Polyvariant Constructor Specialisation

Dirk Dussart,* Eddy Bevers, Karel De Vlaminck
Katholieke Universiteit Leuven
Departement Computerwetenschappen
Celestijnenlaan 200A
B-3001 Leuven
Belgium
{dirkd,eddy,kdv}@cs.kuleuven.ac.be

Abstract

In his PEPM'93 paper Mogensen introduced a new off-line specialisation technique, *constructor specialisation*. What differentiates constructor specialisation from conventional specialisation techniques is the ability to specialise constructors. Like functions, constructors are specialised with respect to their static arguments. As these constructors are also part of a program's type declaration, the effect of constructor specialisation is no longer limited to the algorithmic part of a program. Specialising constructors requires introducing new type definitions, specialised variants of the original definitions. The effect of constructor specialisation is, therefore, best understood as a combination of specialisation and a retyping transformation (changing the types of a program).

In this paper we develop a new constructor specialiser that solves some of the remaining problems in [Mog93]. Moreover, we show that this specialiser provides an alternative solution for a problem in [Lau91b], where a mix-style [JSS89] specialiser failed to remove the projection-injection overhead from specialised versions of the specialiser.

1 Introduction

Meta-programming, partial evaluation and program manipulation in general has been mainly studied in the context of dynamically typed languages. Very few attempts have been made to carry these results over to the world of statically typed languages. The reason is that statically typed languages lack the flexibility with which dynamically typed languages handle situations where values of different types are used interchangedly. The only way to achieve the same effect in statically typed languages is by introducing algebraic data types (a.k.a. sum types or disjoint unions): this involves introducing constructors or tags for distinguishing between the different types. This corresponds to mimicing dynamic typing at the program level. The effect on meta-programs, such as interpreters and specialisers, is that in the case where they have to deal with values of more than one type, values will have to be represented as part of a universal data type. This has severe effects on the size of values and hence also on the memory consumption of the interpreters and specialisers. But more importantly this encoding introduces an extra layer of interpretational overhead: values will have to be projected from and injected back into this universal domain. What is more, most existing specialiser fail to remove this overhead. The size problem has been dealt with in [Lau91b] and [DNBDV91], where delayed evaluation is used for keeping data structures from growing beyond manageable size. These solutions, however, do not tackle the interpretational overhead problem. The handwritten compiler generator approach from [HL91], on the other hand, addresses both problems in the limited case of specialised versions of the specialiser. Compiler generators never evaluate code and, therefore, need not represent values. But those derived by self-application do so! The solution in this particular case is simple: write the compiler generator by hand. But since most handwritten compiler generators still rely on mix-style technology, they are unable to cope with projection-injection overhead part of the input program (e.g. when generating a compiler from an interpreter: see the example in section 3).

This should raise some fundamental questions about the limits of mix-style specialisers. One major objection is the inherent difficulty of exploiting properties beyond the staticness, dynamicness or partial staticness of an expression. There are cases when an expression has to be classified dynamic according to the binding-time rules, but where this expression has some specific structure, e.g. an application of a constructor with some static arguments. According to the mix-style binding-time rules, these static arguments need to be coerced to dynamic expressions. Constructor specialisation liberalises this rule by allowing static arguments as part of dynamic constructor applications. A constructor specialiser will generate a new constructor, in which the static arguments have been incorporated. This single transformation can be viewed as a retyping transformation. Moreover, this retyping induces further changes in the residual program and requires the specialiser to propagate type information (i.e. the constructor plus its static arguments and the corresponding newly introduced constructor). One such induced change is the retyping of residual inspection-expressions: in order to reflect the type changes, residual case-expressions have to retyped as well. Because the type information includes static values, retyping a case-expression includes specialising the case-branches with respect to these static values. This means that constructor specialisation is capable of performing deeper specialisation. Because constructor specialisation is done off-line, it shows that deeper specialisation is not necessarily beyond the off-line framework. Furthermore, because of the ability to propagate type information constructor specialisation provides a way to remove some of the projection-injection attributed to interpreters and specialisers implemented in statically typed languages.

1.1 Overview

In section 1.2 we relate our work with that of Mogensen and others. Section 2 introduces the language of interest. Sec-

*Funded by the National Fund for Scientific Research (Belgium)

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1995 ACM 0-89791-720-0/95/0006...$3.50

PEPM '95 La Jolla, CA USA

tion 3 discusses specialisation of a simple meta-circular interpreter for an extended version of the language of section 2. In section 4 we develop a new binding-time analysis (*BTA* for short) which improves upon the analysis in [Mog93]. In section 5 we develop a highly modular specialiser, which in section 6 we apply to the meta-circular interpreter from section 3 in section 6.

1.2 Related Work

This work is based on [Mog93], where a constructor specialiser for a first-order subset of Standard ML with monomorphic data type declarations is developed. We address some of the remaining problems and weaknesses mentioned in the further work section of [Mog93]. Mogensen's specialiser is a specialiser in the mix tradition [JSS89], an off-line specialiser, which relies on a BTA for deciding between residualisation and specialisation. The quality of Mogensen's specialiser is, therefore, governed by the quality of its analysis. A first weakness of Mogensen's specialiser is certainly its analysis: all calls to the same function and all expressions having type T, where T is an algebraic type share the same binding-time property. This results in an analysis which is monovariant over different uses of the same function and constructor. The analysis developed in section 4 enhances Mogensen's by allowing a single constructor to have multiple binding-time properties (*bt-properties* for short), depending on the context it is used in. For the moment, functions are still treated monovariantly[1].

Constructor specialisation enhances specialisation in that it is able to exploit static argument information of dynamic constructor applications. Tracking constructor argument information paves the path to specialising or re-typing dynamic operations on algebraic data types (e.g., the case expressions). The result is deeper specialisation. However, it does not come for free and relies on the precision of the propagated types. The more precise this type information, the better residual programs and the less specialisation-time errors will occur. Mogensen's specialiser, however, performs some crude type approximations: all expression having type T, where T is an algebraic data type, share the same property or type. This facilitates later type definition generation a great deal: all specialised constructors from one and the same original type end up in one and the same specialised type. For this reason Mogensen's specialiser can be qualified as being monovariant over type definitions. Mogensen himself argues that his approximation may lead his specialiser to generate dead case branches, and even worse, to commit specialisation-time errors. Our solution to this precision problem is to combine specialisation with abstract interpretation. This gives us more precision, leads to better residual programs, and allows us to generate more than one specialised version for one and the same type definition.

The use of abstract interpretation in partial evaluation is certainly nothing new. Haraldson's Redfun [Har77] already included an abstract interpreter. Haraldson's work was later generalised and formalised in Consel and Khoo's work on parameterised partial evaluation [CK91]. Parameterised partial evaluation is a very general framework for reasoning about and describing the interaction of a set of abstract interpreters and a specialiser. Constructor specialisation, however, cannot be formulated in this framework as it requires abstract properties to be returned from residual function calls. Parametrised partial evaluation only computes facets (here abstract properties) for expressions that will be unfolded at specialisation-time[2]. An advantage of this restriction is that fixed point computation is avoided. Other work that comes close to ours[2], is Ruf and Weise's on avoiding redundant specialisation [RW91] by means of a non-standard type system (here fixed point computation is needed).

[1] Including function polyvariance in the typed-based analysis approach is still ongoing research.

2 The language

The subject language for the BTA in section 4 is an explicitly monomorphically typed (i.e., expressions carry extra type information) first-order functional language: see figure 1. A program is a sequence of recursive functions. Polymorphism can be added at the cost of some extra bookkeeping in both the specialiser and analysis. The language can be considered an explicitly typed variant of the one in [Mog93]. Note that explicit typing alleviates the need for type declarations.
An expression is either a constant (an integer, boolean or string), a variable, a primitive operation (e.g., integer addition), a function application, an application of a constructor or a case-expression.

For legibility reasons all example and residual program are written in a sugared version of the language in figure 1. All programs are written down as a sequence of type declarations followed by a sequence of function definition. The reader, familiar with ML or Haskell, should feel pretty at ease.

3 Specialising a Meta-circular Interpreter

One of the sole qualitive measures for specialisers is the so-called meta-circular interpreter test. In this test we specialise a meta-circular interpreter with respect to an input program and we expect the specialiser to remove all interpretational overhead. The closer the residual program to the input program the better the specialiser. In this section we will be specialising the meta-circular interpreter from figure 2 (an interpreter for a sugared version of the language in figure 1) with respect to a program for computing the length of an integer list: see figure 3.

A characteristic feature of all interpreters (including ours) and specialisers, operating on values of more than one type and written in a statically typed language[3] is the use of a universal type for values. In the case of our interpreter the universal type is "univ", a combination of booleans, integers, strings, algebraic values, etc. In order to implement the basic operations, we need coercions (so-called projections/injections) to and from this universal data type. See for example the code for interpreting an integer addition: the values returned from the recursive calls to `int` are first projected from "univ" to int, added together and afterwards injected into "univ". Because we specialise the interpreter with respect to a static program and a list of dynamic input values (the spine of the list is known), all operations on values turn out to be dynamic. And this includes the projection-injection operations. In other words, the corresponding residual program is a program operating on values of type "univ" and performing the necessary projections and

[2] Thanks to the PEPM-referee who pointed this out to us!

[3] with an ML-like type system

$$
\begin{array}{lcl}
c & \in & \textbf{Const} \\
x & \in & \textbf{Var} \\
p & \in & \textbf{Prim} \\
f & \in & \textbf{Fnames} \\
con & \in & \textbf{Cnames} \\
t & \in & \textbf{Types} \\
e & \in & \textbf{Expr}
\end{array}
$$

$$
\begin{array}{lcl}
t & ::= & \text{int} \mid \text{bool} \mid \text{string} \mid (t_1 * \ldots * t_n) \mid con_1 t_1 + \ldots + con_m t_m \mid t_1 \rightarrow t_2 \mid \mu x.t \\
e & ::= & c : t \mid x : t \mid p(e_1, \ldots, e_n) : t \mid f(e_1, \ldots, e_n) : t \mid con(e_1, \ldots, e_n) : t \\
 & & \text{case } e_0 \text{ of } con_1(x_1, \ldots, x_n) \Rightarrow e_1; \ldots; con_m(x_1, \ldots, x_n) \Rightarrow e_m : t \\
\text{Prog} & ::= & \{\ f_1(x_1 : t, \ldots, x_n : t) = e_1 \ldots f_m(x_1 : t, \ldots, x_n : t) = e_m\ \} \\
 & & \quad \text{(where } f_1 \text{ is the main function)}
\end{array}
$$

Figure 1: The Syntactic Domains

```
data univ = Bool bool | Int int | String string |
            Constr string [univ] | Error

data expr = BoolConst bool | IntConst int | StringConst string |
            Var string | Plus expr expr | ConstrApp string [expr] |
            Case expr [branch] | Fcall string [expr] | ...

int(progr,input) =
  let (types,funs) = progr
  in case funs of
       [] ⇒ Error;
       (f:fs) ⇒ let (name,args,e) = f
                in int_expr(e,zip(args,input),funs)

int_expr(e,env,funs) =
  case e of
    (BoolConst const) ⇒ Bool const;
    (IntConst const) ⇒ Int const;
    (StringConst const) ⇒ String const;
    (Var v) ⇒ lookup(v,env);
    (Plus e1 e2) ⇒ case (int_expr(e1,env,funs)) of
                     (Int i1) ⇒ case (int_expr(e2,env,funs)) of
                                  (Int i2) ⇒ Int (i1+i2);
    (ConstrApp c es) ⇒ Constr c (int_exprs(es,env,funs));
    (Case e brs) ⇒ int_branches(int_expr(e,env,funs),brs,env,funs);
    (Fcall fname es) ⇒ let vals = int_exprs(es,env,funs)
                       in let (fname,vars,e) = find(fname,funs)
                          in int_expr(e,zip(vars,vals),funs)
    ...

int_branches(val,brs,env,funs) =
  case brs of
    [] ⇒ Error;
    (b:brs) ⇒ let (constr1,vars,e) = b
              in case val of
                   (Constr constr2 vals) ⇒
                     if constr1 == constr2 then
                       int_expr(e,extend_env(vars,vals,env),funs)
                     else int_branches(val,brs,env,funs)
```

Figure 2: Excerpts of a meta-circular interpreter

injections. The result after arity-raising can be found in figure 4. This program shares not that much resemblance with the original length program and is, therefore, not the program we had hoped for! Most, if not all, projection/injection overhead is still part of the residual program. We conclude

```
data IntList = Nil | Cons Int IntList

data Nat = Zero | Succ Nat

length(xs) =
  case xs of
    (Nil) ⇒ Zero;
    (Cons x xs) ⇒ Succ(length(xs))
```

Figure 3: Length Program

```
data univ = Bool bool | Int int | String string |
            Constr string [univ] | Error

int_length(inp) =
  case inp of
    (Constr str vs) ⇒
      if str=="Nil" then Constr "Zero" []
      else if str=="Cons" then
             Constr "Succ" [(int_length(let [v1,v2] = vs in v2))]
           else Error
```

Figure 4: Specialised version of the interpreter

that the used specialiser is not optimal! What exactly went wrong? The length program's input is dynamic, which automatically implies that the result of the function **int_expr** is dynamic and hence also all expressions operating on the values returned from recursive calls to this function. From a type perspective, making an input dynamic means that the residual program can take any element of the original type, here the universal type, as input value. This is definitely too general a statement! Can we fix it? Yes, a more precise statement about the residual program is that its input, inp, is confined to integer lists, expressed in terms of the type "univ". This property can be expressed by means of the following inductive definition or grammar, where i has to be read as a placeholder for an arbitrary integer:

intlist ::= Constr "Nil" [] | Constr "Cons" [Int i,intlist]

In what follows we shall refer to this kind of properties as subdomain properties. The above subdomain property for *inp* allows us to optimise the residual program. First of all we are able to transform the case-expression. We can split

the pattern into two separate cases: one for the "Nil" argument case and one for the "Cons" argument case. What is more, because the string arguments of Constr are known, the string comparison and hence also the choice of an if-branch can be performed at optimisation-time. The result is the program from figure 5[4]. This new residual program is

```
data univ = Bool bool | Int int | String string |
            Constr string [univ] | Error

int_length(inp) =
  case inp of
   (Constr "Nil" []) ⇒
       Constr "Zero" []
   (Constr "Cons" [i,val]) ⇒
       Constr "Succ" [int_length(val)]
```

Figure 5: Specialised version of the interpreter (after optimisation)

very close to the original program, but contains some minimal amount of interpretational overhead: string matching. Fortunately, this can be avoided by introducing new constructors and corresponding type definitions. This whole process has been coined type specialisation by Launchbury in [Lau91b]. The program after type specialisation, see figure 6, and the original length program are equal (modulo renaming). Notice that two specialised versions of the original type "univ" have been introduced. Semantically the input/output behaviour of the residual programs has changed. This is not a problem because, if desired, we could always introduce wrappers that translate values from one representation to another.

```
data univ1 = Constr_Nil | Constr_Cons int univ1

data univ2 = Constr_Zero | Constr_Succ univ2

int_length(inp) =
  case inp of
   (Constr_Nil) ⇒
       Constr_Zero
   (Constr_Cons i val) ⇒
       Constr_Succ (int_length(val))
```

Figure 6: Specialised version of the interpreter (after type specialisation)

Constructor specialisation combines all of the above steps: at specialisation-time subdomain properties (in the form of grammars) are computed for each expression in the program. Then these grammars are used to specialise the dynamic case-expression (see for example figure 5). And finally they are used to generate the new type definitions.

[4] Generalising from this example, the reader might be tempted to conclude that the very same program can be extracted by a simple post-processor. For reasons, that will become apparent in section 5 this seemingly simple optimisation requires cleverness beyond most post-processors.

4 Binding-time analysis

In the previous section it was shown that by enriching the partial evaluation value domain with subdomain properties we are able to achieve deeper specialisation. Here, a subdomain property has to be interpreted as an extra constraint on the set of possible input values. In this section we develop a BTA which captures the use of subdomain properties in the specialiser.

Bt-time properties are decorated standard types and are inductively defined by the grammar in figure 7. At first

$$
\begin{array}{lcl}
BTType & \rightarrow & Type \\
 & | & \underline{Type} \\
 & | & (\underline{BTType} * \ldots * BTType) \\
 & | & \mu Tvar.BTType \\
 & | & Tvar \\
Type & \rightarrow & int \\
 & | & bool \\
 & | & string \\
 & | & c_1 BTType + \ldots + c_n BTType
\end{array}
$$

Figure 7: Binding-time types

sight the bt-properties from figure 7 correspond to Launchbury's projections in [Lau91a]. The correspondence is, however, only cosmetic. All Launchbury's projection can be translated to the bt-properties of figure 7. The projection ABSENT_τ, where τ is a base type is simply our $\underline{\tau}$, and his ID_τ is our τ. The translation carries through for projections corresponding to partially static values (i.e. products and recursive or non-recursive algebraic data types). On the other hand, Launchbury's projection $\text{ABSENT}_{c_1\tau_1+\ldots+c_n\tau_n}$ is translated into an underlined algebraic type in which also all the subcomponents τ_i (and their subcomponents and so on) are underlined[5]. In our BTA, however, we also allow underlined binding-time types in which not all of the subcomponents are underlined: this covers the case of a dynamic constructor application with static arguments. These kind of properties clearly cannot be interpreted as projections. An example of such a bt-property or type is that of a the dynamic list of integers represented as an element of type "univ" (refer to figure 2 for the definition of "univ"):

$$
\begin{array}{l}
\mu\text{univ. } \underline{\text{Bool bool}+} \\
\quad \underline{\text{Int int}+} \\
\quad \underline{\text{String string}+} \\
\quad \underline{\text{Constr string [univ]}+} \\
\quad \underline{\text{Error}}
\end{array}
$$

This bt-property has to be read as follows: at specialisation-time the corresponding values are completely unknown, but a constraint on the set of possible run-time values is given in the form a subdomain property. In this subdomain property all first arguments for `Constr` will be known. Let us stress the fact that this kind of property is considered incorrect in most conventional BTA's. The corresponding conventional bt-property would look like:

[5] This representation expresses better the intent of Launchbury's property than ABSENT, see the analysis of the case expression in [Lau91a].

μuniv. Bool bool+
Int int+
String string+
Constr string [univ]+
Error

where all subcomponents have been underlined.

The BTA is specified as a non-standard type system. The set of inference rules from figure 8 is a sound approximation to the set of requirements that a well-annotated program should satisfy, in order to guarantee binding-time error free constructor specialisation. We adopt Nielson style annotation [Nie88] for distinguishing between specialisation-time and run-time expressions: specialisation-time expressions are overlined whereas the others are underlined.

In all inference rules two environments are used, Δ binds function names to the function's respective bt-property (since the analysis is monovariant over different uses of the same function, each function is only attributed one single property) and Γ maps variable names to bt-property for the corresponding variables.

The most interesting inference rule is that for dynamic constructor applications: even though the application is classified dynamic there is no constraint on the arguments (they may be static although the complete structure is dynamic).

Other seemingly uncommon rules, are those for fold and unfold. By means of the fold rule uniformity for recursive data structures can be enforced: all components of a recursive data structure must have the same bt-property. Intentionally, the fold rule enforces equality of those bt-properties for which the underlying types have to be equal in the underlying type system. In fact, one of the main reasons for choosing the intentional approach was the ease with which such constraints can be specified[6]. Unfold simply unfolds the recursive bt-property one level. Both the fold and unfold rules are the logical counterparts of the rules for recursive data structures in [Lau91a]. For determinacy reasons folds will always be lined up with the dynamic constructor application and unfold with case-expressions.

Note that there is only one single rule for functions and function applications. No call or function annotations are provided during BTA. In our perspective, classifying functions as unfoldable or residual is best left to a separate phase. For what follows we will assume that this phase, which has been left out for space reasons, annotates functions and calls in some consistent way: a function is either residual or unfoldable and all calls to it are marked accordingly.

The main difference with Mogensen's analysis is that this one operates on explicitly typed programs, whereas Mogensen's operates on implicitly typed programs. We obtain constructor polyvariance through type copying, each expression of type T has its own copy of the definition for T. In Mogensen's analysis this definition is shared.

The inference rules from figure 8 can be given algorithmic meaning by observing that they induce constraints (equality-inequality constraints) on binding-time types. Binding-time analysis is the process, of finding a minimal decoration (a decoration with the least underlined parts) of the standard type inference tree given a set of assumptions, *inp*, for the input parameters of the main function, fd_1. In terms of constraints, binding-time analysis is the process of finding a minimal solution, i.e. a substitution, over the domain of bt-properties that satisfies the set of inequations and equations. This work extends Henglein's in [Hen91] for algebraic data types.

[6] On the other hand treating functions polyvariantly seems a lot easier in the extensional approach: see [Mog89]. We are currently investigating how the use of either conjunctive types or parametric polymorphism could help us with incorporating function polyvariance in the intentional approach.

5 The Specialiser

Constructor specialisation comprises specialisation and abstract interpretation. In this section we will see that this combination poses some extra problems. Because the results of the abstract interpretation affect specialisation and vice versa, and because of the recursive nature of our language some of the subdomain properties, can only be computed by means of fixed-point computation. A naïve way to perform this fixed-point computation is to re-iterate the whole specialisation process until a fixed-point is reached and a valid program is generated. This approach is expensive for the simple reason that it leads to the generation of useless code, code that may become obsolete in one of the next iterations. In [Mog93] Mogensen suggests that re-iteration can be avoided by backpatching; however no algorithmic details are provided. Our solution is to factor the specialisation process into separate phases. Phase one performs the abstract interpretation part (or ***abstract specialisation***) over the domain of values and subdomain properties. The fixed-point is computed by means of a minimal function graph computation [JM86]. If this phase terminates, the resulting minimal function graph contains the set of needed argument/result pairs for each program function. Phase two traverses the minimal function graph and generates for each of the argument/result pairs for residual functions a specialised variant. Phase three finally performs a union/find-based analysis to compute the set of needed type definitions for the newly introduced constructors.

5.1 Fixed-Point Computation

In the case of constructor specialisation an expression either partially evaluates to a value or to a piece of code and a subdomain property (the part of the domain the result may vary over at run-time). Like in conventional specialisers residual and unfoldable are treated differently. Calls to functions classified as unfoldable, are inlined and calls to residual functions will be specialised. The generation of duplicate specialised version is avoided by means of memoising (or caching) all argument descriptions a function has been specialised with. This only ensures termination in the situation the possibly infinite call tree can be represented by means of a finite call graph. Matters become more complicated in the constructor specialisation case, where a call to a residual function yields a specialised function plus a subdomain property for the function's result. To avoid having to recompute this property for all calls with the same call pattern, the specialiser caches both call pattern and results. Where does fixed-point computation enter the picture? Imagine we specialise a residual call to a recursive function. In the process of specialising the body of the function we encounter a recursive call with exactly the same set of argument descriptions as those for the top-level call – in other words both calls will be replaced with a call to one and the same function and more importantly the subdomain property for the second call is in fact the one we

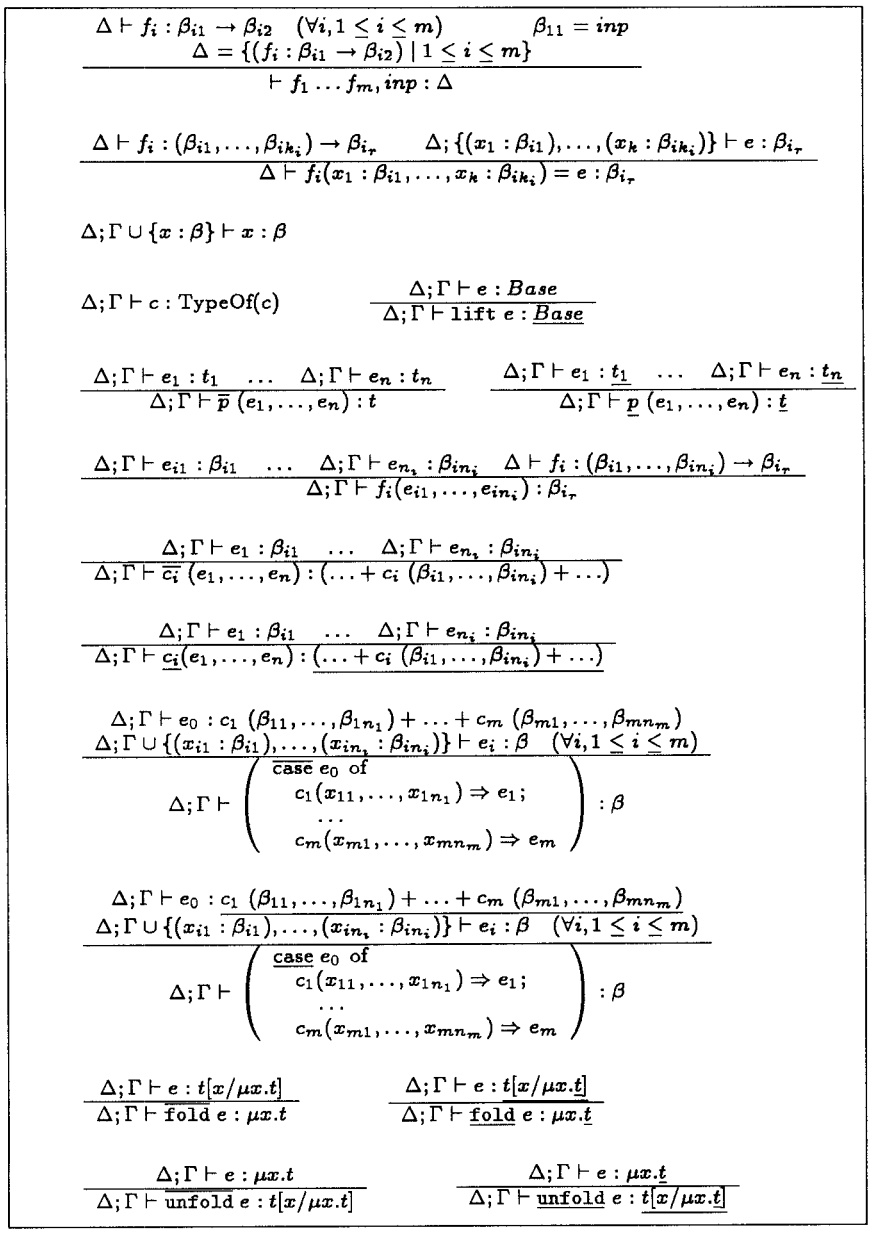

Figure 8: Binding Time Inference System

are still trying to compute! The only way to stop the specialiser from looping infinitely is to approximate this value and compute better approximations through a fixed-point process. In this section we develop an abstract specialiser (or interpreter) for computing these properties. The abstract interpreter manipulates ordinary values (completely static values), partially static values and subdomain properties. We represent all of them with grammars, see figure 9. A grammar is a (possibly empty) sequence of production rules, with as left-hand side non-terminal symbols and as right-hand sides base values (terminals), symbols (terminal or non-terminal), a constructor followed by a production, a tuple of productions, or a tagged sum of productions. We represent base values by grammars with a single production

$$
\begin{array}{llll}
 & & v \in & \textbf{BaseValue} \\
 & & s \in & \textbf{Symbol} \\
 & & c \in & \textbf{Cnames} \\
 & & p \in & \textbf{Production} \\
\text{Gram} & ::= & \{s_1 ::= p_1 \ldots s_n ::= p_n\} & \\
 & & \text{(where } s_1 \text{ is the start symbol)} & \\
p & ::= & v \mid s \mid c\ p \mid (p_1, \ldots, p_n) \mid c_1\ p_1 + \ldots + c_n\ p_n &
\end{array}
$$

Figure 9: The Domain of Grammars

rule { s_1 ::= v } and dynamic base values by the grammars

{ s_1 ::= h }, where h is a terminal symbol (placeholder) – this terminal symbol is an abbreviation of the set of possible values. A constructor followed by a production is used for partially static (or completely static) non-base values or subdomain properties. A sum of productions on the other hand is always used for subdomain properties. For our purposes the domain of grammars is a complete partial order, with $\leq$ defined as $G_1 \leq G_2$ iff $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$, where $\mathcal{L}(G)$ is the set of values generated by grammar G. The bottom element is the empty grammar. $G_1 \sqcup G_2$ is defined as the smallest grammar describing the values of $\mathcal{L}(G_1) \cup \mathcal{L}(G_2)$. For example {$s_1$ ::= Constr "Cons" [(Int i,s_1)]} $\sqcup$ {s_1 ::= Constr "Nil" [] } = {s_1 ::= Constr "Cons" [(Int i,s_1)] + Constr "Nil" [] }.

In step one we develop the notion of an abstract specialisation semantics. In step two we derive from this semantics a corresponding minimal function graph semantics, that will be used for computing the fixed-points. The abstract specialiser operates on two-level programs, a sequence of residual and/or unfoldable functions, with two-level expressions for function bodies. The syntax for two-level expressions can be found in figure 8, as part of the binding-time inference rules[7]. For convenience we abbreviate grammars by $[G ::= p]$, which reads as "the grammar with start symbol G and starting production p". The rest of the production rules remain hidden. Furthermore, we use as notation for environment extensions, $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]\rho$. As a matter of notational convenience we assume all functions and constructors to take the same number of arguments (this has to be relaxed for practical purposes). The rest of the notation should be pretty standard.

The abstract specialisation semantics in figure 10 uses two valuation functions, one for two-level programs $\mathcal{S}^{\#}_{\text{Prog}}$ and one for two-level expressions $\mathcal{S}^{\#}$. The abstract meaning of a two-level program is defined as the least fixed-point of a set of equations. The meaning of a constant is a grammar consisting of a single production rule and with the constant's standard semantics as production. The denotation of a lifted expression is a grammar with one production rule and a placeholder as production. Note, that the analysis specifies the grammar for e is computed and never used. This is simply because this definition is used as the basis for a minimal function graph semantics. The denotation of a static primitive application is the grammar with as production the result of the application in the standard semantics – note that v's in the grammars for the argument expression are also values in the standard semantics. The denotation for both a static and dynamic constructor application is a grammar built from the grammars for the arguments. So the treatment of static and dynamic constructor applications is the same. However, different are the ways in which the results are used: properties generated by a dynamic application can be subjected to a lub operation, while those generated by a static application never will. The difference between the denotation of a static and dynamic case expression lies in the denotation of the case's argument expression e_0. For a static case the denotation will always be a grammar with a production consisting of a single constructor applied to some other grammars. The constructor is then used to decide which branch to take. The argument of a dynamic case may evaluate to a sum of productions. This sum contains 0 or more entries for each construtor. For each of these sum elements the corresponding branch is abstractly specialised. The final result is the lub of all these denotations. Both residual and unfoldable function applications are treated in the same way by the abstract specialiser, as applications of the abstract version of the functions.

From this abstract specialisation semantics one can derive the corresponding abstract minimal function graph semantics along the lines of [JM86] – the derivation and semantics is fairly standard and omitted here for space considerations. This semantics is then used for computing argument/result pairs for each of the two-level functions (both residual and unfoldable) – those pairs that are actually needed during the abstract specialisation process.

This brings us to the issues of termination and generalisation. It should not come as a surprise that in some cases the minimal function graphs corresponding to a given two-level program and a set of input descriptions is infinite. In other words for these two-level program and those argument descriptions specialisation will not terminate. This is not only a problem that is solely related to constructor specialisation as conventional specialisers also suffer from this non-termination disease. Because in constructor specialisation more static information is preserved, the problem occurs more frequently. Non-termination of a specialisation is generally solved in off-line specialisation by going back to the original program and manually inserting generalising coercions (operations that force the specialiser to lose static information) into the program. The same technique applies for constructor specialisation. It is, however, not always necessary to use as drastic an approach. Some situations can be handled within the tight constraints imposed by the annotations. It is often the case that the grammars for subdomain properties grow beyond control, e.g. grammars for dynamic recursive data structures. Generalisation can be of help here. We therefore introduce the notion of approximation for grammars. Grammar G_1 is an approximation for grammar G_2, iff $G_2 \leq G_1$. The above definition of approximation is, however, not adequate for off-line specialisation: the set of approximations for a given grammar includes grammars that are not valid given a fixed bt-property. We therefore, introduce the notion of approximation under binding-time preservation. A grammar G_1 approximates G_2 with binding-time preservation, iff G_1 approximates G_2 and the bt-properties corresponding to G_1 and G_2 are equal. Approximation may have positive effects on the convergence of the fixed-point iteration but may not be applied unwieldy in the case of constructor specialisation: approximation equals loss of precision, precision which is crucial for generating correct code. Therefore, approximation may lead to the generation of dead code and, even worse, to specialisation-time errors[8]. An approximation we have found to be useful is the approximation of grammars for dynamic recursive data structure with some degree of redundancy, by means of flat grammars. A flat grammar is a grammar without nested recursive levels. For instance {s_1 ::= Constr "Cons" [(Int 1,s_2)], s_2 ::= Constr "Cons" [(Int 1,s_3)], s_3 ::= Constr "Cons" [(Int 2,s_4)], s_4 ::= Constr "Nil" [] } is approximated by the flat grammar {s_1 ::= Constr "Cons" [(Int 1,s_1)] + Constr "Cons" [(Int 2,s_1)] + Constr "Nil" [] }. In this particular example the original grammar contains some redundancy because 1 occurs twice as list element. We should

[7] To simplify later code generation all two-level expression carry their respective bt-property.

[8] The reason Mogensen's specialiser generates dead code or commits specialisation-time errors is simply because it uses poor approximations.

Semantic Domains

$e \in \mathbf{TwoLExpr}$
$v \in \mathbf{Value}$
$G \in \mathbf{Grammar}$
$\rho \in \mathbf{Env} = \mathbf{Var} \to \mathbf{Grammar}$
$\phi \in \mathbf{FnEnv} = \mathbf{FnId} \to \mathbf{Grammar}^n \to \mathbf{Grammar}$

Valuation Functions

$\mathcal{S}^{\#}_{\text{Prog}} : \mathbf{TwoLProg} \to \mathbf{FnId}$
$\mathcal{S}^{\#} : \mathbf{TwoLExpr} \to \mathbf{FnEnv} \to \mathbf{Env} \to \mathbf{Grammar}$

$$\mathcal{S}^{\#}_{\text{Prog}}[\![\{f_1(x_1:\beta_{11},\ldots,x_n:\beta_{1n}) = e_1 : \beta_1 \ldots f_m(x_1:\beta_{m1},\ldots,x_n:\beta_{mn}) = e_m : \beta_m\}]\!]$$
$$\text{fix}\lambda\phi.\{\ f_1 \mapsto \lambda(G_1,\ldots,G_n).\mathcal{S}^{\#}[\![e_1:\beta_1]\!]\phi\{x_1 \mapsto G_1,\ldots,x_n \mapsto G_n)\},\ldots,$$
$$f_m \mapsto \lambda(G_1,\ldots,G_n).\mathcal{S}^{\#}[\![e_m:\beta_m]\!]\phi\{x_1 \mapsto G_1,\ldots,x_n \mapsto G_n)\}\}$$

$$\mathcal{S}^{\#}[\![c:\beta]\!]\phi\rho = [G ::= [\![c]\!]]$$
$$\mathcal{S}^{\#}[\![\textit{lift}\ e : \underline{\beta}]\!]\phi\rho = \text{let } G_1 = \mathcal{S}^{\#}[\![e:\beta]\!]\phi\rho \text{ in } [G ::= h] \text{ (where } h \text{ is a place holder)}$$
$$\mathcal{S}^{\#}[\![x:\beta]\!]\phi\rho = \rho(x)$$
$$\mathcal{S}^{\#}[\![\overline{p}(e_1:\beta_1,\ldots,e_n:\beta_n):\beta]\!]\phi\rho = \text{let } [G_1 ::= v_1] = \mathcal{S}^{\#}[\![e_1:\beta_1]\!]\phi\rho$$
$$\vdots$$
$$[G_n ::= v_n] = \mathcal{S}^{\#}[\![e_n:\beta_n]\!]\phi\rho \text{ in} [G ::= [\![p]\!](v_1,\ldots,v_n)]$$
$$\mathcal{S}^{\#}[\![\underline{p}(e_1:\beta_1,\ldots,e_n:\beta_n):\beta]\!]\phi\rho = \text{let } G_1 = \mathcal{S}^{\#}[\![e_1:\beta_1]\!]\phi\rho$$
$$\vdots$$
$$G_n = \mathcal{S}^{\#}[\![e_n:\beta_n]\!]\phi\rho \text{ in} [G ::= h] \text{ (where } h \text{ is a place holder)}$$
$$\mathcal{S}^{\#}[\![c_i(e_1:\beta_1,\ldots,e_n:\beta_n):\beta]\!]\phi\rho = \text{let } G_1 = \mathcal{S}^{\#}[\![e_1:\beta_1]\!]\phi\rho$$
$$\vdots$$
$$G_n = \mathcal{S}^{\#}[\![e_n:\beta_n]\!]\phi\rho \text{ in } [G ::= c_i(G_1,\ldots,G_n)]$$
$$\mathcal{S}^{\#}[\![(\overline{\text{case}}\ e_0:\beta_0 \text{ of } c_1(x_1,\ldots,x_n) \Rightarrow e_1:\beta;\ldots;c_m(x_1,\ldots,x_n) \Rightarrow e_m:\beta):\beta]\!]\phi\rho =$$
$$\text{let } [G ::= c_i(G_1,\ldots,G_n)] = \mathcal{S}^{\#}[\![e_0:\beta_0]\!]\phi\rho \text{ in } \mathcal{S}^{\#}[\![e_i:\beta]\!]\phi([x_1 \mapsto G_1,\ldots,x_n \mapsto G_n]\rho)$$
$$\mathcal{S}^{\#}[\![(\underline{\text{case}}\ e_0:\beta_0 \text{ of } c_1(x_1,\ldots,x_n) \Rightarrow e_1:\beta;\ldots;c_m(x_1,\ldots,x_n) \Rightarrow e_m:\beta):\beta]\!]\phi\rho =$$
$$\text{let } [G ::= c_1(G_{111},\ldots,G_{11n}) + \ldots + c_1(G_{1r_11},\ldots,G_{1r_1n}) + \ldots +$$
$$c_m(G_{m11},\ldots,G_{m1n}) + \ldots + c_m(G_{mr_m1},\ldots,G_{mr_mn})] = \mathcal{S}^{\#}[\![e_0:\beta_0]\!]\phi\rho$$
$$\text{in } \bigsqcup_{i=1}^{m}\bigsqcup_{j=1}^{r_i} \mathcal{S}^{\#}[\![e_i:\beta]\!]\phi([x_1 \mapsto G_{ij1},\ldots,x_n \mapsto G_{ijn}]\rho)$$
$$\mathcal{S}^{\#}[\![f_i(e_1:\beta_1,\ldots,e_n:\beta_n):\beta]\!]\phi\rho = \text{let } G_1 = \mathcal{S}^{\#}[\![e_1:\beta_1]\!]\phi\rho$$
$$\vdots$$
$$G_n = \mathcal{S}^{\#}[\![e_n:\beta_n]\!]\phi\rho \text{ in } \phi(f_i)(G_1,\ldots,G_n)$$

Figure 10: Abstract Specialisation Semantics

be aware of the fact that precision is lost by going to the flat representation, and might lead to specialisation-time errors (e.g., if the code manipulating the above list relies on the first list element being 1). Redundancy checks are best introduced at the program points where grammars for recursive data structures are built from existing components, i.e., at the dynamic constructor application points. To avoid having to check at each constructor application, including the

non-recursive cases, the checks are best introduced at the fold (see figure 8), since they are always lined up with the dynamic recursive constructor applications.

5.2 Code Generation

In the first phase we have computed the minimal function graph for a given two-level program and a set of input descriptions. When restricted to activation/result pairs for the residual functions, the minimal function graph is the final cache computed in the corresponding naïve constructor specialiser. The difference with a naïve constructor specialiser is that we have avoided code generation during cache computation. Since we know the complete cache, specialisation is reduced to generating a specialised version for each of the activation/result pairs for the residual functions in the graph. A simple graph traversal suffices (see figure 11: for each activation $(G_1, \ldots, G_n)$ of $\underline{f_i} \in \mathrm{Dom}(\mathcal{M})$ a specialised version is generated).

In the specification we make great use of auxiliary functions. Since most of them are pretty common in the partial evaluation literature they are not formally specified. Stat and dyn are functions that convert partially static values into their static, respectively dynamic part. The result of stat is used for generating new names for specialised functions and constructors in (residual) applications. The result of dyn (which is in general a list) makes up the list of arguments for the specialised function and constructor applications. $\mathrm{Stat}_{\mathcal{G}}$ does essentially the same as stat, but for grammars instead of partially static values. It is used in the generation of new constructor names in the (left hand sides of) case branches and of new function names for the (residual) function definitions. Finally stat+var is a function that replaces all dynamic parts by new variables, and also returns the list of newly introduced variables. Both results are needed in the construction of specialised case branches and specialised function definitions. More details can be found in [Lau91a], where exactly the same factorisation is described. Only the factorisation for a dynamic property with static subcomponents (to which no projection corresponds) must be treated specially. Such a property is considered to be completely dynamic in our factorisation. We use a function for generating new constructor names (gen-constr) from a given constructor and a set of static values, and a function for generating new function names (gen-fnid) from a given function name and a set of static arguments. Furthermore, there are functions for generating residual function definitions, gen-fundef, and functions for generating residual expressions: gen-const (converts base values into constants), gen-primap (generates primitive applications), gen-constrap (generates constructor applications), gen-case (generates case-expression), gen-funap (generates function applications) and gen-branch (generates case-branches).

In the specification for $\mathcal{G}$ in figure 11 and figure 12 we use three environments, ϕ a function environment, θ an environment that binds variables to grammars and ρ an environment that binds the same set of variables to two-level values. The point of having the extra θ environment is to propagate the subdomain properties. For example at the point of a dynamic case-expression we need the subdomain property for the expression e_0. This, basically, means we have to rerun the abstract specialiser, $\mathcal{S}^{\#}_{mfg}$[9], in this environment θ. In order to reflect environment changes it also means that we shall have to rerun the abstract specialiser for the arguments of an unfoldable function call. This definitely requires some recomputation! But the point is that it is only very local, since we have computed the minimal function graph, $\mathcal{M}$, in a previous step.

Because of the resemblance with the specification in figure 10 we only discuss the interesting parts of the specification in figure 11 and figure 12. For example, handling the application of a dynamic primitive p involves generating code for the arguments and building a new application of p. For a dynamic constructor application we first specialise the arguments, then we generate from the static part and the constructor a new constructor. Finally, we build a new constructor application with the dynamic parts of the argument as new arguments. The most interesting case is the dynamic case-expression. First we specialise the case's argument, and compute the corresponding subdomain property with $\mathcal{S}^{\#}_{mfg}$. For each of the sum elements of this property we generate a new case-branch, that matches against the new constructor associated with this sum element, and in which the body is specialised with respect to the static parts of the subdomain property. These branches are then combined into a new case-expression. The treatment of both residual and unfoldable function calls corresponds to this found in existing specialisers with that difference that subdomain properties are generated for the arguments.

After the minimal function graph is traversed we end up with a specialised version of the original program, a program without type definitions.

5.3 Generating Type Definitions

The programs generated by the code generators cannot be type checked by a standard ML-type checker. The residual programs lack type definitions. These definitions can be generated by inspecting the residual program and the grammars and dividing the set of of newly introduced constructors into equivalence classes by means of a union/find based analysis. The starting assumption is that all new constructors end up in separate equivalence classes. Two equivalence classes will be joined into one equivalence class if we can proof that the type checker will need a type consisting of elements of both classes (e.g., take the example in figure 4 where one branch of the case-expression evaluates to *Zero* and the other to *Succ* followed by a set of arguments). Further algorithmic details of the analysis are left out of the paper for space reasons. They will be included in an extended version. It should be investigated whether the use of Mishra and Reddy's declaration free type checker [MR85] on program generated by the code generator alleviates the need for type definition generation!

6 Example revisited

The ability to propagate and exploit subdomain properties about dynamic values is what makes constructor specialisation superior to conventional techniques. In section 3 we saw that a mix-style specialiser failed to remove any of the projection-injection overhead from a specialised meta-circular interpreter. We also discussed how subdomain properties could be used to optimise the specialised interpreter. In this section we will show that the program from figure 6 can be obtained by the specialiser from section 5.

[9] $\mathcal{S}^{\#}_{mfg}$ is the version of $\mathcal{S}^{\#}$ that is used in the minimal function graph computation.

Semantic Domains

$e \in \mathbf{TwoLExpr}$
$v, w \in \mathbf{TwoLValue} = \mathbf{Value} + \mathbf{Expr}$
$\mathcal{M} \in \mathbf{MFGraph} = \mathbf{FnId} \to \mathbf{Grammar}^n \to \mathbf{Grammar}$
$G \in \mathbf{Grammar}$
$\rho \in \mathbf{Env} = \mathbf{Var} \to \mathbf{TwoLValue}$
$\phi \in \mathbf{FnEnv} = \mathbf{FnId} \to (\mathbf{TwoLValue}^n * \mathbf{Grammar}^n) \to \mathbf{Grammar}$
$\theta \in \mathbf{Env1} = \mathbf{Var} \to \mathbf{Grammar}$

Auxiliary Functions

stat	:	$(\mathbf{BTType} * \mathbf{TwoLValue}) \to \mathbf{Value}$
dyn	:	$(\mathbf{BTType} * \mathbf{TwoLValue}) \to \mathbf{TwoLValue}$
stat_G	:	$(\mathbf{BTType} * \mathbf{Grammar}) \to \mathbf{Value}$
stat+var	:	$(\mathbf{BTType} * \mathbf{Grammar}) \to (\mathbf{TwoLValue} * [\mathbf{Symbol}])$
gen-constr	:	$(\mathbf{Cname} * \mathbf{Value}) \to \mathbf{Cname}$
gen-fnid	:	$(\mathbf{FnId} * \mathbf{Value}) \to \mathbf{FnId}$
gen-fundef	:	$(\mathbf{FnId} * [\mathbf{Symbol}] * \mathbf{Expr}) \to \mathbf{Function}$
gen-const	:	$\mathbf{Value} \to \mathbf{Const}$
gen-primap	:	$(\mathbf{Prim} * \mathbf{Expr}^n) \to \mathbf{Expr}$
gen-constrap	:	$(\mathbf{Cname} * \mathbf{Expr}^n) \to \mathbf{Expr}$
gen-case	:	$(\mathbf{Expr} * \mathbf{Branch}^n) \to \mathbf{Expr}$
gen-funap	:	$(\mathbf{FnId} * \mathbf{Expr}^n) \to \mathbf{Expr}$
gen-branch	:	$(\mathbf{Cname} * \mathbf{Symbol}^n * \mathbf{Expr}) \to \mathbf{Branch}$

Valuation Functions

$\mathcal{G}_{\text{Prog}}$	:	$\mathbf{TwoLProg} \to \mathbf{MFGraph} \to \mathbf{Prog}$
$\mathcal{G}$	:	$\mathbf{TwoLExpr} \to \mathbf{FnEnv} \to \mathbf{Env1} \to \mathbf{Env} \to \mathbf{TwoLValue}$
$\mathcal{S}^{\#}_{mfg}$	:	$\mathbf{TwoLExpr} \to \mathbf{Env1} \to \mathbf{MFGraph} \to \mathbf{Grammar}$

$$\mathcal{G}_{\text{Prog}}[\![\{f_1(x_1 : \beta_{11}, \ldots, x_n : \beta_{1n}) = e_1 : \beta_1 \ldots f_m(x_1 : \beta_{m1}, \ldots, x_n : \beta_{mn}) = e_m : \beta_m\}]\!]\mathcal{M} =$$
$$\{\text{gen-fundef}(\text{fid}, \text{vars}_i, \Phi(\underline{f_i})(v_1, \ldots, v_n)(G_1, \ldots, G_n)) \mid \forall \underline{f_i} \in \text{Dom}(\mathcal{M}), \forall (G_1, \ldots, G_n) \in \text{Dom}(\mathcal{M}(\underline{f_i}))\}$$
$$\text{where } \Phi = \text{fix}\lambda\phi.\{\ f_1 \mapsto \lambda((v_1, \ldots, v_n), (G_1, \ldots, G_n)).$$
$$\mathcal{G}[\![e_1 : \beta_1]\!]\phi\{x_1 \mapsto G_1, \ldots, x_n \mapsto G_n\}\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, \ldots,$$
$$f_m \mapsto \lambda((v_1, \ldots, v_n), (G_1, \ldots, G_n)).$$
$$\mathcal{G}[\![e_m : \beta_m]\!]\phi\{x_1 \mapsto G_1, \ldots, x_n \mapsto G_n\}\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}\}$$
$$\text{statpart}_i = \text{stat}_G((\beta_{i1}, \ldots, \beta_{in}), (G_1, \ldots, G_n))$$
$$((v_1, \ldots, v_n), \text{vars}_i) = \text{stat+var}((\beta_{i1}, \ldots, \beta_{in}), (G_1, \ldots, G_n))$$
$$\text{fid} = \text{gen-fnid}(f_i, \text{statpart}_i)$$

$$\mathcal{G}[\![c : \beta]\!]\phi\theta\rho = [\![c]\!]$$
$$\mathcal{G}[\![\mathit{lift}\ e : \underline{\beta}]\!]\phi\theta\rho = \text{gen-const}(\mathcal{G}[\![e : \beta]\!]\phi\theta\rho)$$
$$\mathcal{G}[\![x : \beta]\!]\phi\theta\rho = \rho(x)$$
$$\mathcal{G}[\![\overline{p}(e_1 : \beta_1, \ldots, e_n : \beta_n) : \beta]\!]\phi\theta\rho = [\![p]\!](\mathcal{G}[\![e_1 : \beta_1]\!]\phi\theta\rho, \ldots, \mathcal{G}[\![e_n : \beta_n]\!]\phi\theta\rho)$$
$$\mathcal{G}[\![\underline{p}(e_1 : \beta_1, \ldots, e_n : \beta_n) : \beta]\!]\phi\theta\rho = \text{gen-primap}(p, \mathcal{G}[\![e_1 : \beta_1]\!]\phi\theta\rho, \ldots, \mathcal{G}[\![e_n : \beta_n]\!]\phi\theta\rho)$$
$$\mathcal{G}[\![\overline{c_i}(e_1 : \beta_1, \ldots, e_n : \beta_n) : \beta]\!]\phi\theta\rho = [\![c_i]\!](\mathcal{G}[\![e_1 : \beta_1]\!]\phi\theta\rho, \ldots, \mathcal{G}[\![e_n : \beta_n]\!]\phi\theta\rho)$$

Figure 11: Code Generator Part I

The interpreter from figure 2 is first subjected to the BTA from figure 8 with a completely static description for the program and the example bt-property from section 4. The most interesting part of the annotated version of the interpreter can be found in figure 13[10]. Notice, that both the equality test and the if expression are overlined, static operations, and can be executed at specialisation-time. The subdomain property for `val` allows re-typing the case-expression. For each of the grammar elements a branch will be generated. After the introduction of type definitions we obtain the residual program from figure 6. Contrast this program with the residual program generated by Mogensen's specialiser in figure 14. Its type definition is overly general, as it allows natural numbers, integer lists and other artifacts – objects that do not correspond to elements in the original type definition. Notice also that the case expression contains three dead branches. This stems from the fact that Mogensen uses one single subdomain property for all program expressions of type `univ` – hence a very general property.

In the case, where all specialised expressions of the orig-

[10] int_branches is the most interesting part because it handles the case-expression

$$
\begin{array}{l}
\mathcal{G}[\![\underline{c_i}(e_1:\beta_1,\ldots,e_n:\beta_n):\beta]\!]\phi\theta\rho = \\
\quad \text{let } w_1 = \mathcal{G}[\![e_1:\beta_1]\!]\phi\theta\rho \\
\qquad \vdots \\
\qquad w_n = \mathcal{G}[\![e_n:\beta_n]\!]\phi\theta\rho \\
\qquad \text{cid} = \text{gen-constr}(c_i, \text{stat}((\beta_1,\ldots,\beta_n),(w_1,\ldots,w_n))) \\
\quad \text{in gen-constrap}(\text{cid}, \text{dyn}((\beta_1,\ldots,\beta_n),(w_1,\ldots,w_n))) \\
\mathcal{G}[\![\overline{\text{case}}\ e_0:\beta_0 \text{ of } c_1(x_1,\ldots,x_n)\Rightarrow e_1:\beta;\ldots;c_m(x_1,\ldots,x_n)\Rightarrow e_m:\beta):\beta]\!]\phi\theta\rho = \\
\quad \text{let } c_i(v_1,\ldots,v_n) = \mathcal{G}[\![e_0:\beta_0]\!]\phi\theta\rho \\
\qquad [G ::= c_i(G_1,\ldots,G_n)] = \mathcal{S}^{\#}_{mfg}[\![e_0:\beta_0]\!]\theta\mathcal{M} \\
\quad \text{in } \mathcal{G}[\![e_i:\beta]\!]\phi([x_1\mapsto G_1,\ldots,x_n\mapsto G_n]\theta)([x_1\mapsto v_1,\ldots,x_n\mapsto v_n]\rho) \\
\mathcal{G}[\![(\underline{\text{case}}\ e_0:\beta_0 \text{ of } c_1(x_1,\ldots,x_n)\Rightarrow e_1:\beta;\ldots;c_m(x_1,\ldots,x_n)\Rightarrow e_m:\beta):\beta]\!]\phi\theta\rho = \\
\quad \text{let } v = \mathcal{G}[\![e_0:\beta_0]\!]\phi\theta\rho \\
\qquad [G ::= c_1(G_{111},\ldots,G_{11n}) + \ldots + c_1(G_{1r_11},\ldots,G_{1r_1n}) + \ldots + \\
\qquad\qquad c_m(G_{m11},\ldots,G_{m1n}) + \ldots + c_m(G_{mr_m1},\ldots,G_{mr_mn})] = \mathcal{S}^{\#}_{mfg}[\![e_0:\beta_0]\!]\theta\mathcal{M} \\
\qquad \left\{\begin{array}{l}
(c_1(p_{111},\ldots,p_{11n}),\text{vars}_{11}) = \text{stat+var}(\beta_0, c_1(G_{111},\ldots,G_{11n})) \\
c_1v_{11} = \text{stat}_G(\beta_0, c_1(G_{111},\ldots,G_{11n})) \\
e_{11} = \mathcal{G}[\![e_1:\beta]\!]\phi([x_1\mapsto G_{111},\ldots,x_n\mapsto G_{11n}]\theta)([x_1\mapsto p_{111},\ldots,x_n\mapsto p_{11n}]\rho) \\
c_{11} = \text{gen-constr}(c_1, v_{11}) \\
br_{11} = \text{gen-branch}(c_{11}, \text{vars}_{11}, e_{11})
\end{array}\right. \\
\qquad \vdots \\
\qquad \left\{\begin{array}{l}
(c_m(p_{mr_m1},\ldots,p_{mr_mn}),\text{vars}_{mr_m}) = \text{stat+var}(\beta_0, c_m(G_{mr_m1},\ldots,G_{mr_mn})) \\
c_mv_{mr_m} = \text{stat}_G(\beta_0, c_m(G_{mr_m1},\ldots,G_{mr_mn})) \\
e_{mr_m} = \mathcal{G}[\![e_m:\beta]\!]\phi([x_1\mapsto G_{mr_m1},\ldots,x_n\mapsto G_{mr_mn}]\theta)([x_1\mapsto p_{mr_m1},\ldots,x_n\mapsto p_{mr_mn}]\rho) \\
c_{mr_m} = \text{gen-constr}(c_m, v_{mr_m}) \\
br_{mr_m} = \text{gen-branch}(c_{mr_m}, \text{vars}_{mr_m}, e_{mr_m})
\end{array}\right. \\
\quad \text{in gen-case}(v, br_{11},\ldots,br_{mr_m}) \\
\mathcal{G}[\![\overline{f_i}(e_1:\beta_1,\ldots,e_n:\beta_n):\beta]\!]\phi\theta\rho = \text{let } G_1 = \mathcal{S}^{\#}_{mfg}[\![e_1:\beta_1]\!]\theta\mathcal{M} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad w_1 = \mathcal{G}[\![e_1:\beta_1]\!]\phi\theta\rho \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad G_n = \mathcal{S}^{\#}_{mfg}[\![e_n:\beta_n]\!]\theta\mathcal{M} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad w_n = \mathcal{G}[\![e_n:\beta_n]\!]\phi\theta\rho \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in } \phi(f_i)(w_1,\ldots,w_n)(G_1,\ldots,G_n) \\
\mathcal{G}[\![\underline{f_i}(e_1:\beta_1,\ldots,e_n:\beta_n):\beta]\!]\phi\theta\rho = \\
\quad \text{let } w_1 = \mathcal{G}[\![e_1:\beta_1]\!]\phi\theta\rho \\
\qquad \vdots \\
\qquad w_n = \mathcal{G}[\![e_n:\beta_n]\!]\phi\theta\rho \\
\qquad \text{fid} = \text{gen-fnid}(f_i, \text{stat}((\beta_1,\ldots,\beta_n),(w_1,\ldots,w_n))) \\
\quad \text{in gen-funap}(\text{fid}, \text{dyn}((\beta_1,\ldots,\beta_n),(w_1,\ldots,w_n)))
\end{array}
$$

Figure 12: Code Generator Part II

inal type T should have the same residual type, as is the case with the parser example from [Mog93], Mogensen's specialiser performs equally well.

7 Conclusion

With this work we have addressed some of the weak points of Mogensen's specialiser in [Mog93]. First of all we enhanced the BTA, by allowing constructors to have more than one single property for the whole program. Section 5 describes a new constructor specialiser. This specialiser is factored over different phases. Furthermore, it addresses an important issue of Mogensen's specialiser: the lack of precision that resulted in the generation of dead code or, even worse, in specialisation-time errors. And last but not least we have shown that constructor specialisation provides a general solution to the projection-injection problem in [Lau91b], a solution which is also useful in the context of a handwritten compiler generator. More philosophically, constructor specialisation shows that type information is simply too important information to neglect in the program transformation business. Types deserve first class status!

```
int_branches(val,brs,env,funs) =
  case brs of
   [] ⇒ Error;
   (b:brs) ⇒ let (constr1,vars,e) = b
             in case val of
                (Constr constr2 vals) ⇒
                   if constr1 ≡≡ constr2 then
                     int_expr(e,extend_env(vars,vals,env),funs)
                   else int_branches(val,brs,env,funs)
```

Figure 13: Excerpts of the annotated meta-circular interpreter

```
data spec_univ = Constr_Nil | Constr_Cons int spec_univ |
                 Constr_Zero | Constr_Succ spec_univ |
                 Constr_Error

int_length(inp) =
  case inp of
   (Constr_Nil) ⇒
      Constr_Zero
   (Constr_Cons i val) ⇒
      Constr_Succ (int_length(val))
   (Constr_Zero) ⇒ Constr_Error
   (Constr_Succ) ⇒ Constr_Error
   (Constr_Error) ⇒ Constr_Error
```

Figure 14: Program obtained by Mogensen's specialiser

8 Further Work

We are currently extending constructor specialisation to the higher-order case. The BTA is a simple generalisation of the one found in the paper. The abstract specialiser is currently set up as a non-standard type system requiring full conjunction to model the polyvariant specialisation. It is worrisome that higher-order control forces one to use a control-flow analyser. As for now, we are investigating whether other techniques apply. Another topic which we will be investigating in the near future is function polyvariance as part of the BTA.

9 Acknowledgements

Special thanks go to Torben Mogensen for his invaluable comments throughout the project. We also wish to thank Fritz Henglein for commenting on prior versions of the BTA. And last but not least we want to thank all the anonymous PEPM-referees for going through the paper in great details and for providing us with useful comments.

References

[CK91] C. Consel and S.C. Khoo. Parameterized partial evaluation. In *SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991, Toronto, Canada (Sigplan Notices, vol. 26, no. 6, June 1991)*, pages 92–106. New York: ACM, 1991.

[DNBDV91] A. De Niel, E. Bevers, and K. De Vlaminck. Partial evaluation of polymorphically typed functional languages: The representation problem. In M. Billaud et al., editors, *Analyse Statique en Programmation Équationnelle, Fonctionnelle, et Logique, Bordeaux, France, Octobre 1991 (Bigre, vol. 74)*, pages 90–97. Rennes: IRISA, 1991.

[Har77] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation.* PhD thesis, Linköping University, Sweden, 1977. Linköping Studies in Science and Technology Dissertations 14.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Berlin: Springer-Verlag, 1991.

[HL91] C.K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

[JM86] Neil D Jones and Alan Mycroft. Data Flow Analysis of Applicative Programs using Minimal Function Graphs. In *13th POPL, St. Petersburg, Florida*, pages 296–306, Jan. 1986.

[JSS89] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[Lau91a] J. Launchbury. *Projection Factorisations in Partial Evaluation.* Cambridge: Cambridge University Press, 1991.

[Lau91b] J. Launchbury. A strongly-typed self-applicable partial evaluator. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 145–164. ACM, Berlin: Springer-Verlag, 1991.

[Mog89] T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *TAPSOFT '89. Proc. Int. Conf. Theory and Practice of Software Development, Barcelona, Spain, March 1989 (Lecture Notes in Computer Science, vol. 352)*, pages 298–312. Berlin: Springer-Verlag, 1989.

[Mog93] T. Mogensen. Constructor specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 22–32. New York: ACM, 1993.

[MR85] P. Mishra and U. Reddy. Declaration-free type checking. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 7–21. ACM, Jan. 1985.

[Nie88] F. Nielson. A formal type system for comparing partial evaluators. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 349–384. Amsterdam: North-Holland, 1988.

[RW91] E. Ruf and D. Weise. Using types to avoid redundant specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 321–333. New York: ACM, 1991.