma \vdash \textbf{let}\ [x_i = e_i]_{i=1}^{n}\ \textbf{in}\ e_0 \hookrightarrow e'_0 : \tau'_0}$$

2.4.5 Dynamic letrec-Expressions

Dynamic **letrec**-expressions are specialised the same way as dynamic **let**-expressions with one difference. The bindings of the variables declared should not only be added to the environment when specialising the top level expression, the same extended environment should be used when specialising the expressions of the declarations.

$$\textit{(DLETREC)}\quad \frac{\begin{array}{c}[\Gamma, [x_j \hookrightarrow x'_j : \tau'_j]_{j=1}^{n} \vdash e_i \hookrightarrow e'_i : \tau'_i]_{i=1}^{n} \\ \Gamma, [x_i \hookrightarrow x'_i : \tau'_i]_{i=1}^{n} \vdash e_0 \hookrightarrow e'_0 : \tau'_0\end{array}}{\Gamma \vdash \underline{\textbf{letrec}}\ [x_i = e_i]_{i=1}^{n}\ \underline{\textbf{in}}\ e_0 \hookrightarrow \textbf{let}\ [x'_i = e'_i]_{i=1}^{n}\ \textbf{in}\ e'_0 : \tau'_0}$$

$$x'_i\ \textit{fresh}$$

Note that the keyword **let** is used in the residual code. The residual code should be possible to compile in an ordinary Haskell compiler.

2.4.6 Static letrec-Expressions

The only expressions that are possible to occur as the right hand sides of static **letrec** declarations are static functions. Constructs such as $ones = Cons\ 1\ ones$ cannot be declared in a static **letrec**, it would cause an infinite loop when trying to unfold. Trying to unfold a recursive dynamic function would also cause the specialiser loop.

To specialise a static **letrec**, specialise its top level expression in an environment which is the the current environment extended with bindings for the variables declared in the **letrec** in point. The variables should be bound to representations of their closures. We cannot however, use the same closures as we did for static functions described above, because now the functions are recursive. A recursive static function $\lambda x \rightarrow e$ would have the specialisation

$$\begin{array}{l} f \hookrightarrow e' : \tau \\ e' = {<}e_1, \ldots, e', \ldots, e_n{>} \\ \tau = \textbf{clos}\langle {<}x_1, \ldots, f, \ldots, x_n{>}, {<}\tau_1, \ldots, \tau, \ldots, \tau_n{>}, x, e \rangle \end{array}$$

Both the residual expression and type are infinite which we do not want to handle. We therefore introduce a new type, **rec**$\langle\ldots\rangle$, for typing recursive static functions which functions much like the **clos**$\langle\ldots\rangle$ type. It too, along with a static tuple of expressions, represents a closure.

Consider that there might be many declarations in the same **letrec** defining mutually recursive functions. Such a closure representation must therefore hold definitions for all functions declared in the same **letrec**. It must also hold bindings for the free variables of all the right hand sides which are not one of the declared functions.

Two variables declared in the same **letrec** should therefore have identical types. There is one small exception, the type should also contain the name of the variable it is the type of.

The k:th declaration, $f_k = \lambda x_k \to e_k$, of a **letrec** should therefore have the following binding in the environment in which the top level expression of the **letrec** is specialised.

$$f_k \hookrightarrow \texttt{<}e_1, \ldots, e_n\texttt{>} : \mathbf{rec}\langle f_k, \texttt{<}y_1, \ldots, y_n\texttt{>}, \texttt{<}\tau_1, \ldots, \tau_n\texttt{>}, [(f_i, x_i, e_i)]_{i=1}^{m}\ \texttt{>}\rangle$$

Here, the last component of **rec** is a list of representations of the declared functions and the three tuples together form the environment holding bindings for the variables not being one of those functions. The first component of **rec** is the variable it types. The rule becomes

$$\textit{(SLETREC)}\quad \frac{\begin{gathered}\left[\begin{array}{l}\Gamma \vdash \lambda x_i \to e_i \hookrightarrow \texttt{<}e_{i,1}, \ldots, e_{i,m_i}\texttt{>} \\ \quad : \mathbf{clos}\langle \texttt{<}y_{i,1}, \ldots, y_{i,m_i}\texttt{>}, \texttt{<}\tau_{i,1}, \ldots, \tau_{i,m_i}\texttt{>}, x_i, e_i\rangle\end{array}\right]_{i=1}^{n} \\ \Gamma, \left[\begin{array}{l} f_i \hookrightarrow \texttt{<}e'_1, \ldots, e'_k\texttt{>} \\ \quad : \mathbf{rec}\langle f_i, \texttt{<}y'_1, \ldots, y'_k\texttt{>}, \\ \quad \texttt{<}\tau'_1, \ldots, \tau'_k\texttt{>}, [(f_j, x_j, e_j)]_{j=1}^{n}\rangle\end{array}\right]_{i=1}^{n} \vdash e_0 \hookrightarrow e'_0 : \tau'_0\end{gathered}}{\Gamma \vdash \mathbf{letrec}\ [f_i = \lambda x_i \to e_i]_{i=1}^{n}\ \mathbf{in}\ e_0 \hookrightarrow e'_0 : \tau'_0}$$

$$\textit{where}\ [\Gamma_i = [y_{i,j} \hookrightarrow e_{i,j} : \tau_{i,j}]_{j=1}^{m_i}]_{i=1}^{n}$$

$$[y'_i \hookrightarrow e'_i : \tau'_i]_{i=1}^{k} = [y \hookrightarrow e : \tau \mid\ y \hookrightarrow e : \tau \in\ \textstyle\bigcup_{i=1}^{n}\Gamma_i,\ y \notin \bigcup_{i=1}^{n}\{f_i\}]$$

The declared functions are specialised by the *(SLAM)* rule first. There are n declarations in the **letrec**. Each function, f_i, has m_i free variables. Then, we have to build an environment of the results. We use the new **rec** type for typing the functions declared. In the first equation, Γ_i is defined to be the environment that holds bindings for f_i. There are n such environments, one for each declared function. In the second equation these environments are merged by the union operator (which excludes duplicates) to form an environment that holds bindings for the free variables of all right-hand sides. From these bindings we form a new environment by excluding all variables declared in this **letrec**, i.e. all f_is. We have now accomplished the environment described above.

As we have introduced a new type for functions we are in need of a new rule for application which handles this type. The rule is similar to the one for static function application *(SAPP)*. The difference is the environment in which the function is specialised. To build it we produce bindings for the all function representations first. Then re-assemble the three tuples and finally bind the λ-bound variable of the function we are specialising to the results of specialising the argument.

$$\textit{(RSAPP)}\quad \frac{\begin{array}{c}\Gamma \vdash f_k \hookrightarrow f'_k : \mathbf{rec}\langle f_k, {<}y_1,\ldots,y_n{>}, {<}\tau_1,\ldots,\tau_n{>}, [(f_i,x_i,e_i)]_{i=1}^{n}\rangle \\ \Gamma \vdash e \hookrightarrow e' : \sigma' \\ \left\{\begin{array}{l} \left[\begin{array}{l} f_i \hookrightarrow f'_k : \mathbf{rec}\langle f_i, {<}y_1,\ldots,y_n{>}, \\ \qquad {<}\tau_1,\ldots,\tau_n{>}, [(f_j,x_j,e_j)]_{j=1}^{n}\rangle \end{array}\right]_{i=1}^{n}, \\ {[y_i \hookrightarrow \pi_i\ f'_k : \tau'_i]_{i=1}^{n}}, \\ x_k \hookrightarrow e' : \sigma' \end{array}\right\} \vdash e_k \hookrightarrow e'_k : \tau' \end{array}}{\Gamma \vdash f_k\ e \hookrightarrow e'_k : \tau'}$$

2.4.7 Polyvariant Functions

Dynamic function specialisation has so far been accomplished by letting the λ-bound variable of the function take on the same type as the argument. Look at the example

$$\begin{array}{lll} \underline{\mathbf{let}} & a & = 1 \\ & b & = 2 \\ & f & = \underline{\lambda} x \rightarrow \mathbf{lift}\ x \\ \underline{\mathbf{in}} & \multicolumn{2}{l}{(f\ \underline{@}\ a, f\ \underline{@}\ b)} \end{array}$$

What type should f have? According to the first application $\mathbf{1} \underline{\rightarrow} \mathbf{int}$ and to the second $\mathbf{2} \underline{\rightarrow} \mathbf{int}$. This types are incompatible. **1** and **2** are distinct residual types and an expression has exactly one type. We are unable to specialise this example. Our specialiser is monovariant.

What is needed is the possibility to let a function have many specialised types. This is accomplished by doing one specialisation of the function for every type we want it to have. Resulting in several residual definitions, they are tupled to become one expression. The call for this behaviour is made through the use of the **poly** operator. Such a **poly**-function no longer has a function type but rather a tuple of functions. To make use of it, projections are applied. When applying the **poly**-function we must let the specialiser know it is not an ordinary function. This is done by the use of the operator **spec**. The rules are

$$\textit{(POLY)}\quad \frac{[\Gamma \vdash e \hookrightarrow e_i : \tau_i]_{i=1}^{n}}{\Gamma \vdash \mathbf{poly}\ e \hookrightarrow {<}e_1,\ldots,e_n{>} : {<}\tau_1,\ldots,\tau_n{>}}$$

$$\textit{(SPEC)}\quad \frac{\Gamma \vdash e \hookrightarrow e' : {<}\tau'_1,\ldots,\tau'_n{>}}{\Gamma \vdash \mathbf{spec}\ e \hookrightarrow \pi_k\ e' : \tau'_k}$$

All specialisations of a **poly**-function are put into a tuple and again we use static tuples from which we can easily select elements by the use of projections. We will later see how projection unfolding transforms the tuples into separate declarations for every definition.

Revisiting our example, we can now make it work by inserting the operators **poly** and **spec** in appropriate places

$$\begin{array}{llll}
\vdash & \underline{\mathbf{let}} & a & = 1\\
 & & b & = 2\\
 & & f & = \mathbf{poly}\ \underline{\lambda} x \to \mathbf{lift}\ x\\
 & \underline{\mathbf{in}} & (\mathbf{spec}\ f \mathbin{\underline{\text{@}}}\, a, \mathbf{spec}\ f \mathbin{\underline{\text{@}}}\, b) & \\
\hookrightarrow & \mathbf{let} & a & = \bullet\\
 & & b & = \bullet\\
 & & f & = <\lambda x \to 1, \lambda x \to 2>\\
 & \mathbf{in} & (\pi_1\ f\ a, \pi_2\ f\ b) & \\
: & (\mathbf{int}, \mathbf{int}) & &
\end{array}$$

2.4.8 Dynamic Constructors and case-Expressions

The specialisation of user defined data types should be consistent with how base types are treated. If we write the integer type

$$\mathbf{int} = \ldots \mid \mathbf{0} \mid \mathbf{1} \mid \ldots$$

it is easier to see the symmetry. For example the *Maybe a* data type. It is defined by

$$\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Yes}\ a \mid \mathit{No}$$

When specialising a dynamic integer, it is given as residual code the integer itself, while the type is **int**. A dynamic $\underline{\mathit{No}}$ should consequentially be specialised to the constructor itself with type *Maybe* α. We cannot instantiate the type variable a, so therefore the α. On the other hand, if for example $\underline{\mathit{Yes}}\ 1$ is specialised, a becomes instantiated. Here, α should be unified with the residual type of the argument to *Yes*. In this example the argument is 1 which is specialised to $\bullet : \mathbf{1}$ and the residual type thus becomes *Maybe* **1**. As residual code we keep the constructor *Yes* and apply it to the residual code of its argument. The whole specialisation is written

$$\vdash \underline{\mathit{Yes}}\ 1 \hookrightarrow \mathit{Yes}\ \bullet : \mathit{Maybe}\ \mathbf{1}$$

Thus the specialiser must know which type a constructor belongs to.

Our specialiser is not fully developed. It does not produce type definitions in its output but it does read the definitions in the source so that it can recognise the type of dynamic constructors. The syntax of the data type definitions do not allow annotations so it is not possible to express static types in those definitions. The escape is the possibility of using variables.

The types of the constructors are kept in the environment, Γ. A constructor is given the type of a function from the type of its arguments to its data type constructor. If we want these constructors and types to be polymorphic, which we do, we must use type schemes to represent the type of the variables in the type definition. The *Maybe* type is represented by the following environment.

$$[\mathit{Yes} \hookrightarrow \mathit{Yes} : \forall\sigma.\sigma \to \mathit{Maybe}\ \sigma,\ \ \mathit{No} \hookrightarrow \mathit{No} : \forall\sigma.\mathit{Maybe}\ \sigma]$$

This is how the type schemes are constructed. If $C\ t_1 \cdots t_m$ is a constructor of type $D\ x_1 \cdots x_n$ bind every x_i to a fresh type variable σ_i. The t_js are either one of the x_is or another type constructor (possibly having arguments). Let C have the type

$\tau_1 \to \cdots \to \tau_m \to D\ \sigma_1 \ldots \sigma_n$ where τ_j is the type of t_j. If $t_j = x_i$ then $\tau_j = \sigma_i$. If t_j for example is *Maybe* x_i then τ_j gets the type *Maybe* σ_i. The type is generalised to a type scheme $\forall \sigma_1 \ldots \sigma_n . \tau_1 \to \cdots \to \tau_m \to D\ \sigma_1 \ldots \sigma_n$ to allow polymorphic usage.

We are now ready to state the rule

$$\textit{(DCON)} \qquad \frac{[\Gamma, (C \ldots) \vdash e_i \hookrightarrow e'_i : \tau'_i]^m_{i=1}}{\begin{array}{c}\Gamma, (C \hookrightarrow C : \forall \sigma_1 \cdots \sigma_n . \tau_1 \to \cdots \to \tau_m \to \underline{D}\ \sigma_1 \ldots \sigma_n) \vdash \\ \underline{C}\ e_1 \ldots e_m \hookrightarrow C\ e'_1 \ldots e'_m : D\ \tau'_{k_1} \ldots \tau'_{k_n}\end{array}}$$

where the type of C is instantiated and τ'_i is the instance of τ_i

The components of a constructor application are retrieved by the usage of pattern matching in **case**-expressions. All decisions in a specialisation must be made upon static information, i.e. residual types. The static content of a dynamic constructor application does not include which constructor is applied, only the type to which it belongs. This implies that a **case**-expression examining such a constructor application must also be dynamic, i.e. it must be residualised with all its branches, as the information is not sufficient for selecting one of them.

We must now deal with a new syntactical category; patterns. As the expression which is cased upon is specialised, the patterns to match it against must also be specialised. The specialisation rules for patterns works much the same way as the corresponding rules for expressions. We have not yet discussed static constructors and therefore postpone the presentation of the specialisation rules for patterns until we have done so. We will only state the notation for it

$$\Gamma \vdash p \rightsquigarrow p' : \tau\ , \Gamma'$$

This means that the pattern p is specialised to the pattern p' with residual type τ in the environment Γ. The specialisation does not need an environment to take place in. The variables of a pattern *become* bound by the pattern matching and thus earlier bindings are irrelevant. The reason why Γ is supplied is that it holds the information on which type a constructor belongs to.

Although the outer constructors of the patterns in a dynamic **case** are dynamic, they may have static or partially static arguments which affect the specialisation of the expressions in the branches. Therefore, the environment in which these expressions are specialised must also hold bindings for the variables which are bound by the pattern matching. When we consider dynamic **case**-expressions, these bindings are produced by the specialisation rules for patterns. This environment, represented by Γ' in the relation above, binds all variables of the pattern to fresh variable names and the appropriate residual type. We are in need of fresh variable names here as there is a risk of name capture.

The rule for specialising dynamic **case**-expressions is

$$\textit{(DCASE)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 \hookrightarrow e'_0 : \tau'_0 \\ [\Gamma \vdash p_i \rightsquigarrow p'_i : \tau'_0\ , \Gamma'_i \qquad \Gamma'_i \cup \Gamma \vdash e_i \hookrightarrow e'_i : \tau']^n_{i=1}\end{array}}{\begin{array}{l}\Gamma \vdash\ \ \underline{\mathbf{case}}\ e_0\ \underline{\mathbf{of}}\ [p_i \to e_i]^n_{i=1} \hookrightarrow \\ \qquad\qquad \mathbf{case}\ e'_0\ \mathbf{of}\ [p'_i \to e'_i]^n_{i=1} : \tau'\end{array}}$$

To begin with, the expression which is cased upon, e_0, is specialised to have the type τ_0'. This should also be the type of all specialised patterns. The application of the specialisation rules to the pattern p_i constructs an environment which is used in conjunction with the environment for the entire **case**-expression when the expression in the same branch, e_i, is specialised. The specialised versions of all expressions and patterns are used to build the resulting **case**-expression.

Look at an enlargement of the *Maybe* -example.

$$\begin{array}{lllllllll}
\vdash & \underline{\mathbf{let}} & a = \underline{\mathit{Yes}}\ 1 & & \hookrightarrow & \mathbf{let} & a = \mathit{Yes}\ \bullet & & : \mathbf{int} \\
& \underline{\mathbf{in}} & \underline{\mathbf{case}}\ a\ \underline{\mathbf{of}} & & & \mathbf{in} & \mathbf{case}\ a\ \mathbf{of} & & \\
& & \quad \underline{\mathit{Yes}}\ x & \to \mathbf{lift}\ x & & & \quad \mathit{Yes}\ x' & \to 1 & \\
& & \quad \underline{\mathit{No}} & \to \mathbf{lift}\ 2 & & & \quad \mathit{No} & \to 2 &
\end{array}$$

As the constructor *Yes* in a is annotated as dynamic the **case**-expression taking a apart must also be dynamic as well as the constructors in the patterns. Specialisation of the expression in the first branch is taken place in an environment which holds the binding $(x \hookrightarrow x' : 1)$. The residual types of a and the two patterns are *Maybe* $\mathbf{1}$, *Maybe* α and *Maybe* β respectively. These types are unifiable with $\alpha = \beta = \mathbf{1}$. If they were not, we would have an error and would not be able to continue.

There are restrictions of how the patterns can be built. A discussion on this is held in section 2.4.13 after all rules for **case**-expressions and constructors have been presented.

In the above example we could easily have substituted the static integer with its dynamic correspondence but when we try a more difficult example we encounter problems. The problem being our type constructors are monovariant. Consider the type for lists

$$\mathbf{data}\ \mathit{List}\ a = \mathit{Cons}\ a\ (\mathit{List}\ a) \mid \mathit{Nil}$$

If we want to declare a dynamic list of static integers, the type variable a must be instantiated to the type of the elements of the list. The type of a static integer is the integer itself, so we can only restrict the list to hold one specific integer, which will instantiate a.

For example the list $\underline{\mathit{Cons}}\ 1\ (\underline{\mathit{Cons}}\ 2\ \underline{\mathit{Nil}})$. The outer *Cons* instantiates the type variable a to $\mathbf{1}$ while the inner to $\mathbf{2}$, which of course results in an unification error. We will get back to this problem at a later stage when we have introduced constructor specialisation below. Note that there is no problem to declare, for example, the dynamic list of static ones.

2.4.9 Static Constructors and case-Expressions

Continuing the comparison between integers and constructors, what about static constructors?

Recall the way static integers where treated. A new type was constructed having the integer itself as its name and $\bullet$ as its only element. This can be written as

$$\mathbf{data\ n} = \bullet$$

Consequently, for user defined static constructors, construct a new type with only one constructor. The name of the type should come from the original constructor.

Call the only constructor of the new type Void_n, where n indicates its arity. This is the difference from integers; there might be arguments to a constructor. Therefore, define the new type as

$$\textbf{data}\ C\ a_1 \ldots a_n = \mathit{Void}_n\ a_1 \ldots a_n$$

The constructor is turned into a type constructor with the same arity, n. The type variables are unified with the residual types of the arguments to the original constructor to derive the type of the residual code. The rule becomes

$$\textit{(SCON')}\quad \frac{[\Gamma \vdash e_i \hookrightarrow e'_i : \tau'_i]_{i=1}^{n}}{\Gamma \vdash C\ e_1 \ldots e_n \hookrightarrow Void_n\ e'_1 \ldots e'_n : C\ \tau'_1 \ldots \tau'_n}$$

So far so good. The solution seems quite natural and intuitive but when trying to specialise static **case**-expressions we encounter problems. A static **case**-expression should disappear from the code, leaving just the expression of the succeeding branch. How are we to specialise the following example?

$$\begin{array}{ll}
\underline{\textbf{let}} & a = \mathit{Cons}\ (\textbf{lift}\ 1)\ \mathit{Nil} \\
\underline{\textbf{in}} & \quad \textbf{case}\ a\ \textbf{of} \\
 & \qquad \begin{array}{ll} \mathit{Cons}\ x\ \mathit{xs} & \rightarrow \textbf{lift}\ x \\ \mathit{Nil} & \rightarrow \textbf{lift}\ 0 \end{array}
\end{array}$$

Specialising a will give $\mathit{Void}_2\ 1\ \mathit{Void}_0 : \mathit{Cons}\ \textbf{int}\ \mathit{Nil}$ and to specialise the **case** we would still have to case upon a to get x, forcing us to keep the **case** and the branch we know will succeed. Leaving

$$\begin{array}{ll}
\textbf{let} & a = \mathit{Void}_2\ 1\ \mathit{Void}_0 \\
\textbf{in} & \quad \textbf{case}\ a\ \textbf{of} \\
 & \qquad Void_2\ x\ \mathit{xs} \rightarrow x
\end{array}$$

This cannot be the intention of a static **case**. Imaging substituting the word *Tuple* for *Void*. The specialisation of a becomes $\vdash \mathit{Cons}\ (\textbf{lift}\ 1)\ \mathit{Nil} \hookrightarrow \mathit{Tuple}_2\ 1\ \mathit{Tuple}_0 : \mathit{Cons}\ \textbf{int}\ \mathit{Nil}$. To go all the way: $\vdash \mathit{Cons}\ (\textbf{lift}\ 1)\ \mathit{Nil} \hookrightarrow <1, <> > : \mathit{Cons}\ \textbf{int}\ \mathit{Nil}$. If there is a constructor carrying no information, with the only purpose to bundle some expressions together we might just as well use a tuple instead. Therefore, define the type alias

$$\textbf{type}\ C\ a_1 \ldots a_n = <a_1, \ldots, a_n>$$

instead of the new data type. Using a tuple, we know how to get at the elements without casing, with projections. The following specialisation is now possible

$$\begin{array}{ll}
\textbf{let} & a = <1, <> > \\
\textbf{in} & \pi_1\ a
\end{array}$$

Note that static tuples are used so that further improvements can be performed by the projection unfolding. The binding-time of the elements of the list does not matter, the **lift**-operator could have been moved from the construction of a to the branch in the **case**-expression. The reason for this is that every time a static constructor is encountered, a new type alias is constructed and there is thus no problem with monovariance.

The new rule for static constructors and the rule for static **case** are

$$\textit{(SCON)} \quad \frac{[\Gamma \vdash e_i \hookrightarrow e'_i : \tau'_i]_{i=1}^{n}}{\Gamma \vdash C\ e_1 \ldots e_n \hookrightarrow {<}e'_1, \ldots, e'_n{>} : C\ \tau'_1 \ldots \tau'_n}$$

$$\textit{(SCASE)} \quad \frac{\begin{array}{c}\Gamma \vdash e_0 \hookrightarrow e'_0 : \tau'_k \\ (\mathit{mk}\Gamma_S\ p_k\ e'_0\ \tau'_k)\ \cup\ \Gamma \vdash e_k \hookrightarrow e'_k : \tau'\end{array}}{\Gamma \vdash \textbf{case}\ e_0\ \textbf{of}\ [p_i \to e_i]_{i=1}^{n} \hookrightarrow e'_k : \tau'}$$

To be able to specialise the expression of the succeeding branch, the variables of its corresponing pattern must be bound. This is done by the application of $\mathit{mk}\Gamma_S$. $\mathit{mk}\Gamma_S$ will be defined below.

2.4.10 Rules for Specialising Patterns

Now that we have rules for the correspondences of all patterns, we present rules for specialising patterns. What they state is very similar to their correspondences so little extra explanation should be necessary. The exception is the rule for variables that produce bindings for the new environment rather than looks up bindings in the old environment. The rules for tuples and constructors simply merges and propagates the newly constructed environments.

Literals of base types are specialised to • with their values as types. They produce no bindings.

$$\textit{(PSINT)} \quad \Gamma \vdash n \rightsquigarrow \bullet : \mathbf{n}\, ,\ []$$

$$\textit{(PSSTR)} \quad \Gamma \vdash s \rightsquigarrow \bullet : \mathbf{s}\, ,\ []$$

Variables are renamed to avoid name capture.

$$\textit{(PVAR)} \quad \Gamma \vdash x \rightsquigarrow x' : \tau\, ,\ [x \hookrightarrow x' : \tau] \quad x'\ \textit{fresh}$$

To specialise a tuple, specialise its components and rebuild the tuple.

$$\textit{(PDTUP)} \quad \frac{[\Gamma \vdash e_i \rightsquigarrow e'_i : \tau'_i\, ,\ \Gamma_i]_{i=1}^{n}}{\Gamma \vdash (e_1, \ldots, e_n) \rightsquigarrow (e'_1, \ldots, e'_n) : (\tau'_1, \ldots, \tau'_n)\, ,\ \bigcup_{i=1}^{n}\ \Gamma_i}$$

$$\textit{(PSTUP)} \quad \frac{[\Gamma \vdash e_i \rightsquigarrow e'_i : \tau'_i\, ,\ \Gamma_i]_{i=1}^{n}}{\Gamma \vdash {<}e_1, \ldots, e_n{>} \rightsquigarrow {<}e'_1, \ldots, e'_n{>} : {<}\tau'_1, \ldots, \tau'_n{>}\, ,\ \bigcup_{i=1}^{n}\ \Gamma_i}$$

Specialising constructors in patterns is similar to specialising tuples.

$$\textit{(PDCON)} \quad \frac{[\Gamma, (C \ldots) \vdash e_i \rightsquigarrow e'_i : \tau'_i\, ,\ \Gamma_i]_{i=1}^{m}}{\begin{array}{c}\Gamma, (C \hookrightarrow C : \forall \sigma_1 \cdots \sigma_n . \tau_1 \to \cdots \to \tau_m \to \underline{D}\ \sigma_1 \ldots \sigma_n) \vdash \\ \underline{C}\ e_1 \ldots e_m \rightsquigarrow C\ e'_1 \ldots e'_m : D\ \tau'_{k_1} \ldots \tau'_{k_n}\, ,\ \bigcup_{i=1}^{m}\ \Gamma_i\end{array}}$$

where the type of C is instantiated and τ'_i is the instance of τ_i

$$\textit{(PSCON)} \quad \frac{[\Gamma \vdash e_i \rightsquigarrow e'_i : \tau'_i\, ,\ \Gamma_i]_{i=1}^{n}}{\Gamma \vdash C\ e_1 \ldots e_n \rightsquigarrow {<}e'_1, \ldots, e'_n{>} : C\ \tau'_1 \ldots \tau'_n\, ,\ \bigcup_{i=1}^{n}\ \Gamma_i}$$

2.4.11 Defining $mk\Gamma_S$

The purpose of the $mk\Gamma_S$-function is to produce an environment which binds the variables of a pattern so that its corresponding expression can be specialised. The function takes three arguments; the *source* pattern of a branch and the *residual* code and type of the expression which is cased on. It is partial; it is not possible to derive bindings for the variables of a pattern if the pattern does not match the expression.

If the pattern is a variable is should be bound to the residual expression and its type.

$$mk\Gamma_S\ x\ e\ \tau \;=\; [x \hookrightarrow e : \tau]$$

Values of base types produce no bindings.

$$mk\Gamma_S\ n\ \bullet\ n \;=\; []$$

$$mk\Gamma_S\ s\ \bullet\ s \;=\; []$$

If the pattern to produce bindings for is a static tuple, the expression might also be a static tuple of static tuple type. Merge the bindings produced by their components.

$$mk\Gamma_S\ {<}p_1,\ldots,p_n{>}\ {<}e_1,\ldots,e_n{>}\ {<}\tau_1,\ldots,\tau_n{>} \;=\; \bigcup_{i=1}^{n}\ (mk\Gamma_S\ p_i\ e_i\ \tau_i)$$

If the pattern is a static constructor and the residual expression is a specialised static constructor, i.e. a static tuple with a static constructor type, apply $mk\Gamma_S$ to the components and merge the results.

$$mk\Gamma_S\ (C\ p_1 \ldots p_n)\ {<}e_1,\ldots,e_n{>}\ (C\ \tau_1 \ldots \tau_n) \;=\; \bigcup_{i=1}^{n}\ (mk\Gamma_S\ p_i\ e_i\ \tau_i)$$

It is also possible that the pattern is a static tuple or static constructor and the type of the expression is static tuple but the expression itself is not a static tuple. If so, we apply projections to the expression to represent the components.

$$mk\Gamma_S\ {<}p_1,\ldots,p_n{>}\ e\ {<}\tau_1,\ldots,\tau_n{>} \;=\; \bigcup_{i=1}^{n}\ (mk\Gamma_S\ p_i\ (\pi_i\ e)\ \tau_i)$$

$$mk\Gamma_S\ (C\ p_1 \ldots p_n)\ e\ (C\ \tau_1 \ldots \tau_n) \;=\; \bigcup_{i=1}^{n}\ (mk\Gamma_S\ p_i\ (\pi_i\ e)\ \tau_i)$$

Dynamic tuples and constructors cannot occur in the patterns of a static **case**.

2.4.12 Constructor Specialisation

Mogensen argues in [Mog96] that limits present in the source code should not be inherited to the residual code. One such limit is the number of constructors of a data type. We do not allow specialisation of constructors [Mog93] of user defined data types in general. There is only one specialisable constructor, *In*. We shall see that by using this constructor we can achieve essentially the same result as general constructor specialisation.

Let us continue with the list example. If we want to specialise the dynamic list of static integers, $\underline{\mathit{Cons}}\ 1\ (\underline{\mathit{Cons}}\ 2\ \underline{\mathit{Nil}})$, we would be in need of the type

$$\textbf{data}\ \mathit{List} = \mathit{Cons}_1\ \mathbf{1}\ \mathit{List} \mid \mathit{Cons}_2\ \mathbf{2}\ \mathit{List} \mid \mathit{Nil}$$

where the *Cons* constructor is specialised to residual types **1** and **2**. Write the dynamic list as follows

$$\underline{In}\ (Cons\ 1\ (\underline{In}\ (Cons\ 2\ (\underline{In}\ Nil))))$$

The binding-time of the list constructors are changed to static while we apply a dynamic $\underline{In}$ to each of them. The latter will then be specialised to their arguments forming the new data type

$$\mathbf{data}\ In_{List} = In_1\ \mathbf{1}\ List \mid In_2\ \mathbf{2}\ List \mid In_3$$

which is essentially the same as the type we seek. Actually, the constructed type is never named, it is only shown here for the sake of clarification.

The specialisation rule is

$$\textit{(IN)}\qquad \frac{\Gamma \vdash e : \tau \hookrightarrow e' : \tau'_k}{\Gamma \vdash \underline{In}\ e \hookrightarrow In_k\ e' : \Sigma_{i=1}^{n} In_i\ \tau'_i}$$

To be able to inspect the arguments of the newly constructed constructors we need a **case**-expression which is also possible to specialise. There should be only one branch in the source expression corresponding to the only constructor *In*, while in the residual expression there should be one branch for every specialisation of *In*. The specialisation of the expression of every branch should utilise the information carried in the specialised constructor. The rule is

$$\textit{(INCASE)}\qquad \frac{\begin{array}{c}\Gamma \vdash e_0 \hookrightarrow e'_0 : \Sigma_{i=1}^{n} In_i\ \sigma'_i \\ [\Gamma, x \hookrightarrow x' : \sigma'_i \vdash e_1 \hookrightarrow e'_{1,i} : \tau']_{i=1}^{n}\end{array}}{\begin{array}{ll}\Gamma \vdash & \underline{\mathbf{case}}\ e_0\ \underline{\mathbf{of}}\ \underline{In}\ x \rightarrow e_1\ \hookrightarrow \\ & \quad \mathbf{case}\ e'_0\ \mathbf{of}\ [In_i\ x' \rightarrow e'_{1,i}]_{i=1}^{n} : \tau'\end{array}}\qquad x'\ \text{fresh}$$

Note that the rules for specialising patterns are not used here. This is due to the syntax of the pattern; it should be the constructor *In* applied to a variable and nothing else.

Specialisation of data types are desirable in its own right to remove inherited limits. It can lead to stronger over-all specialisation in certain cases. Information is moved from the residual code to its type. In the list example, the list constructors are made static and are thus moved to the type. There are no restrictions against dynamic constructors having static arguments, the problem is the monovariance of the type constructors. Constructor specialisation can for this reason be viewed as a way of overcoming this problem. The technique can be used in another way and still yield a similar result. Continue the list example; Instead of specialising the list constructors, inject the static integers, which causes the problem, into the same domain by applying $\underline{In}$ to them.

$$\underline{Cons}\ (\underline{In}\ 1)\ (\underline{Cons}\ (\underline{In}\ 2)\ \underline{Nil})$$

Thus creating the type

$$\mathbf{data}\ In_{Int} = In_1\ \mathbf{1} \mid In_2\ \mathbf{2}$$

yielding the specialised list

$$Cons\ (In_1\ \bullet)\ (Cons\ (In_2\ \bullet)\ Nil)$$

where the list constructors remain unchanged.[5] It remains to be investigated what the differences of these two approaches lead to. Here is an example that indicates that the differences are small. Define a function that adds the element of a list and specialises it to the list we have been dealing with. Using the former approach

```
$\vdash$  $\underline{\textbf{letrec}}$  $add = \underline{\lambda} x \rightarrow$  $\underline{\textbf{case}}\ x\ \underline{\textbf{of}}$
                                  $In\ y \rightarrow$  $\textbf{case}\ y\ \textbf{of}$
                                               $Cons\ z\ zs$  $\rightarrow \textbf{lift}\ z \underline{+} add\ zs$
                                               $Nil$          $\rightarrow \textbf{lift}\ 0$
     $\underline{\textbf{in}}$        $\underline{\textbf{let}}$  $a = \underline{In}\ (Cons\ 1\ (\underline{In}\ (Cons\ 2\ (\underline{In}\ Nil))))$
                 $\underline{\textbf{in}}$   $add\ \underline{@}\, a$
$\hookrightarrow$  $\textbf{let}$  $add = \lambda x \rightarrow$  $\textbf{case}\ x\ \textbf{of}$
                                $In_1\ y$  $\rightarrow 1\ +\ add\ (\pi_2\ y)$
                                $In_2\ y$  $\rightarrow 2\ +\ add\ (\pi_2\ y)$
                                $In_0\ y$  $\rightarrow 0$
     $\textbf{in}$     $\textbf{let}$  $a = In_1 <\bullet, In_2 <\bullet, In_0\ <> >>$
               $\textbf{in}$   $add\ a$
   $:$  $\textbf{int}$
```

While using the latter gives

```
$\vdash$  $\underline{\textbf{letrec}}$  $add = \underline{\lambda} x \rightarrow$  $\underline{\textbf{case}}\ x\ \underline{\textbf{of}}$
                                   $\underline{Cons}\ y\ ys$  $\rightarrow$  $\underline{\textbf{case}}\ y\ \underline{\textbf{of}}$
                                                          $\underline{In}\ z \rightarrow \textbf{lift}\ z \underline{+} add\ ys$
                                   $\underline{Nil}$          $\rightarrow \textbf{lift}\ 0$
     $\underline{\textbf{in}}$        $\underline{\textbf{let}}$  $a = \underline{Cons}\ (\underline{In}\ 1)\ (\underline{Cons}\ (\underline{In}\ 2)\ \underline{Nil})$
                 $\underline{\textbf{in}}$   $add\ \underline{@}\, a$
$\hookrightarrow$  $\textbf{let}$  $add = \lambda x \rightarrow$  $\textbf{case}\ x\ \textbf{of}$
                                $Cons\ y\ ys$  $\rightarrow$  $\textbf{case}\ y\ \textbf{of}$
                                                      $In_1\ z$  $\rightarrow 1\ +\ add\ ys$
                                                      $In_2\ z$  $\rightarrow 2\ +\ add\ ys$
                                $Nil$          $\rightarrow 0$
     $\textbf{in}$     $\textbf{let}$  $a = Cons\ (In_1\ \bullet)\ (Cons\ (In_2\ \bullet)\ Nil)$
               $\textbf{in}$   $add\ a$
   $:$  $\textbf{int}$
```

as we can see, the residual codes are very similar.

2.4.13 Mixed Binding-Times in case-Patterns

When writing an unannotated definition of the *add* function above, we would normally use a single **case**-expression. It might seem annoying that we must add yet another **case** when adding the *In*s. Why couldn't we simply add the *In*s to the existing patterns and make them nested?

Certain nested patterns are allowed but there are restrictions; When specialising a static case expression, it must be possible to conclude which branch that matches.

[5] The • terms will be removed by the void erasure mechanism.

This selection cannot rely on dynamic information, therefore patterns of static **case**-expressions cannot contain dynamic constructors. Nested static constructors are allowed. We can still case on static constructors holding dynamic dittos by using nested **case**s, matching the dynamic parts against variables and perform dynamic **case**s on them.

When specialising dynamic **case**s, it is possible to have static constructors in the patterns but it is not likely to be of any practical use. The residual patterns of a **case**-expression must all be of the same residual type but static constructors of the same source type are specialised to different residual types. Let therefore the static constructors match variables and perform static **case**s on them. Nested dynamic constructors causes no problems, with one exception; the constructor *In* must not occur in them.

It would be nice to allow all kinds of constructors to be mixed in patterns. This could be accomplished by letting some preprocessor compile **case**-expressions with patterns of mixed kinds into nested **case**-expressions which have the above properties. This is left for future work.

2.4.14 if-Expressions

if-expressions can be considered a special case of **case**-expressions. Rather then compiling the **if**s into **case**s, we supply rules for the **if**s. With the comments on the *SCASE* and *DCASE* in mind, there is not much to add. When specialising the conditional of the static variant, its residual type must be either *True* or *False*, so that we can conclude which branch to specialise further.

$$\textit{(DIF)}\quad \frac{\Gamma \models e_0 \hookrightarrow e_0' : \textbf{bool} \quad \Gamma \models e_1 \hookrightarrow e_1' : \tau \quad \Gamma \models e_2 \hookrightarrow e_2' : \tau}{\Gamma \models \underline{\textbf{if}}\ e_0\ \underline{\textbf{then}}\ e_1\ \underline{\textbf{else}}\ e_2 \hookrightarrow \textbf{if}\ e_0'\ \textbf{then}\ e_1'\ \textbf{else}\ e_2' : \tau}$$

$$\textit{(SIF}_{\textit{True}}\textit{)}\quad \frac{\Gamma \models e_0 \hookrightarrow e_0' : True \quad \Gamma \models e_1 \hookrightarrow e_1' : \tau}{\Gamma \models \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \hookrightarrow e_1' : \tau}$$

$$\textit{(SIF}_{\textit{False}}\textit{)}\quad \frac{\Gamma \models e_0 \hookrightarrow e_0' : False \quad \Gamma \models e_2 \hookrightarrow e_2' : \tau}{\Gamma \models \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \hookrightarrow e_2' : \tau}$$

2.5 Void Erasure

In the residual code void values are numerous. They were originally placed there as some kind of placeholder for entirely static expressions. They have the mere function of maintaining type correctness but hold no information that affects the behaviour of the residual program. For example

$$\vdash (\lambda x \to \textbf{lift}\ x)\ 1 \hookrightarrow (\lambda x \to 1)\ \bullet : \textbf{int}$$

where $\bullet$ replaces the 1 so that there is an argument for the λ abstraction. Both x and $\bullet$ have the same type, namely 1 which is known to be a trivial type. A type containing only one element. It is known that the whole information of this values lies in their type and has therefore already been exploited. The void and λ abstraction can be removed resulting in the expression 1. We call this phase void erasure and write it as follows.

$$
\begin{array}{l}
|e^\nu| \longrightarrow \bullet^\nu \\
|x^{\nu\to\tau}| \longrightarrow |x^\tau| \\
|\lambda x^\nu \to e| \longrightarrow |e| \\
|\lambda x \to e| \longrightarrow \lambda x \to |e| \\
|e_1^{\nu\to\tau} e_2^\nu| \longrightarrow |e_1^{\nu\to\tau}| \\
|e_1 e_2| \longrightarrow |e_1||e_2| \\
|n| \longrightarrow n \\
|s| \longrightarrow s \\
|\bullet^\nu| \longrightarrow \bullet^\nu \\
\left|\begin{array}{lll}
\textbf{let} & x_1 & = e_1 \\
 & & \vdots \\
 & {x_i}^\nu & = {e_i}^\nu \\
 & & \vdots \\
 & x_n & = e_n \\
\textbf{in} & e_0 &
\end{array}\right|
\longrightarrow
\left|\begin{array}{lll}
\textbf{let} & x_1 & = e_1 \\
 & & \vdots \\
 & x_{i-1} & = e_{i-1} \\
 & x_{i+1} & = e_{i+1} \\
 & & \vdots \\
 & x_n & = e_n \\
\textbf{in} & e_0 &
\end{array}\right| \\
|\textbf{case}\ e\ \textbf{of}\ [p_i \to e_i]_{i=1}^n| \longrightarrow \textbf{case}\ |e|\ \textbf{of}\ [|p_i| \to |e_i|]_{i=1}^n \\
|\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2| \longrightarrow \textbf{if}\ |e_0|\ \textbf{then}\ |e_1|\ \textbf{else}\ |e_2| \\
|(e_1, \ldots, e_i^\nu, \ldots, e_n)| \longrightarrow |(e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n)| \\
|(e_1, \ldots, e_n)| \longrightarrow (|e_1|, \ldots, |e_n|) \\
|< e_1, \ldots, e_n >| \longrightarrow< |e_1|, \ldots, |e_n| > \\
|C\ e_1 \cdots e_i^\nu \cdots e_n| \longrightarrow |C\ e_1 \cdots e_{i-1}\ e_{i+1} \cdots e_n| \\
|C\ e_1 \cdots e_n| \longrightarrow C\ \ |e_1| \cdots |e_n| \\
|e_1 \oplus e_2| \longrightarrow |e_1| \oplus |e_2|
\end{array}
$$

Figure 9: Transformation rules for the void erasure mechanism.

$$(\lambda x \to 1)\ \bullet \longrightarrow 1$$

It is not only the value $\bullet$ which has a trivial type but also the constant function returning $\bullet$. Define a predicate *triv* that can tell if a type is trivial.

$$
\begin{array}{l}
\textit{triv}\ \mathbf{n} \quad \textit{where } n \textit{ is an integer and denotes the type } \mathbf{n} \\
\textit{triv}\ \mathbf{s} \quad \textit{where } s \textit{ is a string and denotes the type } \mathbf{s} \\
\textit{triv}\ \sigma \to \tau = \textit{triv}\ \tau \\
\textit{triv}\ (\tau_1, \ldots, \tau_n) = \textit{triv}\ \tau_1 \wedge \cdots \wedge \textit{triv}\ \tau_n \\
\textit{triv}\ <\tau_1, \ldots, \tau_n> = \textit{triv}\ \tau_1 \wedge \cdots \wedge \textit{triv}\ \tau_n
\end{array}
$$

With this definition, let ν stand for such a trivial type, while τ can be any type. In the following transformation rules, where $|e| \longrightarrow e'$ denotes that when e is void erased the result is e', the types are given as superscripts. The rules are shown in figure 9. Note that the same transformation rules are used for both expressions and patterns.

When applying a function, if the argument is of trivial type simply remove it. The λ abstracted variable of the function should also be removed, wherever the function is defined. If it is defined outside the application and referred to by a variable, only the type of the variable is here changed. For example

$$|\textbf{let}\ f = \lambda x \rightarrow 1\ \textbf{in}\ f\bullet| \longrightarrow \textbf{let}\ f = 1\ \textbf{in}\ f$$

where the type of changes from $1 \rightarrow \textbf{int}$ to **int**. Any expression of trivial type ν can be replaced by $\bullet^{\nu}$.

Integers and strings have their named types and are not affected.

A declaration in a **let** expression having a trivial type can be removed. The variable which is defined will not be present in the void erased top level expression. It will have been replaced by $\bullet^{\nu}$. If all declarations are removed the result is just the top level expression.

Tuple elements of trivial type can be removed from the tuple, lowering its arity. Tuples where all the elements are of trivial type, has a trivial type according to the definition of *triv*.

Arguments of trivial type, to constructors are removed, lowering the arity of that constructor.

The conditional and branches of an **if** expression are void erased in the same way as the arguments of a primitive operation.

2.6 Projection Unfolding

The intention of a program specialiser is that no static expressions should be present in the final residual code. In ours, this is valid for all constructs except tuples. Rather than removing static tuples, it *creates* such. For example, **poly**-expressions and static constructors are residualised as static tuples and **spec**-expressions are residualised as projections on them. The handling of static tuples does not fit well into the structure of inference rules that specifies the specialiser. This has to be accomplished in a separate phase.

We could easily get around the problem by residualising the static tuples by converting them into dynamic ones. The projections could be replaced by selector functions, which would be very easy to generate code for. Static constructors with only entirely static arguments are specialised to nested static tuples with only $\bullet$ as elements. These terms are removed by the void erasure.

This is, however, not satisfactory. Motivation can be found in [Mog96] and is discussed in section 2.7. Therefore we have developed a mechanism which we call *projection unfolding* which performs this removal.

The output of the specialiser contains static tuples in the code. These code tuples can have different types. A residualised static function has a closure type. A residualised static constructor has a constructor type. The target of this phase is tuples in the *code* and they should all be handled the same way independent of their type. From the specialisers point of view, the static function has a closure type. When the function is residualised as a static tuple, from the point of view of the reader of the output, it is just a tuple. Therefore the closure type can be converted to a static tuple type. The obvious conversion is

$$\begin{aligned}&<e_1, \ldots, e_n> : \textbf{clos}\langle <x_1, \ldots, x_n>, <\tau_1, \ldots, \tau_n>, x, e\rangle \\ &\quad \Rightarrow <e_1, \ldots, e_n> : <\tau_1, \ldots, \tau_n>\end{aligned}$$

It is after all τ_i which is the type of e_i. The same reasoning applies to the other types which are the types of static code tuples. Recursive static functions are converted by

$$\begin{aligned}&<e_1, \ldots, e_n> : \mathbf{rec}\langle f_k, <y_1, \ldots, y_n>, <\tau_1, \ldots, \tau_n>, [(f_i, x_i, e_i)]_{i=1}^{m} >\rangle \\ &\quad \Rightarrow <e_1, \ldots, e_n> : <\tau_1, \ldots, \tau_n>\end{aligned}$$

Static constructors are converted by

$$<e_1, \ldots, e_n> : C\ \tau_1 \ldots \tau_n \Rightarrow <e_1, \ldots, e_n> : <\tau_1, \ldots, \tau_n>$$

The residual type of specialised **poly**-functions already has the correct type and does not need to be converted.

Now that all static code tuples have static tuple types, let $\sigma = <\sigma_1, \ldots, \sigma_n>$ denote such a type for the rest of this section. σ_i can then denote any type. Furthermore, let τ denote any type that is not a static tuple. Write e^σ to denote that e has the type σ.

The description of the method is divided into two cases. The first is when there for every tuple is a set (possibly empty) of projections selecting elements from it. For example, when specialising **poly**-expression. The definition of the **poly**-function becoming a tuple and the **spec**s becoming projections. Assuming that the program is type correct implies that every projection is applied to an expression that reduces to either a tuple or a variable that refers to a tuple. To deal with the former, move the projection into the expression and apply it where the tuple is reached. This must be done in a controlled way and it is this that the method is mainly about. The latter is handled by renaming the resulting variable by giving it the same index as that of the projection. The tuple which the variable refers to is split up into its elements[Rom90]. They are named by taking the variables original name and adding to it the index corresponding to their places in the tuple.

The second case is when the components of a static tuple is retrieved by pattern matching in a **case**-expression. When a dynamic constructor which has a static constructor as an argument is specialised, the result is a constructor with a static tuple as one argument. To handle this the tuple is split up so that every element becomes an argument to the constructor. The arity of the constructor is then raised. The same thing is done with the patterns.

Let us look at an example. Suppose the following program is the output of the specialiser.

$$\begin{array}{lll}\mathbf{let} & f & = <\lambda x \rightarrow x, \lambda x \rightarrow x> \\ & a & = <<1, 2>, <3, 4>> \\ \mathbf{in} & \multicolumn{2}{l}{\pi_1((\pi_1 f)(\pi_2 a))}\end{array}$$

The result of running this program should be 3. The objection is to apply the projections to the tuples so that they are removed from the code. In the expression $\pi_1\ f$, the projection is applied to a variable that refers to a tuple. Replace the expression with f_1, which is the variable name with an added index. The same index as for the projection. This calls for a declaration of the new variable name, $f_1 = \lambda x \rightarrow x$. The right-hand side of this declaration is the tuple that f refers to with the projection applied to it. The projection is moved from the top level expression of the **let**-expression into one of its declarations. The variable name reflects this operation. Note that the types of f and f_1 are not the same. The expression $\pi_2\ a$ is treated in the same way resulting in $a_2 = <3, 4>$. There is only one projection left in the program, which now is

$$\begin{array}{lll} \textbf{let} & f_1 & = \lambda x \rightarrow x \\ & a_2 & = <3, 4> \\ \textbf{in} & \multicolumn{2}{l}{\pi_1(f_1\ a_2)} \end{array}$$

The remaining projection is applied to an application. The result of this application is, of course, the result of the function f_1, which is x. x will be bound to a_2 *after* the application has been performed.

To remove the last projection it looks as if it should be applied to the argument a_2. This is, however, not a method that works in general when projections are applied to applications. Instead, we will apply the projection to the function body. Before we can do that, we perform arity raising, splitting both the argument and the λ-abstracted variable. This transforms the function to $(\lambda x_1 \rightarrow \lambda x_2 \rightarrow <x_1, x_2>)$ and the argument into its two components $a_{2,1}$ and $a_{2,2}$. We have moved the tupling of the two components from the argument into the body of the function! It is now possible to apply the projection to it. The type of the result of the application does not change but the type of the function does. It should therefore get a new name reflecting the new result type, $f_{1,1}$.

The projection unfolding is now finished and results in the program

$$\begin{array}{lll} \textbf{let} & f_{1,1} & = \lambda x_1 \rightarrow \lambda x_2 \rightarrow x_1 \\ & a_{2,1} & = 3 \\ & a_{2,2} & = 4 \\ \textbf{in} & \multicolumn{2}{l}{f_{1,1}\ a_{2,1}\ a_{2,2}} \end{array}$$

Note that information, originally present in the program, has now disappeared. During projection unfolding it was discovered that first component of a, $<1, 2>$, is not used but it was *not* discovered that the 4 is not used. Our mechanism is capable of removing *some* dead code. It is, however, doubtful if this has any practical impact.

2.6.1 Specification of the Projection Unfolding Mechanism

Consider the projection last removed from the example. It was originally applied to an expression which was an application. It was removed by being pushed into the application and applied to the function. It is possible that several projections should be moved inwards at the same time. Let therefore Π be a *list of projections*, projections that have been removed from the code but have not yet been applied to a tuple or a variable. In the general case, it is such a list that should be moved inwards rather than a single projection. Furthermore, let $|e|^{\Pi}$ denote that a list of projections is applied to the expression e, i.e. the projections in Π should be pushed into e and eventually applied to variables or tuples. In other words; $|e|^{\Pi}$ is equivalent to $(\pi_{k_n} \cdots (\pi_{k_1} e) \cdots)$ where $\Pi = [\pi_{k_1}, \ldots, \pi_{k_n}]$

The method is specified as a set of transformation rules. It is the syntax of an expression that determines which rule to apply. For some syntactic constructs there are two rules. To choose between these two we must observe the types of the respective sub-expressions. The two rules differ in that one of them can only be applied when a certain sub-expression is of static tuple type while the other rule can be applied where this sub-expression is of any type. Choose the first, less general rule if possible. This difference is indicated in the rules by superscripting expressions with their types as described above. The σ denoting the static tuple type. The exception is variables for which there also are two rules but where the choice is based upon the length of the list of projections.

The arrow $\leadsto$ denotes one or many transformation steps. The rules apply to both expressions and patterns (if such a syntactic construct exists).

To initiate projection unfolding of an expression, e, apply the empty list of projections to it, $|e|^{[]}$. This is not possible if the residual type of that expression contains static tuples. There must be a balance between the static tuples and static projections so that they all can be removed. This is not a problem. It cannot be the intention that the residual code should have a static or partially static type!

The example above shows the spirit of the method but as a matter of fact, it works by removing the outer projections before the inner, successively recursing down the syntax tree. In the example, the first transformation step is therefore written as $|\pi_1((\pi_1\ f)(\pi_2\ a))|^{[]} \leadsto |(\pi_1\ f)(\pi_2\ a)|^{[\pi_1]}$.

As we shall see, both code and types will be changed. It is important that type correctness is maintained. We will not give a proof for this but rather give an informal argument for every rule why this is the case.

2.6.2 Static Tuples and Projection

First of all, how should the static tuples and projection be treated? They are after all the main target of this phase. A projection can only be applied to an expression of static tuple type but the expression itself need not be a static tuple. Therefore the operation of picking an element from a tuple must be postponed until the actual tuple is reached by recursing down the syntax tree. So an expression consisting of a projection applied to an other expression is dealt with by just putting the projection first in the list and continuing to projection unfold the other expression. The rule becomes

$$|\pi_i e|^{\Pi} \leadsto |e|^{(\pi_i : \Pi)}$$

Assuming type-correctness of the input program to this phase ensures us that when a static tuple in encountered, the list of projections is not empty. The first element of the list should be applied to the tuple.

$$|{<}e_1, \ldots, e_n{>}|^{(\pi_i : \Pi)} \leadsto |e_i|^{\Pi}$$

The rule for projections puts the projection in the list. Looking at the definition of what the list of projections is, reveals that the two sides of the $\leadsto$ are equivalent. The rule for static tuples just reduces a projection application without changing the type of the expression.

2.6.3 Variables

Recall the example. Where π_1 was applied to f, it was substituted with f_1. Actually, the projection was first put in the list of projections and then immediately consumed by the next transformation, which is described below.

Variables can have either a static tuple type or any other type. In the latter case, the list of projections applied will be empty and can be left as it is.

$$|x^\tau|^{[]} \rightsquigarrow x^\tau$$

If it is of type static tuple, the list of projections will not be empty. The variable denotes a term of static tuple type while the list of projections applied to that term denotes an element of any other type. The result of projection unfolding the variable must then denote that element. Construct a new variable name by adding the indices of the projections in the list to the original variable name.

$$|x^{<\sigma_1,\ldots,\sigma_n>}|^{(\pi_i:\Pi)} \rightsquigarrow |x_i^{\sigma_i}|^\Pi$$

We have now used a new variable name and it must be bound to the correct element. A variable name is bound either in a **let**-declaration, a **case**-expression or in a λ-abstraction. Corresponding changes must be done where the original variable name is bound. When the whole list of projections is consumed, the first rule for variables is applied.

Thereby $|v|^\Pi$ is distinct from any other $|v'|^{\Pi'}$ unless $v = v'$ and $\Pi = \Pi'$. The variable name $a_{2,1}$ in the example is constructed like this.

$$|a|^{[\pi_2,\pi_1]} \rightsquigarrow |a_2|^{[\pi_1]} \rightsquigarrow |a_{2,1}|^{[]} \rightsquigarrow a_{2,1}$$

This preserves type correctness as long as the declarations, **case**s and λ-abstractions are handled properly. The renaming just reflects which term is referenced.

2.6.4 Functions and Function Application

Variable names are changed if they refer to terms of static tuple type. This must be considered when transforming functions and function applications.

We identify two different cases of function application; when the argument *is* respectively *is not* of type static tuple. The solution to the former is arity raising[Rom90]. The argument can be split into several arguments, each corresponding to one element of the tuple. The splitting is achieved by applying projections to the argument. The function must also be changed so that it takes the new number of arguments.

$$|e_1\ e_2^{<\sigma_1,\ldots,\sigma_n>}|^\Pi \rightsquigarrow |e_1(\pi_1 e_2)^{\sigma_1}\cdots(\pi_n e_2)^{\sigma_n}|^\Pi$$

$$|\lambda x^{<\sigma_1,\ldots,\sigma_n>} \to e|^\Pi \rightsquigarrow |\lambda x_1^{\sigma_1} \to \cdots \to \lambda x_n^{\sigma_n} \to e|^\Pi$$

Note that no projections are consumed and that the lists of projections still are applied to the whole of the expressions, one of the σ_is might be a static tuple type. When they are not, the expression will be transformed by repeated application of the following two rules.

When the argument is not of type static tuple just apply the current list of projections to the body of the function. The intuition is that, it is the body that is responsible for making this whole application an expression of type static tuple. It cannot be the argument because it has just been transformed so that it is not of static tuple type.

$$|e_1\ e_2{}^\tau|^\Pi \leadsto |e_1|^\Pi\ |e_2{}^\tau|^{[\,]}$$

$$|\lambda x^\tau \to e|^\Pi \leadsto \lambda x^\tau \to |e|^\Pi$$

No λ-abstracted variable will be of static tuple type after the last transformation has been applied. Now the variables in the function body that are of static tuple type and was abstracted by the original λ-abstractions will be free but as the transformation carries on, this will be taken care of by the rules for variables.

When an argument to a function is a static tuple it is split into several arguments. The same thing is done with the abstracted variables in functions. Thus the body of the function, which is the result of the whole application, is not changed. A list of projections applied to a function application can be applied to the function body.

2.6.5 let-Expressions

The result of a **let**-expression is the result of the top level expression. Therefore the list of projections applied to a **let**-expression should be applied to its top level expression. The declarations must also be handled. A declaration might be of type static tuple and therefore, where the declared variable name is used, it will be changed to reflect which element in the tuple is referred to. The variable name in the declaration must also be changed. In the above example, a is declared to be a nested static tuple. There are four elements which would be referred to by the names

$$\begin{aligned} a_{1,1} &= 1 \\ a_{1,2} &= 2 \\ a_{2,1} &= 3 \\ a_{2,2} &= 4 \end{aligned}$$

One approach is that we could just produce these declarations. As we see in the example, $a_{1,1}$ and $a_{1,2}$ are never used. So instead, projection unfold the top level expression first, then we know which elements are referred to and which declarations are needed. Generate these declarations. The expressions in the new declarations might refer to further elements, so further new declarations might be needed and must thus be generated. Let v_Π be the variable name we get when performing $|v|^\Pi$.

$$\left|\begin{array}{ll}\textbf{let} & [x_i = e_i]_{i=1}^{n} \\ \textbf{in} & e_0\end{array}\right|^{\Pi} \leadsto \begin{array}{ll}\textbf{let} & [(x_{k_j})_{\Pi_j} = e'_{k_j}]_{j=1}^{m} \\ \textbf{in} & e'_0\end{array}$$

$$\begin{array}{lrl} \textit{where} & e'_0 & = |e_0|^{\Pi} \\ & \Big[(x_{k_j})_{\Pi_j}\Big]_{j=1}^{m} & = \Big[\ x_\Pi \mid x_\Pi \in \Big(fv(e'_0) \cup (\bigcup_{j=1}^{m} fv(e'_{k_j}))\Big) \ \bigwedge \ x \in [x_i]_{i=1}^{n}\Big] \\ & [(e'_{k_j})]_{j=1}^{m} & = [|e_{k_j}|^{\Pi_j}]_{j=1}^{m} \end{array}$$

The first equation says that to get the new top level expression, apply Π to the original.

The second tells which variables should be declared. m is the number of new declarations. The left operand to $\wedge$ in the list comprehension first selects the free variables of the new top level expression and then of the right-hand sides of *all* new declarations. The definition is thus circular. We must know the right-hand sides of all declarations to know which declarations to produce![6] There is thus a potential infinite loop here but the problem only arises when we have infinitely nested static tuples, which also gives infinite types. We shall not construct declarations for all the free variables. Only those produced of variables declared in *this* **let**-expression. This is controlled by the right operand to $\wedge$.

The third equation tells us that to get the right-hand side in the declaration of the new variable $(x_{k_j})_{\Pi_j}$ (where x_{k_j} is one of the original variables x_i), apply Π_j to the original right-hand side of the declaration of x_{k_j} which is e_{k_j}.

The type of the whole **let**-expression is the type of the top level expression. So if a list of projections can be applied to the whole **let**-expression, it can be applied to the top level expression without changing any types. The list of projections that were used to construct the new variable names from the names in the declarations, are the same list of projections that are applied to the corresponding expressions in the declarations. Thus the same list of projections is applied to both sides in the declarations and type correctness is maintained.

2.6.6 Constructor Application

As the type of a constructor application cannot be static tuple, the list of projections applied to such an expression must be empty. The arguments to the constructor might have type static tuple. These arguments can only be retrieved in a **case**-expression by pattern matching. The elements of the tuple are referred by projection application. Thus when projection unfolding these references new variable names will be constructed as described above. Therefore these new variable names must be introduced in the patterns. This it achieved by splitting the corresponding variable in the pattern into as many new variables as there are elements in the tuple thus arity

[6] This is analogous to the *pending* and *done* list in a traditional partial evaluator. If a specialisation of a function is needed, its name is put in the pending list together with the arguments to specialise it to. To produce the specialisations, pick an item of the pending and look if it is in the done list, then it has already been made and we can just continue. If it is not in the done list, perform the specialisation and add it to the done list. This specialisation might add new items to the pending list. Proceed until the pending list is empty.

raising the constructor. This must also be done where the constructor is applied. The rule for constructors and case expressions becomes

$$\left|C\ e_1 \cdots e_i^{<\sigma_1,\ldots,\sigma_n>} \cdots e_m\right|^{[\,]} \rightsquigarrow \left|C\ e_1 \cdots (\pi_1 e_i)^{\sigma_1} \cdots (\pi_n e_i)^{\sigma_n} \cdots e_m\right|^{[\,]}$$

$$|C\ e_1 \cdots e_m|^{[\,]} \rightsquigarrow C\ |e_1|^{[\,]} \cdots |e_m|^{[\,]}$$

$$|\mathbf{case}\ e_0\ \mathbf{of}\ [p_i \to e_i]_{i=1}^{m}|^{\Pi} \rightsquigarrow \mathbf{case}\ |e_0|^{[\,]}\ \mathbf{of}\ [|p_i|^{[\,]} \to |e_i|^{\Pi}]_{i=1}^{m}$$

Note that the patterns are also projection unfolded by the same transformation rules.

The first two rules both apply to expressions and patterns. The first rule might be applied zero or many times repeatedly depending on the depth of nested tuples. The second rule is always applied once for each constructor expression and pattern.

Constructors appearing in both patterns and applications in expressions are transformed by the same rule. If a constructor is transformed in one place, it will be so in all other places, including patterns.

2.6.7 Tuples

The previous discussion on constructor applications is valid in the case of dynamic tuples as well. It is only the empty list that can be applied to a dynamic tuple and its components must be projection unfolded because they can contain static tuples.

$$\left|(e_1 \cdots e_i^{<\sigma_1,\ldots,\sigma_n>} \cdots e_m)\right|^{[\,]} \rightsquigarrow \left|(e_1 \cdots (\pi_1 e_i)^{\sigma_1} \cdots (\pi_n e_i)^{\sigma_n} \cdots e_m)\right|^{[\,]}$$

$$|(e_1 \cdots e_m)|^{[\,]} \rightsquigarrow (|e_1|^{[\,]} \cdots |e_m|^{[\,]})$$

2.6.8 if-Expressions

The result of an **if**-expression is either of the branches. Therefore they must both be projection unfolded with the list of projections applied to the whole expression. The conditional is of type **bool** but might hold a term of static tuple type. It is projection unfolded with the empty list.

$$|\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2|^{\Pi} \rightsquigarrow \mathbf{if}\ |e_0|^{[\,]}\ \mathbf{then}\ |e_1|^{\Pi}\ \mathbf{else}\ |e_2|^{\Pi}$$

The type of an **if**-expression is the same as the type of its branches. Therefore the list of projections can be moved without types having to be changed.

2.6.9 Base Types

The list of projections applied to base type values must be empty and they should be left as they stand

$$|n|^{[\,]} \rightsquigarrow n \quad \textit{where } n \textit{ is an integer}$$

$$|s|^{[\,]} \rightsquigarrow s \quad \textit{where } s \textit{ is a string}$$

Primitive operations can be considered binary functions which can not have static tuple result types. Use the second rule for function application twice and derive

$$|e_1 \oplus e_2|^{[\,]} \rightsquigarrow |e_1|^{[\,]} \oplus |e_2|^{[\,]}$$

which tells that neither the operands nor the result can have static tuple type but the operands can contain terms of static tuple type and must therefore be projection unfolded.

Only empty lists of projections are allowed and no code is changed. The types do not change either.

2.6.10 Void

The element void should not be present in the code after void erasure, therefore no projection unfolding rule is really necessary. All the same, it has been included for completeness. One might want to apply the projection unfolding mechanism before the void erasure pass.

$$|\bullet|^{[\,]} \rightsquigarrow \bullet$$

Only empty lists of projections are allowed and no code is changed. The types do not change either.

2.6.11 Projection Unfolding the Example Using the Method

We can now use the rules to projection unfold the example. To begin with, apply the empty list to the entire program

$$\left|\begin{array}{lll}\textbf{let} & f & = <\lambda x \to x, \lambda x \to x> \\ & a & = <<1,2>,<3,4>> \\ \textbf{in} & \multicolumn{2}{l}{\pi_1((\pi_1\ f)(\pi_2\ a))}\end{array}\right|^{[\,]}$$

Apply the list which is applied to the whole **let**, to its top level expression

$$|\pi_1((\pi_1\ f)\ (\pi_2\ a))|^{[\,]} \rightsquigarrow \quad |(\pi_1\ f)\ (\pi_2\ a)|^{[\pi_1]} \rightsquigarrow$$

We have an application in which the argument is of type static tuple. It is therefore split.

$$|(\pi_1\ f)\ (\pi_1\ (\pi_2\ a))\ (\pi_2\ (\pi_2\ a))|^{[\pi_1]} \rightsquigarrow$$

The arguments are not of type static tuple so the list of projections is applied to the function.

$$|(\pi_1\ f)(\pi_1\ (\pi_2\ a))|^{[\pi_1]}\ |\pi_2\ (\pi_2\ a)|^{[\,]} \rightsquigarrow \quad |\pi_1\ f|^{[\pi_1]}\ |\pi_1\ (\pi_2\ a)|^{[\,]}\ |\pi_2\ (\pi_2\ a)|^{[\,]} \rightsquigarrow$$

Put the projections in the respective lists and rename the variables.

$$|f|^{[\pi_1,\ \pi_1]}\ |\pi_2\ a|^{[\pi_1]}\ |\pi_2\ a|^{[\pi_2]} \rightsquigarrow \quad |f|^{[\pi_1,\ \pi_1]}\ |a|^{[\pi_2,\ \pi_1]}\ |a|^{[\pi_2,\ \pi_2]} \rightsquigarrow$$

$$|f_1|^{[\pi_1]}\ |a_2|^{[\pi_1]}\ |a_2|^{[\pi_2]} \rightsquigarrow \quad |f_{1,1}|^{[\,]}\ |a_{2,1}|^{[\,]}\ |a_{2,2}|^{[\,]} \rightsquigarrow$$

$$f_{1,1}\ a_{2,1}\ a_{2,2}$$

The projection unfolding of the top level expression is now finished and we inspect it for free variables and find $f_{1,1}$, $a_{2,1}$ and $a_{2,2}$. Next step is to generate declarations for these. To generate the expression for $f_{1,1}$ we apply the list $[\pi_1,\ \pi_1]$ to f.

$$|{<}\lambda x \rightarrow x, \lambda x \rightarrow x{>}|^{[\pi_1,\pi_1]} \leadsto \quad |\lambda x \rightarrow x|^{[\pi_1]} \leadsto$$

Here, x is of type static pair and is thus split in two.

$$|\lambda x_1 \rightarrow \lambda x_2 \rightarrow x|^{[\pi_1]} \leadsto \quad \lambda x_1 \rightarrow |\lambda x_2 \rightarrow x|^{[\pi_1]} \leadsto$$

$$\lambda x_1 \rightarrow \lambda x_2 \rightarrow |x|^{[\pi_1]} \leadsto \quad \lambda x_1 \rightarrow \lambda x_2 \rightarrow |x_1|^{[\,]} \leadsto$$

$$\lambda x_1 \rightarrow \lambda x_2 \rightarrow x_1$$

The right-hand side of the declaration of $f_{1,1}$ is finished and we continue with $a_{2,1}$ and $a_{2,2}$ in a similar fashion giving

$$|{<}{<}1,2{>},{<}3,4{>}{>}|^{[\pi_2,\ \pi_1]} \leadsto \quad 3$$

$$|{<}{<}1,2{>},{<}3,4{>}{>}|^{[\pi_2,\ \pi_2]} \leadsto \quad 4$$

We have how generated declaration for all free variables in the top level expression. No new declaration where called for so it is time to re-assemble the **let**-expression.

$$\begin{array}{lll}\textbf{let} & f_{1,1} & = \lambda x_1 \rightarrow \lambda x_2 \rightarrow x_1 \\ & a_{2,1} & = 3 \\ & a_{2,2} & = 4 \\ \textbf{in} & \multicolumn{2}{l}{f_{1,1}\ a_{2,1}\ a_{2,2}}\end{array}$$

2.7 Removing Inherited Limits

As Mogensen argues in [Mog96] it is of great importance that limits (number of functions, arguments to functions, constructors, data types, etc.) in the program to be specialised, are not inherited into the residual program. One entity in the source code should, as well as be potentially completely removed, also hold the predisposition to become many entitys in the residual code. Therefore we need a way to express this 'entity raising'. Let us for example look at polyvariant function specialisation:

$$\begin{array}{ll}\textbf{let} & f = \textbf{poly}\ \lambda x \rightarrow \textbf{lift}\ x \\ \textbf{in} & (\textbf{spec}\ f\ 1, \textbf{spec}\ f\ 2)\end{array}$$

In a traditional partial evaluator the program would have specialised to

$$\begin{array}{lll}\textbf{let} & f_1 & = 1 \\ & f_2 & = 2 \\ \textbf{in} & \multicolumn{2}{l}{(f_1, f_2)}\end{array}$$

Our specialiser gives, before projection unfolding and void erasure

$$\begin{array}{ll}\textbf{let} & f = {<}\lambda x \rightarrow 1, \lambda x \rightarrow 2{>} \\ \textbf{in} & (\pi_1\ f\ \bullet, \pi_2\ f\ \bullet)\end{array}$$

In the first case one declaration in the **let** expression, becomes two. In the second case it is still one but a tuple, holding two definitions. These are two examples of representations of entity raising. In both cases, one can say that an inherited limit has been removed. One function definition has become two. Although, in the second case, there is still just one function name and one declaration. To fully exploit the fact that there are two function definitions there should be two names and declarations like in the first case. Projection unfolding can accomplish this task, returning[7]

$$\begin{array}{ll} \textbf{let} & f_1 \;\; = \lambda x \rightarrow 1 \\ & f_2 \;\; = \lambda x \rightarrow 2 \\ \textbf{in} & (f_1 \bullet, f_2 \bullet) \end{array}$$

Now consider the example

$$(\lambda f \rightarrow (\textbf{spec}\ f\ 1, \textbf{spec}\ f\ 2))\ (\textbf{poly}\ \lambda x \rightarrow \textbf{lift}\ x)$$

Traditional partial evaluation would not be able to specialise this expression at all, whilst our method still works.

$$(\lambda f \rightarrow (\pi_1\ f\ \bullet, \pi_2\ f\ \bullet))\ {<}\lambda x \rightarrow 1, \lambda x \rightarrow 2{>}$$

The function is represented by a tuple holding two definitions. Again, projection unfolding helps us to fully exploit a removed limit; this time the number of arguments to a function. (Of course both that the function takes more arguments and that more arguments are supplied at application.) This gives us[8]

$$(\lambda f_1 \rightarrow \lambda f_2 \rightarrow (f_1 \bullet, f_2 \bullet))\ (\lambda x \rightarrow 1)(\lambda x \rightarrow 2)$$

Now look at the two examples

$$\left| \begin{array}{l} \textbf{case}\ (1, {<}2,3{>}, 4)\ \textbf{of} \\ \quad (a, {<}b, c{>}, d) \rightarrow a + b + c + d \end{array} \right|^{[\,]} \rightsquigarrow \begin{array}{l} \textbf{case}\ (1,2,3,4)\ \textbf{of} \\ \quad (a,b,c,d) \rightarrow a + b + c + d \end{array}$$

$$\left| \begin{array}{l} \textbf{case}\ C\ 1\ {<}2,3{>}\ 4\ \textbf{of} \\ \quad C\ a\ {<}b,c{>}\ d \rightarrow a + b + c + d \end{array} \right|^{[\,]} \rightsquigarrow \begin{array}{l} \textbf{case}\ C\ 1\ 2\ 3\ 4\ \textbf{of} \\ \quad C\ a\ b\ c\ d \rightarrow a + b + c + d \end{array}$$

which show how the number of elements in tuples and arguments in constructor application can increase. This amounts to both expressions and patterns.

Other examples are **case**-expressions casing on polyvariantly specialised constructors where one branch becomes many branches and static constructors become static tuples.

In some examples, as with polyvariant **case** expressions, there is a natural and obvious way to express entity raising. In others, as in the example with static constructors, there are not. So we have to seek a representation that suites our purposes.

One criteria on the representation is that it should be one value, opposed to where polyvariant functions becomes many values, each resulting in a declaration. That

[7] Which after void erasure will be **let** $f_1 = 1;\ f_2 = 2$ **in** (f_1, f_2)

[8] Which after void erasure will be $(\lambda f_1 \rightarrow \lambda f_2 \rightarrow (f_1, f_2))\ 1\ 2$

will allow us to place it wherever we want, as a sub-expression to any expression. This leaves us with two possibilitys, a product or a sum. These possibilitys were both investigated when the rules for static constructor specialisation where stated. One approach was the $Void_n$ constructor, the other was the static tuple. In that case tuples seemed easier to handle and that amounts to the case of the polyvariant functions as well. Of course we could have designed a new feature of the language, after all, static tuples are not a part of the residual language, and they are going to be removed anyway. Something with association lists or indexed sets, but nay, we have to deal with the static tuples anyway and they really suites our purposes.

So by using static tuples and projections as a way of representing when one entity is specialised into several, we get, together with the projection unfolding mechanism, a powerful method for removing inherited limits.

3 Implementation

3.1 Type Checker

The type checker takes the syntax tree created by the parser as its input. It recurses through the tree assigning a type to every sub-tree, i.e. every sub-expression of the program. Just as we would do the type inference manually, applying inference rules repeatedly, looking at the syntax which is successively broken down into smaller pieces to know what rules to apply next; the function that accomplishes this task works in the same way. It is recursively defined with one definition for every syntactic construction of the source language, like the inference rules, and is syntax directed.

For example

$$(\lambda x \to x)(1)$$

The inference tree for this program is

$$\dfrac{\dfrac{x : \mathbf{int} \vdash x : \mathbf{int}}{\vdash \lambda x \to x : \mathbf{int} \to \mathbf{int}} \qquad \vdash 1 : \mathbf{int}}{\vdash (\lambda x \to x)\ 1 : \mathbf{int}}$$

The *(SAPP)* rule tells to check the types of the function and the argument. The function, being the identity function, puts no restrictions on the type of its argument, nor its result, more than the two types should be the same. All the same, it has the type $\mathbf{int} \to \mathbf{int}$.

The clue is *type variables.* Let α, $\beta, \ldots$ denote type variables. When applying the *(SAPP)* rule, the function should of course have a function type but between what types is not know. Let it therefore have the type $\alpha \to \beta$. The argument in the application must therefore have the type α and the whole program the type β. When checking the type of the function it is discovered that it should have the same argument and return type. α and β should denote the same type. Checking the argument results in that α should denote **int**. Thus, with $\alpha = \beta = \mathbf{int}$, we get a correct inference of the type of the program. The type of an expression can thus be discovered *after* it has been visited.

To be able to handle these type variables and to assign types to them we need two concepts, that of a *substitution* which is the environment that binds type variables to types (base type as well as complex types as well as other type variables) and of *unification* which is the mechanism that is responsible for adding items to the substitution and to keep it consistent.

It is easy to think of an inconsistent substitution; $\{\alpha = \mathbf{int}, \beta = \mathbf{string}, \alpha = \beta\}$ for example; and such is of course disastrous. A program that is not type correct should be rejected and that task is imposed on the unification algorithm. The principle is that as long as the substitution is consistent, the program is type correct.

If the program to check is large, the substitution can grow fairly large as well. Different techniques are used to make the checking for consistency as efficient as possible. This type checker implemented here is by no means especially efficient, the specialiser is not capable of handling large programs anyway, but sufficient.

The substitution will hold a lot of assignments between type variables and it is a tedious task checking the variables to see what types they refer to, as there could build up large graphs of references. We use a simple technique to make it less tedious.

To join, to *unify*, two types, if one is uninstantiated we could just set this variable to be equal to the other type. This would have the effect that to check if a variable is instantiated we must check all the components it is connected to through the graph. The graph would thus need to be double-directed as this type could appear anywhere in their corresponding graphs.

It is here the trick is applied, make these graphs into directed trees instead. When dealing with a type variable always refer to the root if its corresponding tree. If a variable is still uninstantiated the root element of the tree it is a part of is a variable (potentially itself). If it is instantiated, the root element of its tree will be the type it is instantiated to.

If two types should be unified, look up the root elements in their trees (the tree might be trivial, only consisting of one type) and unify them instead. If one of these elements is a type variable, assign to it the other element. If both variables have already been instantiated, they are compared in the obvious way. A string and an integer cannot be unified for example. If unification fails, an error should be reported and there is no point in continuing the type-checking, the program is not type correct.

A type can also be partially instantiated, as in the example with the identity function above, which had the type $\alpha \rightarrow \alpha$. It is known to be a function but not between what types. So to unify two function type, unify the two argument types and the unify the two result types.

When unifying two type variables using this improved method, their corresponding trees are merged together. From one of the variables point of view, the distance to the root is unchanged while from the others, it is increased with one. Compare this to the naïve method where unifying a variable which is part of a graph of size n and another type part of a graph of size m, would result in both types becoming part of graphs of size $n + m$.

A monad [Wad90] is used for the implementation. It holds the substitution and supplies the algorithm with fresh type variable names. These features are kept in a state. Exceptions are also included in the monad which is how the unification algorithm notifies unification errors. These and other features are used in the monad that controls the specialisation and will be explained there.

3.2 Specialiser

The specialiser is implemented in much the same way as the type checker. After all, it *is* too a type checker. Although a bit more complicated. Everything that has been said about the type checker also applies to the specialiser, so we will just describe what has been added.

The most obvious thing is that the specialiser also infers code, not only types. Further, for example, when the type checker handles a static **if**-expression it checks both branches that they will have the same type, which will be the type of the whole expression. When specialising the same expression, the conditional determines which branch should be the result. The other branch is not considered at all. Thus when type checking, the whole program is visited while when specialising this is not necessarily the case.

This example reveals another difference which is not so easily handled. When type checking, just unify the type of the branches and return that type. This type might be a variable. The branches might (and will probably) have different specialisations. If the type of the conditional is an uninstantiated variable, which branch should be used for further specialisation? For example in

$$\begin{array}{ll} \underline{\textbf{let}} & f = \underline{\lambda} b \to \textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 2 \\ \underline{\textbf{in}} & f\ \underline{@}\ \mathit{True} \end{array}$$

Type checking assigns the type **int** to both branches and gives the whole **if**-expression that type. The conditional is given the source type **bool** which unifies with the type of the argument *True*. When specialising, the conditional is static and must have the *residual* type *True* or *False* but we don't know which. The specialisation must then be delayed until the type of b has been instantiated. This calls for a special mechanism for delayed specialisations.

This is accomplished by the *demon* mechanism. A demon is a delayed specialisation, a specialisation that awaits a type variable to get instantiated. The demon return both code and a type, just like an ordinary specialisation. As we cannot yet perform the specialisation both the code and type are 'uninstantiated'. Considering the code this is accomplish by *forward references*.

When the type variable that the demon awaits eventually get instantiated, the specialisation will run (potentially creating new demons) and the resulting code will be placed in a table, indexed by the forward reference.

The demons are used in many situations. When using the **lift**- and **lifts**-operators we must know what to lift. As the residual code of a **lift**- or **lifts**-expression is the residual type of its argument.

Recall the definitions of the specialisation rules *(POLY)* and *(SPEC)*. The idea was to be able to specialise an expression with regard to several types. This was done by gathering the different specialisations into static tuples. There is though not at any point possible to know what types to specialise a **poly**-expression to and thus not possible to apply the *(POLY)* rule. A **poly**-function can be recursively defined and one specialisation would call for another.

So rather than using tuples during the specialisation we use *indexed sets*.

If we erroneously try to apply a non **poly**-function to arguments leading to different residual types, we would get a unification error when trying to figure out the type of it. So with the **poly**-function, let instead these distinct residual types be elements of a set and let further this set be the residual type of the function and thereby the result of every application of it. When the **spec** operator is encountered; its argument, which will be a **poly**-function, is specialised. A singleton set is constructed of this residual type. This singleton will be unified with other singletons, there will be one for every application. Unification is now easy, to unify two sets just join them. Duplicates are removed. This is an important criteria. Duplicates leads to code duplication, one specialisation would be made for each copy.

To detect duplicates, set elements must be compared individually. To compare two types we must know them but as seen before this might not be the case. It is possible that the set elements are uninstantiated type variables. This is dealt with by using *backtracking*. When adding a type value to a set, unify the value with one of the set elements. If this leads to a unification error later during the specialisation when these types are instantiated, backtrack and unify the type value with another set element and continue like this until no set element remains, then make the value a new member of the set. If no unification error is encountered, then these two types were the same.

Backtracking can be very expensive. It is important that these unification errors are detected as soon as possible. It was reported in [Hug96b] that it is important how specialisation of function application is performed. If the function is specialised before the argument, it will specialise with respect to a type variable and if the application is a **spec**-application that variable will unify with any type and the process can continue. When it is time for the argument to be specialised and its type is incompatible with the type which the variable was unified with, this branch will fail and the algorithm must backtrack. If we on the other hand specialise the argument first, the incompatibility in type will be discovered at once and lead to a failure. The important difference is that we have not yet commenced specialising the function. Hence, little is lost.

In section 3.3 we will present a mechanism for measuring the amount of backtracking and in section 4.1 we will look at an example which reveals the differences caused by how the specialisation is performed.

For every application of the **poly**-function the appropriate element must be chosen from the set. Therefore these sets are indexed. When all elements are known, then a static tuple can be made out of the set. The indices used to the set are mapped to places in the tuple so that the right projections can be inserted where the function is applied.

The same mechanism with sets are used for polyvariant sums. Every time the *In* constructor is applied, a singleton is created. These sets are unified as described above. When specialising the corresponding case expressions, create one branch for every set element.

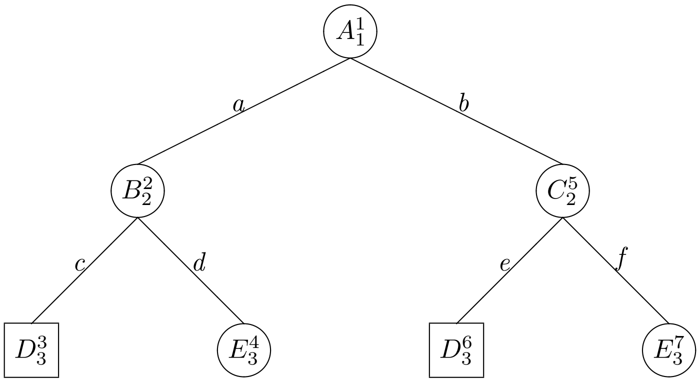

Figure 10: Tree corresponding to the computation $A + (B?C) + (D?E)$ where D fails.

3.3 The Success of Counting Failures

Our specialiser uses backtracking implemented by a monad. We have developed a method for measuring how much effort the specialiser spends on the actual solution and how much is wasted when the algorithm backtracks. Before we describe the monad used in the specialiser we describe a refined version of the measuring method.

Backtracking can be very expensive. As our specialiser is built on that principle, its performance relies heavily on how much time is spent in the failing branches of the computation. This is also identified as a general problem. Unification algorithms (as our specialiser, type-checkers and so on), parser and other applications also often uses backtracking. A measure on how well an algorithm based on backtracking works is the ratio between the effort spent in calculating the actual solution and the total effort. We have developed a mechanism for doing this.

3.3.1 Backtracking is a Tree

A computation using backtracking can be viewed as a tree where every node is a sub-computation. Every sub-computation can lead to failure or success. A computation is the activity of traversing the tree in pre-order. A path from the root ending in a succeeding leaf is a solution to the problem we are solving. A leaf which is a failure forces backtracking. Failing nodes can only be leaves in the tree as there is no point in continuing traversing the sub-tree of such a node.

As an example we can draw the tree representing the computation

$$A + (B?C) + (D?E)$$

We use the same notation as is used in [Wad85]. ? denotes *alternation* and + denotes *sequencing*. Let D be a failing sub-computation while the others are succeeding. The tree is shown in figure 10. A failing node is drawn as a square while a succeeding is drawn as a circle. There are two solutions to this problem. They are represented by the paths A-a-B-d-E and A-b-C-f-E in the figure.

3.3.2 The Cost of Finding a Solution

Finding a solution amounts to finding the leftmost succeeding path. In our example it is *A-a-B-d-E*. To address the subject of this section; What are the costs involved? If we consider visiting one node having unity cost, the total cost of finding the solution is 4. We have to visit nodes, A, B, D and E (in that order). It is only the nodes A, B and E that are involved in the actual solution. The cost to compute the solution is thus 3. We can draw the conclusion that in this particular example 25% of the effort made is wasted.

The numbering of the nodes in the tree represents the respective costs. The superscript represents the total cost of reaching that node and the subscript the cost of reaching that node by following this particular path. (The superscripts are the pre-order numberings and the subscripts the depths of the nodes.) We will refer to this two numberings as the *tree counter* respectively the *path counter*.

3.3.3 Implementing Backtracking and Path Counter

To represent backtracking we have adopted the principles described in [Wad85] where each step of a computation is represented by a list of solutions. The list corresponding to the example above is *[ABE, ACE]*.

We use a monad to implement our mechanism. The features backtracking and path counter are now easy to define. It is well known from literature how to construct such a monad [Wad95] [KW92]. Combine *state* with backtracking. This is shown in figure 11. Here, the state, S_P, is the path counter, which should be initiated to 0, i.e. the monadic computation should be applied to the initial state p_0. We also need a function, *tic*, which increases the counter. By constructing the two functions

$$ut\ a \;=\; tic\ \text{`}bind\text{`}\ \lambda() \;\rightarrow\; unit\ a$$

$$wt \;=\; tic\ \text{`}bind\text{`}\ \lambda() \;\rightarrow\; wrong$$

we can simulate the above example in the program

$$\begin{array}{l}(ut\ \text{"}A\text{"})\ \text{`}bind\text{`}\ \lambda x\ \rightarrow\\ (ut\ (x\ +\!\!+\ \text{"}B\text{"})\ \text{`}orelse\text{`}\ ut\ (x\ +\!\!+\ \text{"}C\text{"}))\,\text{`}bind\text{`}\ \lambda y\ \rightarrow\\ (wt\ \text{`}orelse\text{`}\ ut\ (y\ +\!\!+\ \text{"}E\text{"}))\end{array}$$

The result when applied to p_0 is $[(\text{"}ABE\text{"}, 3), (\text{"}ACE\text{"}, 3)]$, which tells us there are two solutions. Both has a path cost of 3. We take this monad as our starting point when we address the trickier part of the problem; managing the tree counter.

3.3.4 Principles for the Tree Counter

For example, when A has been evaluated, there is an alternation between B and C. The counter valid in A should be propagated to B but what should the value of the tree counter of C be? Here, our example expression can be rewritten

$$A + (B?C) + (D?E) \;\Leftrightarrow\; A + ((B + (D?E))?(C + (D?E)))$$

$$\begin{array}{l}
\mathbf{data}\ M\ a = M(S_P \rightarrow\ [(a, S_P)]) \\
\mathbf{type}\ S_P = \mathbf{int}
\end{array}$$

$$\begin{array}{l}
p_0\ ::\ S_P \\
p_0\ =\ 0
\end{array}$$

$$\begin{array}{l}
\mathit{unit}\ ::\ a\ \rightarrow\ M\ a \\
\mathit{unit}\ a\ =\ M(\lambda s \rightarrow [(a, s)])
\end{array}$$

$$\begin{array}{l}
\mathit{wrong}\ ::\ M\ a \\
\mathit{wrong}\ =\ M(\lambda s \rightarrow [])
\end{array}$$

$$\begin{array}{l}
\mathit{bind}\ ::\ M\ a\ \rightarrow\ (a\ \rightarrow\ M\ b)\ \rightarrow\ M\ b \\
(M\ x)\ \text{‘}\mathit{bind}\text{‘}\ f\ = \\
\quad M(\lambda s \rightarrow \mathit{foldr}\ \mathit{comb}\ []\ [\mathbf{let}\ (M\ y) = f\ v\ \mathbf{in}\ y\ s'\ \mid\ (v, s')\ \rightarrow\ x\ s])
\end{array}$$

$$\begin{array}{l}
\mathit{comb}\ ::\ [(a, S_P)]\ \rightarrow\ [(a, S_P)]\ \rightarrow\ [(a, S_P)] \\
\mathit{comb}\ =\ (+\!\!+)
\end{array}$$

$$\begin{array}{l}
\mathit{orelse}\ ::\ M\ a\ \rightarrow\ (M\ a)\ \rightarrow\ M\ a \\
(M\ x)\ \text{‘}\mathit{orelse}\text{‘}\ (M\ y)\ =\ M(\lambda s \rightarrow (x\ s)\ \text{‘}\mathit{comb}\text{‘}\ (y\ s))
\end{array}$$

$$\begin{array}{l}
\mathit{tic}\ ::\ M\ () \\
\mathit{tic}\ =\ M(\lambda s \rightarrow [((), s + 1)])
\end{array}$$

Figure 11: Definition of monad featuring backtracking and state.

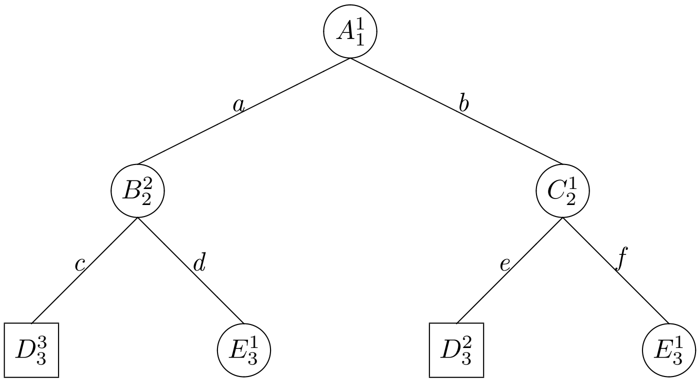

Figure 12: Tree with new numbering.

which more clearly reveals the evaluation order. The counter in C is dependent on the result of the evaluation of the sub-tree *BDE*. This is not how the computations are defined in our monadic implementation. In the latter, when constructing the sub-computation of C, we only know the result of evaluating B and this is evidently not sufficient.

What we do here is; pass the counter of A to B, just like it should be. The correct thing would then be to pass the counter of the left E to C. This we cannot do because it is yet unknown. Instead, pass a reseted counter to C. A reseted counter means a counter set to t_0 (or 0). This will result in the numbering in figure 12. It is always the left node of an alternation which gets the correct counter propagated. This leads to that it is only the leftmost path which is correctly numbered. This also holds for sub-trees.

We have split the counter into two parts which *sum* represents the correct value. Particularly, the sum of the counters of all leaves equals the total cost of exploring the whole tree. Also, for every leaf, the cost of reaching it equals the sum of the tree counters of all the leaves, including itself, to the left. So to reassemble the split counter we need only to perform a series of additions. In our example, the cost of reaching the rightmost E is thus $3 + 1 + 2 + 1 = 7$.

Note that by this method it is not possible to tell the tree counter of an arbitrary node but rather the tree counters of the ancestors of it which are leaves. Translated into counting costs, it is not possible to tell the cost of a sub-solution but rather the cost of a finished solution.

The problem here is that we do not keep the failing leaves, so we do not know the values of the tree counters in the two leaves named D!

If we would just eliminate failing nodes, counts would be lost. We need a way to delete the nodes themselves but still keep the counters. The solution we have adopted it to simply transfer the counter value to another counter and then delete the node. It is the brother node to the right which should get the contribution.

This is possible because as we established above; all failing nodes are leaves and has thus no sub-tree to traverse and also; at each node we know the tree counter of its left brother. So when we discover that a node has failed we can transfer its

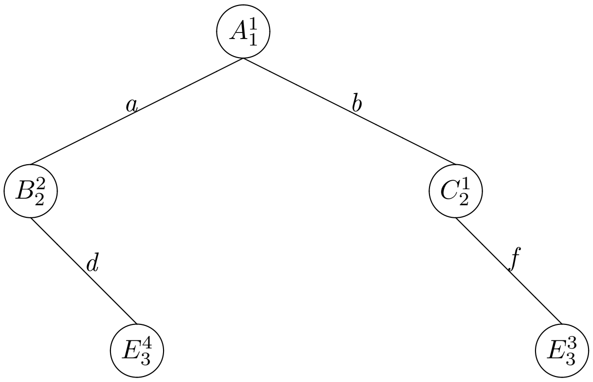

Figure 13: Tree with numbering corresponding to our implementation.

tree counter value to its right brother before it is deleted. That way we do not loose counts. This would give us the numbering in figure 13. In the result the failing leaves are deleted and to reconstruct the tree counter of the second solution add to it the counter of the first. The result is $4 + 3 = 7$.

3.3.5 Implementing the Tree Counter

How shall we extend our monad to cover this? First we must correct the data types. In the simple monad in figure 11 a (sub-)solution has the type (a, S_P), where S_P is the type of the path counter. A computation is represented by a list of such pairs. Now we also have a tree counter. Give it the type S_T and add it to the type analogous to how S_P is used. We then get the type

$$\textbf{data}\ M\ a = M(S_P \to\ S_T \to [(a, S_P, S_T)])$$

where the list represents the sub-solutions produced. This data type is not sufficient and the reason is that even an empty list of solutions is a result. A result which might have a cost associated with it. If the list is empty there is no tree counter which can hold this cost. Therefore we need an extra argument beside the list which can hold this cost. We get the type in figure 14 where the type R represents a result. The *Maybe* is there because it might be the case that there is no tree counter which is not associated with a solution. The new type, S_T, should also have a *zero*, t_0.

To fully understand this type, look at how *comb* is defined in figure 11. *comb* is used both for defining *bind* and *orelse* and is an operator that combines results. In this simple monad, where a result is a list of solutions, combining amounts to list append. Failure is represented by the empty list which works well with append.

In our new altered monad combining is not that simple. If one node fails than its tree counter should be propagated to its right brother. What if the right brother, the right operand to *comb*, is the failing node? Where should its tree counter be stored? The answer is; in the second argument to E. We call this un associated tree counter the *tree counter remainder*. Now a result is a list of solutions (possibly empty) and maybe a remainder. To combine results we use repeated application of *comb*. If one or more of the rightmost operands are failing nodes the cost of reaching

$$\begin{array}{l}
\textbf{data}\ M\ a = M(S_P \to\ S_T \to (R\ a)) \\
\textbf{data}\ R\ a = R\ [(a, S_P, S_T)]\ (\mathit{Maybe}\ S_T) \\
\\
\textbf{type}\ S_P = \textbf{int} \\
\textbf{type}\ S_T = \textbf{int} \\
\\
p_0\ ::\ S_P \\
p_0\ =\ 0 \\
\\
t_0\ ::\ S_T \\
t_0\ =\ 0
\end{array}$$

Figure 14: Data types for the altered monad.

them is not associated with any of the solutions in the list and are therefore summed and considered the remainder of this combined result.

comb should therefore have the following behaviour. If the left operand of *comb* contains a remainder, the remainder should be passed to the first solution, i.e. the first element of the list of solutions, of the right operand. If the right operand contains no solutions, the list is empty, the remainder should be combined with the remainder of the right operand. If the left operand contains no remainder, the lists of solutions are appended and the remainder (possibly non-existing) of the right operand becomes the remainder of the combined result.

The functions of the new monad is defined in figure 15. The main change is the definition of *comb*. All functions are of course rewritten to deal with this new counter as well. *tic* now increases both counters and there is also a *zero* for both counters.

In this altered monad the solution to our example is returned as the list

$$[(\text{"}ABE\text{"}, 3, 4), (\text{"}ACE\text{"}, 3, 3)]$$

To reconstruct the tree counter of the second solution add 4 to 3 and we acheive the correct result 7.

3.3.6 Generalising the Tree Counter

In the monad we started out with, the path counter was implemented as a state. The path counter inspired us when we implemented the tree counter so a natural conclusion must be that the tree counter is some kind of state too. How can it be generalised? Studying the solution, one notices that we must have an operator that combines these states in a meaningful manner and an identity for that operator. This is valid for our counter having addition as a state combiner and 0 for identity.

$$0 + t = t + 0 = t$$

```
unit :: a → M a
unit a = M(λp t → R [(a, p, t)] No)

wrong :: M a
wrong = M(λp t → R [] (Yes t))

bind :: M a → (a → M b) → M b
(M x) ‘bind‘ f =
   M(λp t →  let   R as mr = x p t
             in    foldr comb (R [] mr)
                        [let M y = f v in y p′t′ | (v, p′, t′) → as])

comb :: R a → R a → R a
a ‘comb‘ b =
   case a of
      R as mr →
         let (bs, mq) =
            case mr of
               Yes t →
                  case b of
                     R bs mq →
                        case bs of
                           [] →
                              case mq of
                                 Yes t’ → ([], Yes (t ‘tComb‘ t’))
                                 No → ([], Yes t)
                           ((v, m, t’) : bs’) →
                              (((v, m, t ‘tComb‘ t’) : bs’), mq)
               No →
                  case b of
                     R bs mq → (bs, mq)
         in R (as ++ bs) mq

tComb :: S_T → S_T → S_T
tComb = (+)

orelse :: M a → M a → M a
(M x) ‘orelse‘ (M y) = M(λp t → (x p t) ‘comb‘ (y p t_0))

tic :: M ()
tic = M(λp t → R [((), p + 1, t + 1)] No)
```

Figure 15: Function definitions for the altered monad.

This is also valid for lists in general and strings in particular. The operator for combining strings is list concatenation and its identity is the empty list (string).

$$[] +\!\!+ s = s +\!\!+ [] = s$$

By some simple changes in the definition in figure 14 and figure 15 we can produce output in any part of a computation using backtracking. All we need to change is the definitions of *tComb* and t_0 and add a function for producing output.

3.3.7 Examples

In the next section we will see some examples of specialisation. For every example we will also show the result of counting. The counter has been implemented in the specialiser so that it counts one tic for every application of a specialisation rule. In the end of the section will follow a discussion on the results.

3.4 Monad

A monad was constructed to keep track of the specialisation. It is defined as

$$\begin{array}{l}\textbf{data}\ M\ a = M(S_R \rightarrow S_P \rightarrow S_T \rightarrow (R\ a)) \\ \textbf{data}\ R\ a = R\ [(a, S_P, S_T)]\ (\mathit{Maybe}\ (\mathit{String}, S_T))\end{array}$$

where S_R, S_P and S_T are different states. S_R is a read state holding a list of variable names in scope. This list is used to avoid name clashes when specialising for example dynamic let expressions. Recall the specialisation rules and note where fresh variable names are inserted.

S_P is a state with several components. It is a *path state* according to the terminology in the previous section. It holds the path counter, the substitution of type variables, a counter for supplying fresh type variable names, the demons and the code generated by the demon mechanism.

If there is a failure in the specialiser forcing it to backtrack an error message is produced. This should not be shown if there is another path that succeeds. If, on the other hand, the whole computation fails the message should be available. Therefore error messages are handled by the tree state, S_T. The tree state also holds the tree counter.

A backtracking capability is added to the monad, where the list in the data type R holds the solutions. If the list is empty, all computations have failed. In that case the second component of R will be $\mathit{Yes}(e, s_T)$ where e is an error message. This feature can be considered the exception capability.

4 Examples

Now that we have described our specialiser, let us look at some examples and see how well it works in practice.

```
letrec power = \x n -> if n == 0
                       then 1
                       else x * power x (n - 1)
in     \y-> power y 3
```

Figure 16: The *power* function.

4.1 The *power*-Function

We start out with the example stated in the introduction of this paper, the power function, which we write as

$$\begin{array}{lllll}
\textbf{letrec} & power = \lambda n\ x \rightarrow & \textbf{if} & n == 0 \\
 & & \textbf{then} & 1 \\
 & & \textbf{else} & x\ \times\ power\, x\ (n\ -\ 1) \\
\textbf{in} & \lambda y \rightarrow power\ 3\ y & &
\end{array}$$

In this section we will switch to another layout when displaying program examples. The reason is that the code is cut directly from the in- and output of the specialiser. It now looks like figure 16. Of course we cannot use underlining to annotate the program. Therefore we introduce the notation used by the actual implementation. The keywords `case`, `if`, `let` and `letrec` are prefixed by a `u`, for `u`nfoldable, to denote that the corresponding expressions are static and should thus be unfolded. The sign `@` also implies staticness. It can be placed before a constructor, after the `\` of a λ-expression or a primitive operator, or as the application operator.

We will perform two slightly different specialisations on this example. First, recall the specialisation made in the introduction where the expression becomes a series of multiplications of the unknown term x. The function is specialised with respect to a known value of the exponent n.

To achieve this result, almost everything must be annotated as static so that it can be unfolded. First, both the `letrec` and the `if` expressions should be prefixed by `u`s for unfoldable. If the if-expression is static so must also its conditional be. The equality operator is therefore suffixed by `@`. The applications of the function *power* and thus the function itself should also be unfolded which calls for `@` as application operators and as a suffix to the λ-character `\`. The value of n is known and determines the depth of the recursion made by the unfolded calls to *power*. It should be kept known and the decreasing of n must therefore be static, otherwise we would fall into infinite recursion. The subtraction of n by one, is made static. The only things that should be residualised are the `x`s, the multiplications and the `1` representing the base case of the recursion. To residualise the `1`, we apply the `lift`-operator to it. The multiplication sign is left as it stands as primitive operators without the `@` suffix means dynamic operations. Also the λ-expression in the `in`-part of the `let` should be residualised. Now the whole program is annotated and looks like in figure 17. We use it as input to the specialiser and then get

```
\x-> x * (x * (x * 1))
```

```
uletrec
  power =
    \@x n->
      uif n ==@ 0
      then lift 1
      else x * power @ x @ (n -@ 1)
in
  \x-> power @ x @ 3
```

Figure 17: The annotated *power* function

```
letrec
  power =
    poly
      \x n->
         uif n ==@ 0
         then lift 1
         else x * spec power x (n -@ 1)
in
  \y-> spec power y 3
```

Figure 18: The annotated *power* function where `letrec` is residualised.

with residual type `IntD -> IntD` which stands for a function type from dynamic integers to dynamic integers. This is the result produced by the specialiser. Both the void erasure and projection unfolding mechanisms leave it unchanged as there are neither any void elements nor any static tuples.

4.1.1 Residualising the `letrec`

Let us make another specialisation of the same example and let it demonstrate the effect of void erasure and projection unfolding. Now we want to residualise the function definition. This implies that the `letrec` should be dynamic as well as the the function applications and the *power* function itself. Now *power* is used polyvariantly and we should therefore annotate this by using the operators **poly** and **spec**. The second annotated example is shown in figure 18. The result of running it through the specialiser is shown in figure 19. Here, `proj-`n means projection, earlier referred to as π_n, selecting the nth component of a static tuple. Counting starts at 0. `void` denotes the void element, earlier referred to as $\bullet$.

As *power* is polyvariantly specialised it is residualised as a tuple of function definitions, each definition referring to a specific specialisation with respect to different values of `n`. Projections are used to select the components. The projection unfolding mechanism splits the tuple into separate functions which are renamed according to the rules presented where the mechanism was explained. The projections on the

```
let
  power =
    <\x-> \n-> x * (proj-1 power x void),
     \x-> \n-> x * (proj-2 power x void),
     \x-> \n-> x * (proj-3 power x void),
     \x-> \n-> 1>
in
  \y-> proj-0 power y void
```

Figure 19: The output of the specialiser when specialising *power* where `letrec` is residualised.

```
let
  power_3 =  \x-> 1
  power_2 =  \x-> x * (power_3 x)
  power_1 =  \x-> x * (power_2 x)
  power_0 =  \x-> x * (power_1 x)
in
  \y-> power_0 y
```

Figure 20: The result after void erasure and projection unfolding.

tuple are replaced by a call to the appropriate function.

The `n`s which are arguments in the source code are all residualised as void elements as they are completely static. The λ-abstracted `n`s in the residual code applies to these void elements. This is recognised by the void erasure mechanism which deletes both the `void`s and the abstracted `n`s.

The code after the two passes is presented in figure 20 where all projections, static tuples and void elements are removed.

4.1.2 Counting

We will now refer to the results of the counting mechanism described earlier in this paper. First out is the version of the *power*-example depicted in figure 17 where the `letrec` is unfolded.

Running the example results in both the tree counter, C_T, and the path counter, C_P, showing the value 57. It is not surprising that they show the same result as there is no backtracking involved in this example. Running the example with some other values of the exponent `n` reveals that the value of the counters obey the relation

$$C_T = C_P = 14 \times \mathtt{n} + 15$$

The relation between the computational cost and the problem size is linear.

If we run the second version, shown in figure 18, where the `letrec` is residualised the cost is slightly higher. A value of 3 for `n` gives *both* counters a value of 65. The

n	C_P	C_T	C_P/C_T
0	17	17	1.00
5	97	157	0.62
10	177	397	0.45
15	257	737	0.35
20	337	1177	0.29
25	417	1717	0.24

Table 1: The counters and their ratio for different values of n when specialising the function before its argument.

equations for the counters are

$$C_T = C_P = 16 \times \mathtt{n} + 17$$

This might seem surprising though. The function is defined as polyvariant and polyvariant specialisations are achieved through the use of unification and backtracking. Wouldn't one expect a higher value of C_T? The explanation has already been given and is to be found in section 3.2. The reason is in how function application is specialised. It was argued that the argument should be specialised *before* the function so that unification errors could be discovered early.

To see what we avoid, change the behaviour of the specialiser so that it specialises the function before the argument. The results of the counters and their ratio is shown in table 1 for different values of n. As we can see, the computational cost is much higher. Inspection of the values of table 1 shows that C_T and C_P can be expressed by the following equations

$$C_P = 16 \times \mathtt{n} + 17$$

$$C_T = C_P + 4 \times \textstyle\sum_{i=0}^{\mathtt{n}} i$$

C_P is still the same but C_T has drastically changed. As $\sum_{i=0}^{\mathtt{n}} i = (\mathtt{n} \times (\mathtt{n} - 1))/2$ we get $C_T \in O(\mathtt{n}^2)$ which should be compared to the actual implementation where we instead have $C_T \in O(\mathtt{n})$.

4.2 Interpreter

By specialising an interpreter, written in the source language of the specialiser, with respect to a piece of code to interpret we achieve compilation to the target (residual) language of the specialiser. See the discussion on the first Futamura projection in the introduction.

We will see how well we can do when compiling using our specialiser.

Define a simple interpreter for a simply typed λ-calculus. Expressions in the language we interpret are built up from the data type

$$\mathbf{data}\ Ex = Ap\ Ex\ Ex \mid Lm\ \mathbf{int}\ Ex \mid Vr\ \mathbf{int} \mid Cn\ \mathbf{int}$$

where variables are identified by integer values rather than names. These expressions are handled in the interpreter injected in the universal type

```
ulet
  input =
    @Ap
      (@Lm 1 (@Ap (@Vr 1) (@Cn 3)))
      (@Lm 2 (@Vr 2))
in
  uletrec
    eval =
      \@env e->
        ucase e of
          @Cn n -> @Num (lift n)
          @Lm i e ->
            @Fun
              (\v->
                 ulet
                   env2 =
                     \@j-> uif i ==@ j then v else env @ j
                 in
                   eval @ env2 @ e)
          @Vr i -> env @ i
          @Ap e1 e2 ->
            ucase eval @ env @ e1 of
              @Fun f -> f (eval @ env @ e2)
  in
    eval @ (\@v-> @Wrong) @ input
```

Figure 21: The annotated interpreter.

$$\textbf{data}\ \mathit{Univ} = \mathit{Num}\ \underline{\textbf{int}} \mid \mathit{Fun}\ (\mathit{Univ} \mathrel{\underline{\to}} \mathit{Univ})$$

An annotated interpreter is shown in figure 21. where it is to be specialised with respect to the expression

$$(\lambda f \to f\ 3)(\lambda x \to x)$$

which is coded and given the name `input`.

The interpreter has as arguments the expression to interpret, `e`, and an environment, `env`. Constants are injected in the universal type by being tagged with `Num`. λ-abstractions are interpreted as functions with the body interpreted in an extended environment. The function is tagged with `Fun` and thus injected in the universal type. Variables are looked up in the environment. The first term of an application is interpreted and must result in a term tagged with `Fun` indicating it is a function. The detagged function is applied to the result of interpreting the second term of the original application expression.

An interpretation is initiated by applying the recursive function `eval` in an empty environment, i.e. an environment that maps everything to the error value `Wrong`, to the desired expression. (In this case `input`.)

To annotate the code we make in principle everything static. The only things that should be residualised are the numerical constants, the functions, the looked up values in the environment and the applications. Every trace of how the interpreter works should be removed leaving as result a compiled version of the input code. The output is

```
    (\v-> v 3) (\v-> v)
```

where the names `v` comes from the `Lm`-branch of the large `ucase`. This is exactly, minus renaming, what we input. Our specialiser is thus capable of removing an entire layer of interpretation.

4.2.1 Counting

When specialising this example, both counters stop at 154. There is no polyvariant specialisation so nothing else where to be expected.

4.3 Mogensen

Our last example was originally created by Torben Æ. Mogensen and was presented in [Mog93] to show the benefit of constructor specialisation. We use it to show the strength of our partial evaluator. As Mogensen points out, constructor specialisation is a most useful feature of partial evaluation. To show that our specialiser works just as well as Mogensens, we have tried to acheive the exact same residual code.

4.3.1 The Parser

Figure 22 shows a general parser for a restricted class of grammars. The grammar is specified by the data type `Gram`. `Empty` denotes the empty string. `Rec` refers to the entire grammar so that we can express recursive grammars. `Symb` is a unary constructor that has a terminal symbol, a string, as its argument. `Seq` stands for sequential composition. Finally `Alt` denote alternative.

We do not handle the type characters or lists in our specialiser and must therefore express the string to parse in another way. Define a new data type `StringS` to be a list of strings.

The function `parser` is the main function and takes two arguments, the grammar, `g`, and the string, `s`, to parse. It is the two functions `p` and `nowdo` that does the actual work. `p` takes four arguments; the grammar to use, the string to parse, the original grammar (which is the grammar to use when `Rec` is encountered) respectively a stack of things left to do. The stack is implemented by the data type `ToDo`. The string to parse by the data type `StringS`, which is a list of strings.

The execution starts with `parse` calling `p`. The latter look at the grammar to use to know what to do. If it is a terminal symbol, it should consume this symbol from the string and continue to try to parse what is on the stack. This is done by calling `nowdo` which pops the stack and calls `p` with a new grammar. Furthermore, if the stack is empty, the string ought to be empty as well. If the grammar given to `p` is sequential composition, it should first use the first grammar for parsing and put the second grammar on the stack so that it is the next thing to be done. Alternatives use

```
module Mogensen where

data Gram = Empty
          | Rec
          | Symb String
          | Seq Gram Gram
          | Alt Gram Gram

data ToDo = Done | Do Gram ToDo

data StringS = Nil | Cons String StringS

letrec  p = \g s g0 todo ->
               case g of
                 Empty -> nowdo todo s g0
                 Rec -> p g0 s g0 todo
                 Symb c -> case s of
                             Cons x xs -> (x == c) &&
                                            nowdo todo xs g0
                             Nil -> False
                 Seq g1 g2 -> p g1 s g0 (Do g2 todo)
                 Alt g1 g2 -> p g1 s g0 todo ||
                              p g2 s g0 todo
        nowdo = \todo s g ->
                  case todo of
                    Done -> s == Nil
                    Do g1 todo' -> p g1 s g todo'
in let g = Alt (Seq (Symb "(") (Seq Rec (Seq (Symb ")") Rec))) Empty
   in  let parse = \g s -> p g s g Done
       in  parse g
```

Figure 22: The program in its original form.

```
module Mogensen where

      ⋮

letrec up = \p nowdo g g0 s todo ->
               case g of

      ⋮

in
letrec  p = \g g0 s todo -> up p nowdo g g0 s todo
        nowdo = \todo s g ->

      ⋮

in let g = ...
   in  let  parse = \g -> \s -> p g g s Done
       in  parse g
```

Figure 23: The program extended to have one unfoldable and one non unfoldable definition of `p`.

backtracking, if the first grammar fails the `||` (logical or) -operator tries the second. If `p` is called with `Empty` as grammar, it should just ignore the input string and pop the stack to continue. When a reference to the entire grammar is encountered, call `p` with the original grammar as the grammar to use.

4.3.2 Specialising the Parser

The problem is to specialise this parser to a given grammar, $G \rightarrow (G)G|\epsilon$ where ϵ stands for the empty string. It is represented in the data type `Gram` by

```
Alt (Seq (Symb "(") (Seq Rec (Seq (Symb ")") Rec))) Empty
```

and is capable of generating any string of balanced parenthesis.

We must first decide what we want the residual code to look like. Here we have chosen to let Mogensen do this for us. All calls to `p` except the one done by `parse` and the one in the `Rec` case of `p` should be unfolded. We cannot both unfold and not unfold the same function with our specialiser due to the different approach we have adopted on unfolding functions and declarations. We must apply a trick to accomplish this. Make two definitions of `p`; one which is unfoldable, `up`, with basically the same function body as the original, and one non-unfoldable, `p`, which just calls `up`. Place these two functions in different **letrec**s, the unfoldable version in the outer, which will be annotated as static and thus unfolded. As `up` does not have either `p` or `nowdo` in scope they must be supplied as arguments. Figure 23 shows what this looks like. Now we are ready to annotate the parser.

4.3.3 Annotating the Parser

As already mentioned, the outer `letrec` as well as the function `up` it declares should be static, while the inner should not. The grammar is of course static and so is the stack, while the string to parse is not. `p` and `nowdo` must also be **poly**-functions. This causes the specialiser to loop infinitely. The reason is that the size of the stack depends on the string to parse and thus grows unboundedly. The stack must be made dynamic to cope with this problem. The grammar must still be static, otherwise no specialisation can be made. Mogensen uses constructor specialisation to deal with this. The constructor `Do` is specialised with respect to three different grammars

```
Seq Rec (Seq (Symb ")") Rec)
Seq (Symb ")") Rec
Rec
```

corresponding to the three different appearances of `Seq` in the grammar. The second argument of `Seq` is pushed onto the stack in `p`. It is now possible to specialise the parser. The annotated version is shown in figure 24. Note that neither `p` nor `nowdo` need be **poly**-functions as both the stack and the string are entirely dynamic.

4.3.4 The Residual Program

The result is the specialisation shown in figure 25. Looking at it, there are four specialised constructors; `In3` corresponding to `Done` and `In1`, `In0` and `In2` corresponding to the respectively specialisations of `Do` mentioned above.

This example shows that our specialiser is capable of handling constructor specialisation just as well as Mogensens.

The specialiser produces a function body but not the appropriate data type definitions. That is left for future work.

4.3.5 Counting

When running this example the two counters stops at a value of 361. Polyvariant constructor specialisation is performed but there is still no backtracking. The constructor *In* is applied in two places in the program, one is in the definition of `parse` where it is applied to `@Done` and the other is in the `@Seq` branch of the case expression in `up`. The latter will evaluated many times, the former only once. Every time the latter is evaluated, `g2` will have a different instantiated value and a unification error occurs immediately. No specialisation of type variables which will fail later.

5 Conclusions

It is obvious that the idea of propagating information through two channels, the residual code and the type, during specialisation is capable of giving strong specialisations. One can compare this to how the interpretive style works where information is propagated through the code alone. This can force static information to be thrown away. As an example consider the unannotated expression

```
module Mogensen where

data Gram = Empty
          | Rec
          | Symb SInt
          | Seq Gram Gram
          | Alt Gram Gram

data ToDo = Done | Do Gram ToDo

data StringS = Nil | Cons String StringS

f = uletrec up = \@p nowdo g s g0 todo ->
                   ucase g of
                     @Empty -> nowdo todo s g0
                     @Rec -> p g0 s g0 todo
                     @Symb c -> (case s of
                                   Cons x xs ->
                                     (x == (lifts c)) &&
                                       (nowdo todo xs g0)
                                   Nil -> False
                                 )
                     @Seq g1 g2 -> up @ p @ nowdo @ g1 @ s @ g0 @
                                     (In(@Do g2 todo))
                     @Alt g1 g2 -> (up @ p @ nowdo @ g1 @ s @ g0 @ todo) ||
                                   (up @ p @ nowdo @ g2 @ s @ g0 @ todo)
    in
    letrec  p = \g s g0 todo -> up @ p @ nowdo @ g @ s @ g0 @ todo
            nowdo = \dtodo s g ->
                      case dtodo of
                        In todo ->
                          ucase todo of
                            @Done -> s == Nil
                            @Do g1 todo' -> up @ p @ nowdo @ g1 @ s @ g @ todo'
    in ulet g = @Alt (@Seq (@Symb "(") (@Seq (@Rec) (@Seq (@Symb ")") @Rec)))
                  @Empty
       in ulet  parse = \@g -> \s -> p g s g (In(@Done))
          in  parse @ g
```

Figure 24: The annotated program that uses constructor specialisation.

```
let
  p =
    \s->
      \todo->
        case s of
          Cons x xs ->
            x == "(" && nowdo (In0 todo) xs
          Nil -> False
        ||
          nowdo todo s
  nowdo =
    \dtodo->
      \s->
        case dtodo of
          In0 todo -> p s (In1 todo)
          In1 todo ->
            case s of
              Cons x xs ->
                x == ")" && nowdo (In2 todo) xs
              Nil -> False
          In2 todo -> p s todo
          In3 -> s == Nil
in
  \s-> p s (In3)
```

Figure 25: The result of specialising the program using constructor specialisation.

$$(\lambda f \to f\ 2)(\lambda x \to \mathbf{lift}(x+1))$$

Suppose we want to make the outer application dynamic. Specialising this with the interpretive style would force all the other parts of the expression to be dynamic as well. No evaluation could be performed at all. The reason is that unfolding is so tightly connected to staticness due to the fact that only the code communicates the specialisation.

Using our method, where staticness and unfolding are independent, we make the following annotation

$$(\underline{\lambda} f \to f\, \underline{@}\, 2)\, \underline{@}(\underline{\lambda} x \to \mathbf{lift}(x+1))$$

and obtain the specialisation

$$(\lambda f \to f \bullet)(\lambda x \to 3) : \mathbf{int}$$

where the addition was possible to perform. After void erasure the expression is just

$$(\lambda f \to f)\ 3 : \mathbf{int}$$

The computer science community is more concerned with typed programs than untyped. It is therefore of great importance that partial evaluation should also be made on typed programs. We have here shown one solution to this.

6 Future Work

There are some things that remain to be done. The residual programs would be possible to compile in a Haskell compiler if it had not been for a couple of things. The specialised data types would not be recognised by the compiler. Data type definitions should therefore also be generated. The names of the specialised data type constructors are all In_n. It would be nice to have more natural names, reflecting the arguments to the specialised constructor.

The source language is an extension (as described above) of a subset of Haskell. One would maybe like to include the rest of the language.

The handling of static tuples are now a post-phase. It remains to be investigated whether this handling could be incorporated in the specialiser.

The examples which we have tested our partial evaluator on has been tiny toy programs. There remains a lot of work before one could even think of self-application. [Jon90] [JSS85]

6.1 Void Elements

There is a close relation between void erasure, arity raising (or variable splitting) [Rom90] and projection unfolding. Static constructors are specialised to static tuples. Constructors with arity 0, are therefore specialised to static unit, <>. As we have argued before, static constructors and static base type elements should be treated similar. They are not. A static integer is specialised to $\bullet$. It should also be specialised to static unit. So instead of constructing a one-point-domain with $\bullet$ as its only element, when specialising static base types, construct a one-point-domain consisting of <>. Now void erasure becomes a special case of arity raising, splitting the (static) tuple into its components. The thing is that the tuple has zero components and thus no variables appears. Should we call it arity *lowering*.

The projection unfolding mechanism described here, features arity raising and should thus be capable of also removing what hitherto has been call void elements.

This was late discovered and is therefore left for future work. The order in which the calculations are done in the current implementation of the projection unfolding, do not allow the trivial change that one would expect be possible.

Acknowledgements

I would first of all like to thank my tutor Professor John Hughes who has encouraged and inspired me a great deal. Thanks to Rogardt Heldal for the parser combinators and for showing such a great interest. Thanks to Lars Pareto for the parser. And everybody that has supported me during this period.

References

[BEJ88] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland Publishing Company, Amsterdam, 1988.

[Fut71] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[Hug95] John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.

[Hug96a] John Hughes. An Introduction to Program Specialisation by Type Inference. In *Functional Programming.* Glasgow University, July 1996. published electronically.

[Hug96b] John Hughes. Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *LNCS*. Springer-Verlag, February 1996.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2:323–343, 1992.

[JGS93] N. D. Jones, , C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[Jon90] N. D. Jones. Partial evaluation, self-application and types. In M. S. Paterson, editor, *Automata, Languages and Programming. 17th International Colloquium*, volume 443 of *Lecture Notes in Computer Science*, pages 639 – 659. Springer-Verlag, 1990.

[JSS85] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generation. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124 – 140. Springer-Verlag, 1985.

[KW92] David King and Philip Wadler. Combining monads. In *Glasgow Workshop on Functional Programming*, Ayr, Scotland, July 1992. Workshops in Computing, Springer-Verlag.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[Mog93] Torben Æ. Mogensen. Constructor specialization. In David Schmidt, editor, *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–32, June 1993.

[Mog96] Torben Æ. Mogensen. Evolution of partial evaluators: Removing inherited limits. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation. Proceedings*, Lecture Notes in Computer Science, page to appear. Springer-Verlag, 1996.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Rom90] S. A. Romanenko. Arity raiser and its use in program specialisation. In *Proc. 3rd European Symposium on Programming, Lecture Notes in Computer Science Vol. 432*, pages 341–360. Springer-Verlag, May 1990.

[Wad85] P. Wadler. How to Replace Failure by a List of Successes. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, Nancy, France, 1985.

[Wad90] P. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–77, Nice, France, 1990.

[Wad94] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI series, Series F: Computer and System Sciences.* Springer Verlag, 1994. Proceedings of the International Summer School at Marktoberdorf directed by F. L. Bauer, M. Broy, E. W. Dijkstra, D. Gries, and C. A. R. Hoare.

[Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995. (This is a revised version of [Wad94].).