Principal Type Specialisation

Pablo E. Martínez López*
LIFIA, UNLP
CC.11 Correo Central
(1900) La Plata, Bs.As. Argentina
fidel@info.unlp.edu.ar

John Hughes
Chalmers Tekniska Högskola
Institutionen för Datavetenskap
412 96 Göteborg Sweden
rjmh@cs.chalmers.se

ABSTRACT

Type specialisation is an approach to program specialisation that works with *both* a program and its type to produce specialised versions of each. As it combines many powerful features, it appears to be a good framework for automatic program production, – despite the fact that it was designed originally to express optimal specialisation for interpreters written in typed languages.

The original specification of type specialisation used a system of rules expressing it as a generalised type system, rather than the usual view of specialisation as generalised evaluation. That system, while powerful, has some weaknesses not widely recognized – the most important being the inability to express *principal type specialisations* (a principal specialisation is one that is "more general" than any other for a given specialisable term, and from which those can be obtained by a suitable notion of instantiation). This inability is a problem when extending type specialisation to deal with polymorphism or modules.

This work presents a different formulation of the system specifying type specialisation *capturing the notion of principal specialisation for a language with static constructions and polyvariance.* It is a step forward in the study of type specialisation for polymorphic languages and lazy languages, and also permits modularity of specialisation, and better implementations.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming — *Program Transformation*; D.1.1 [**Programming Techniques**]: Functional Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism*

General Terms

Languages,Algorithms

Keywords

Program Specialisation, Type Specialisation, Type Based Transformation, Qualified Types

1. INTRODUCTION

Program specialisation is a way to generate programs automatically: a given (*source*) program is used to produce one or more versions of it (the *residual* programs), each specialised to particular data. The classic example is the recursive `power` function, which, given, for example, that the exponent is known to be 3, can be specialised to the non-recursive residual version `\x->x*(x*x)`, and similarly for other exponents.

There are different approaches to program specialisation, *Partial Evaluation* [17] being the most popular, well-known and well-engineered. Partial evaluation specialises programs by a form of generalized evaluation: subexpressions which depend on known data are replaced by the result of computing them. It is important to remark that we use the term Partial Evaluation referring to *how the specialisation works*, not any particular partial evaluator.

A good way to determine the limits of a specialisation method is to specialise an interpreter with it and compare the residual programs with the source one: if they are essentially the same, we can be confident that no limits exist – we say that the specialisation is *optimal*. Traditional partial evaluation can achieve optimality in the case of interpreters written in untyped – or dynamically typed – languages. But for typed interpreters, the residual code will contain type information coming from the representation of programs in the interpreter, and thus optimality is lost. This problem was stated by Neil Jones in 1987 as one of the open problems in the partial evaluation field [16].

Type Specialisation [10] was introduced by John Hughes in 1996 as a solution for optimal specialisation of typed interpreters. The basic idea behind type specialisation is to move static information to the type, thus specialising *both* the source term and source type to residual term and residual type. Types can be regarded as a static property of code, approximating known facts about it; for example in a functional language, when some expression has type *Int*, we

*The research reported in this paper was carried out at the Department of Computer Science of the Chalmers University, and it was partially funded by grants received by this author, who is a PhD student of University of Buenos Aires, from the ALFA-CORDIAL project Nr. ALR/B73011/94.04-5.0348.9, and the FOMEC project of Computer Science Department, FCEyN, University of Buenos Aires.

ASIA-PEPM'02, September 12-14, 2002, Aizu, Japan.

statically know that if its evaluation produces some value, it will be an integer. But if the expression is known to be the constant 42, for example, then a better approximation can be obtained by having a *type representing the property* of being the integer 42 – let's call this type $\hat{42}$, and allow the residual type system to have types like this one (and call them *one-point* types). Having all the information in the type, there is no need to execute the program anymore, and thus, we can replace the integer constant by a dummy constant having type $\hat{42}$ – that is, the source expression $42 : Int$ can be specialised to $\bullet : \hat{42}$, where $\bullet$ is the dummy constant. This specialisation of types into ones expressing more detailed facts about expressions – the type *Int* specialised to the type $\hat{42}$ in this example – needs a more powerful residual type system, which is the key fact allowing optimal specialisation for typed interpreters. Type specialisation is specified by a system of rules similar to those used to specify type systems – indeed, type specialisation can be seen as a generalised form of type inference, although up to now, only monomorphic. Thus, the specification of the specialisation procedure is modular: the specialisation of new constructs in the source language can be specified by the addition of new rules, without changing the rest of them.

Although type specialisation resembles partial evaluation in many aspects – it is also a technique for program specialisation, after all –, it is a different approach to program specialisation. This is something deserving attention, as type specialisation features are usually misunderstood for similar features of partial evaluation. The most common misunderstanding is that of annotations. In order to perform specialisation, both techniques use annotated programs. However, while in partial evaluation annotations can be automatically calculated, in type specialisation they are an important part of the input. For example, when specialising an interpreter for a given object language L, compilation for both untyped and typed versions of L can be obtained just varying the annotations. By choosing the annotations the programmer *decides* the static semantics of the language L – with the consequence that the annotations cannot be automatically calculated. This is possible because type checking is an integral part of type specialisation, and so it is embedded in the process of generating the residual program.

It is our goal to extend the power of type specialisation to treat other features of modern languages, such as Hindley-Milner polymorphism, type classes and lazy evaluation – to be able, for example, to type specialise Haskell programs. This work is a first step in the direction of generating and specialising polymorphic and lazy programs, as we illustrate with the examples in the appendix.

The original formulation of type specialisation [10] has some problems. One of them is that there are rules that are not completely syntax directed and so, for some source terms, several different *unrelated* specialisations can be produced. Another problem is related with the treatment of polyvariance – the ability of an expression to produce more than one residual in the same specialisation – and its interaction with dynamic recursion. In type specialisation polyvariance is first class (*any* expression may be polyvariant) and not the default (i.e. polyvariant expressions have to be explicitly annotated as such). This is essential to the expressiveness: for example, the firstifyier interpreter in [10] needs to pass polyvariant functions as arguments. But the way information flows when combining polyvariance with recursion is difficult to predict; the algorithm presented in [10] uses backtracking to calculate the solution.

These problems have important consequences. Firstly, the extension to produce polymorphic residual code or specialising polymorphic source code is very difficult to treat. Hindley-Milner polymorphism can be obtained as a variation of polyvariance (see the Ex. A.1), but for that to be possible an optimal treatment of the latter is mandatory. Secondly, an algorithm for specialisation will fail for terms with more than one specialisation when the context does not indicate which one to choose (or even worse, it may choose one of them arbitrarily!). Thirdly, even when the context provides enough information for the choice, it is too restrictive for the *whole* process of specialising a term to depend on the *whole* context; with this kind of restriction, specialisation of program modules is very hard to achieve, because in order to specialise a module, the specialiser may need to know *all* the program – in general, information about other modules on which the current one has no syntactic dependencies. Finally, due to backtracking, the algorithm may loop on correct inputs.

The main contribution of this paper is the presentation of a different formulation of type specialisation, and the proof of *existence of principal specialisations* for it – that is, specialisations that are more general than any other a given specialisable source term can have, and from which all those can be obtained by a suitable notion of instantiation. We separate the process of specialisation into two phases, one that is syntax directed and the other one based on constraint solving. Our treatment of polyvariance is essentially different from that in the original formulation; we use the notion of *conversion* from [13], but introducing it into the language in order to represent the coercions needed to specialise polyvariant expressions. The resulting system is more expressive than the original one. *This additional expressiveness and the improved treatment of polyvariance are the key to principality*, which is a necessary step in the specialisation of polymorphism – currently our system generates some polymorphic terms, but with a restricted form of polymorphism (see Ex. 3.2 and also Ex. A.1).

The translation of a type specialisation problem into one of constraint solving is *essential* for the improvement and also contributes to understand the interaction between polyvariance and recursion. Although it may seem at first glance that we are shifting the problem, constraint solving provides us with a better understanding of the information flow, allows the introduction of different heuristics that permit variations on the residual code, and permits a precise formulation of the problems posed by recursion. For example, the addition of polymorphism or lazy evaluation for static code can be obtained as variations of the algorithm for solving constraints – see Exs. A.1 and A.3. The contributions of this paper are important because they enable the possibility to add polymorphism in both the source and residual languages, the possibility to consider different evaluation strategies during constraint solving, and the possibility of modular specialisation: each module can be specialised in a principal manner, and the right instantiation produced when linking the residual code to the residual main program.

In the Sect. 2, we start by introducing the idea of type specialisation, we present several simple examples, and we explain some problems of the original formulation – we consider easier to explain the idea of type specialisation and the

problems we want to tackle using the original formulation, because it is more intuitive, and its problems are not widely recognized. The new specification proposed is presented in Sect. 3, and the problematic examples of the previous section are revisited. Section 4 explains constraint solving and post-processing phases used to obtain the final residual code. Extensions to the source language and the different problems that each of them introduce in the specialisation and post-processing phases are discussed in Sect. 5. We discuss related works on Sect. 6 and conclude the paper in Sect. 7. App. A contains a couple of more complex examples showing the possibilities enabled by our approach.

2. TYPE SPECIALISATION IN ITS ORIGINAL FORM

The key question behind type specialisation is "How can we improve the static information provided by the type of an expression?" An obvious first step is to have a more powerful type system. But, we also want to *remove* the static information expressed by this new type from the code, in order to obtain a simpler and (hopefully) more efficient code. So, we will work with two different typed languages: the source language, in which we code the programs to be specialised, and the residual language, which may contain additional constructs to express specialisation features.

Starting with the source language, an obvious question is how to know which information we want to move into the type. In the best scenario, some process will analyze our program, and mark those parts containing static information – a *binding time analyser* (or BTA). But, as shown by Hughes in [10], no completely automatic binding time analysis can be performed for type specialisation, because the only difference between an interpreter for a statically typed language and an interpreter for a dynamically typed one is their binding times: by annotating an interpreter, we *decide* the static semantics of the object language – and we cannot expect a program to do that for us, can we? See Ex. A.1 for a standard interpreter annotated in a non-standard way.

So we will assume that the binding time analysis has already been performed (at present, by hand; semiautomatic binding time analysis is under study), and define the source language as a two level language [7]: expressions with information we want to move from the code into the type will be marked *static* (with a $(\_)^S$ superscript), and those we want to keep in the residual code will be marked *dynamic* (with a $(\_)^D$ superscript). Observe that, as we have mentioned in the introduction, annotations are part of the input (i.e. part of the specification of the problem) – when annotating a term the programmer specifies, among other things, *what specialisations are to be allowed*. It is worthwhile to repeat one of the consequences of this: by annotating an interpreter to a language, the programmer *decides* its static semantics.

Source types will also reflect the static or dynamic nature of expressions – the type of constants, functions and operators will be consistent with the types of arguments. For example, the constant 42^D has type Int^D and the constant 42^S has type Int^S, and a dynamic function can only be dynamically applied, that is, both ^{D}'s in the following expression correspond to each other: $(\lambda^D x.x)\ @^D\ y$. Additionally, we need some extra features, such as casting a static computation into a dynamic one, or allowing a single expression to produce several different residual expressions in the residual code; these features can be obtained by extra annotations (**lift**, **poly**, and **spec**), whose effect is explained in the examples. We can define the source language as:

$$\begin{aligned} e ::= & x \mid n^D \mid e +^D e \mid n^S \mid e +^S e \mid \mathbf{lift}\ e \mid \\ & \lambda^D x.e \mid e\ @^D\ e \mid \mathbf{poly}\ e \mid \mathbf{spec}\ e \\ \tau ::= & Int^D \mid Int^S \mid \tau \to^D \tau \mid \mathbf{poly}\ \tau \end{aligned}$$

This language is a small subset of the language of the type specialiser from [10], but contains enough constructs to illustrate the problem considered here, and can be expanded to the full language (see Sect. 5).

Source type inference is straightforward, but with a major difference from partial evaluation techniques: as we have said, no restriction is imposed on annotations, and thus constraints that a BTA uses to infer a "best" type do not exist – see Ex. 6.1 for a program annotated in a way that is invalid for partial evaluation. It is important to repeat again that annotating a program involves taking decisions about the meaning of what we are specialising; so, annotations need to be very flexible.

The residual language has constructs and types corresponding to all the dynamic constructs and types in the source language, plus additional ones used to express the result of specialising static constructs. In the original formulation, these additional constructs are, in the term language, the dummy constant ($\bullet$), tuples and projections used for polyvariance, and, in the type language, singleton types (or *one-point* types, e.g. $\hat{4}$) and tuple types.

$$\begin{aligned} e' ::= & x' \mid n \mid e' + e' \mid \bullet \mid \\ & \lambda x'.e' \mid e'@e' \mid (e'_1, \ldots, e'_n) \mid \pi_i e' \\ \tau' ::= & Int \mid \hat{n} \mid \tau' \to \tau' \mid (\tau'_1, \ldots, \tau'_n) \end{aligned}$$

In order to express the result of the specialisation procedure, Hughes introduced a new kind of judgment, and a system of rules to derive valid judgements. We write $\Gamma \vdash e : \tau \hookrightarrow e' : \tau'$ to express the fact that, under the assumptions in Γ, source expression e of source type τ specialises to residual expression e' and residual type τ'. For example, $\vdash 42^D : Int^D \hookrightarrow 42 : Int$, and $\vdash 42^S : Int^S \hookrightarrow \bullet : \hat{42}$ are derivable judgments corresponding to the examples discussed in the introduction.

Example 2.1 Observe how every expression annotated as dynamic appears in the residual term (in fact, we have that a fully dynamic expression, that is one in which every annotation is ^{D}, specialises to a copy of itself with the annotations removed).

1. $\vdash (2^D +^D 1^D) +^D 1^D : Int^D \hookrightarrow (2+1)+1 : Int$
2. $\vdash (2^S +^S 1^S) +^S 1^S : Int^S \hookrightarrow \bullet : \hat{4}$
3. $\vdash \mathbf{lift}\ (2^S +^S 1^S) +^D 1^D : Int^D \hookrightarrow 3+1 : Int$

Also observe in 3 how the use of **lift** coerces a static integer into a dynamic one, thus inserting the result of the static computation into the residual term.

The rules (O-INT^D), (O-+^{D}), (O-INT^S), (O-+^{S}), and (O-LIFT)[1] used to derive the judgments on Ex. 2.1 appear on Fig. 1. It is important to note that both (O-+^{S}) and (O-LIFT) force the choice of a suitable one-point type for the subexpressions, n

[1]The O- prefix distinguishes the Original rules from the ones presented later in this paper.

$$\text{(O-VAR)}\quad \Gamma, x:\tau \hookrightarrow e':\tau', \Gamma' \vdash x:\tau \hookrightarrow e':\tau'$$

$$\text{(O-INT}^D\text{)}\quad \Gamma \vdash n^D : Int^D \hookrightarrow n : Int$$

$$\text{(O-+}^D\text{)}\quad \frac{\Gamma \vdash e_i : Int^D \hookrightarrow e_i' : Int}{\Gamma \vdash e_1 +^S e_2 : Int^D \hookrightarrow e_1' + e_2' : Int}$$

$$\text{(O-INT}^S\text{)}\quad \Gamma \vdash n^S : Int^S \hookrightarrow \bullet : \hat{n}$$

$$\text{(O-+}^S\text{)}\quad \frac{\Gamma \vdash e_i : Int^S \hookrightarrow e_i' : \hat{n_i}}{\Gamma \vdash e_1 +^S e_2 : Int^S \hookrightarrow \bullet : \hat{n}}\ (n_1+n_2=n)$$

$$\text{(O-LIFT)}\quad \frac{\Gamma \vdash e : Int^S \hookrightarrow e' : \hat{n}}{\Gamma \vdash \mathbf{lift}\ e : Int^D \hookrightarrow n : Int}$$

$$\text{(O-}\lambda^D\text{)}\quad \frac{\Gamma, x:\tau_2 \hookrightarrow x' : \tau_2' \vdash e : \tau_1 \hookrightarrow e' : \tau_1' \qquad x' \text{ fresh}}{\Gamma \vdash \lambda^D x.e : \tau_2 \to^D \tau_1 \hookrightarrow \lambda x'.e' : \tau_2' \to \tau_1'}$$

$$\text{(O-@}^D\text{)}\quad \frac{\Gamma \vdash e_1 : \tau_2 \to^D \tau_1 \hookrightarrow e_1' : \tau_2' \to \tau_1' \qquad \Gamma \vdash e_2 : \tau_2 \hookrightarrow e_2' : \tau_2'}{\Gamma \vdash e_1\ @^D\ e_2 : \tau_1 \hookrightarrow e_1'@e_2' : \tau_1'}$$

$$\text{(O-POLY)}\quad \frac{(\forall i)(\exists e'\tau')(\Gamma \vdash e : \tau \hookrightarrow e' : \tau' \quad \vec{\tau}'!i = \tau' \quad \vec{e}'!i = e')}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow \vec{e}' : \vec{\tau}'}$$

$$\text{(O-SPEC)}\quad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau \hookrightarrow e' : \vec{\tau}' \qquad \vec{\tau}'!i = \tau'}{\Gamma \vdash \mathbf{spec}\ e : \tau \hookrightarrow \pi_i e' : \tau'}$$

Figure 1: Original type specialisation rules.

in the former, and n_1, n_2 in the latter – this is not (always) syntax directed.

Example 2.2 Assumptions provide the information for the specialisation of free variables, which allows the specialisation of functions.

1. $x : Int^S \hookrightarrow \bullet : \hat{3} \vdash x +^S 1^S : Int^S \hookrightarrow \bullet : \hat{4}$
2. $\vdash (\lambda^D x.x +^S 1^S)\ @^D\ (2^S +^S 1^S) : Int^S \hookrightarrow (\lambda x'.\bullet)@\bullet : \hat{4}$
3. $\vdash (\lambda^D x.\mathbf{lift}\ x +^D 1^D)\ @^D\ (2^S +^S 1^S) : Int^D \hookrightarrow (\lambda x'.3+1)@\bullet : Int$

Observe in Ex. 2.2-2 and Ex. 2.2-3 that a dynamic function is allowed to have a static argument, because the unification of the residual types will provide the value needed for the computation; for example, in Ex. 2.2-2, the residual function $(\lambda x'.\bullet)$ has residual type $\hat{3} \to \hat{4}$ thus providing information about x' to the function body even when the function is not reduced. It is not possible for dynamic functions to have static arguments with the partial evaluation approach [7], as the only way to propagate information is by reduction, and a dynamic function is never reduced. Another example of these freedom of annotations is presented in Ex. 6.1. Our examples are too small to show the actual usefulness of this kind of annotations – we want them to remain simple –, but the absence of restrictions is the key to the new possibilities of type specialisation: the ability to have static parameters under dynamic constructs allows the necessary elimination of type tags to achieve optimal specialisation of typed interpreters. In the appendix we present a couple of bigger examples, one of them eliminating the tags from an interpreter for a typed lambda calculus with let-bound polymorphism.

In Fig. 1 there also appear rules (O-VAR), (O-λ^D), and (O-@^{D}) needed to derive judgments involving free variables and functions. Observe that in (O-@^{D}) there is the expected flow of information, because in a specialisation algorithm the antecedent of the residual function type should be unified with the residual type of the argument. Also observe that the residual type τ_2' assigned to the residual of the λ-bound variable in (O-λ^D) is not restricted, allowing *any* type to be chosen at this stage (this choice will be further restricted by application, as noted above).

One important feature of type specialisation is that there exist correctly annotated terms that cannot be specialised. Consider[2]

$$\mathbf{let}^D\ f = \lambda^D x.\mathbf{lift}\ x +^D 1^D \ \mathbf{in}\ (f\ @^D\ 42^S, f\ @^D\ 17^S)^D : (Int^D, Int^D)^D.$$

What should the specialisation for this term be? As we have seen in Ex. 2.2-3, the body of the function is specialised according to the parameter, but f has two *different* parameters! Thus, the type specialiser fails. This ability to fail is an important feature of type specialisation – when specialising a typed interpreter with an input that does not typecheck, it is this ability to fail which allows producing the error, representing a typing error by a specialisation one.

But what to do if we do not want an error in this case? The solution is to allow f to specialise in more than one way in the same program. *Polyvariance* is the ability of an expression to specialise to more than one residual expression. Type specialisation is not polyvariant by default – as can be noticed from this paper, the treatment of polyvariant expressions is much more complex than that of monovariant ones –, and it is first class, so its use is indicated in the source language by the annotations **poly** and **spec**, the former to produce a polyvariant expression, and the latter to choose the corresponding specialisation of it.

Example 2.3 Observe the use of **poly** in the definition of f (and how that annotation produces a tuple for the definition of f' in the residual code), and the use of **spec** in every application of f to an argument (and how that produces a corresponding projection in the residual code). Also notice that we *chose* to mark the addition as dynamic; the reason for this is to keep the details of its principal specialisation simple – see Exs. 3.3 and 3.4-1.

$$\begin{array}{l}\vdash \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D) \\ \quad \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 42^S, \mathbf{spec}\ f\ @^D\ 17^S)^D : (Int^D, Int^D)^D \\ \quad \hookrightarrow \mathbf{let}\ f' = (\lambda x'.42+1, \lambda x'.17+1) \\ \qquad \mathbf{in}\ (\pi_1 f'@\bullet, \pi_2 f'@\bullet) : (Int, Int)^D\end{array}$$

The size of the residual tuple is arbitrary, provided that it has at least two elements ($\lambda x'.42+1$ and $\lambda x'.17+1$), and the order is arbitrary as well, provided that the projections select the appropriate element. The rules (O-POLY) and (O-SPEC) used to specialise polyvariance reflect that (see Fig.1, where $\vec{\tau}'$ stands for a tuple of residual types, $\vec{\tau}'!i$ being its i-th component, and similarly with $\vec{e}'$). These rules are not syntax directed, because in order to decide the size and order of the residual tuple, every use of the polyvariant

[2] *let* constructs mean the usual abbreviation for a β-redex (in a monomorphic setting); tuples are easily added.

expression must be known, so, they depend on the context where the expression appears.

We remarked that when the context provides enough information, the specialisation can proceed with no problems (Exs. 2.2-2, 2.2-3, and 2.3). But, what happens when specialising any of the expressions in the following example?

Example 2.4 Observe that in all cases there is some static information missing.

1. $\lambda^D x.x +^S 1^S : Int^S \to^D Int^S$
2. $\mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D) : \mathbf{poly}\ (Int^S \to^D Int^D)$
3. $\lambda^D f.\mathbf{spec}\ f\ @^D\ 13^S : \mathbf{poly}\ (Int^S \to^D Int^D) \to^D Int^D$

All have many different *unrelated* specialisations! For example, the function in Ex. 2.4-1 has one specialisation for each possible value for x – in particular, $\lambda x'.\bullet : \hat{n} \to \hat{n}'$, for every value of n and n' such that $n' = n + 1$. If this function appears in a module, but it is applied in another one, then the specialisation should wait until the value n of the argument is known, in order to decide the residual type. The same problem appears in the case of polyvariance, as can be observed in Ex. 2.4-2 and Ex. 2.4-3: the generation of the tuple or the selection of the right projection should be deferred until all the information is available – in the second case, if the expression is annotated as polyvariant, the right projection can even be a different one on each application!

These problems make the design of algorithms to perform type specialisation a difficult task, introducing the need for failures or the need to choose arbitrarily when there is not enough contextual information, and also forcing the specialisation of programs as a whole. They make also difficult dealing with polymorphism. In the next section we present a different formulation of the system specifying type specialisation, designed to cope with this problem.

3. PRINCIPAL TYPE SPECIALISATION

The problem identified is very similar to that appearing in simply typed λ-calculus when typing an expression like $\lambda x.x$: the type of x is determined by the context of use, and different types for this expression have no relation between them expressible in the system. The solution to this problem for the Hindley-Milner type system is to extend the type language in order to allow polymorphism – by introducing type variables – modifying the typing rules accordingly [2], and defining a notion of instantiation for types. Then it can be proved that there exists a particular type for every term – the *principal* type – such that every other valid type for the same term can be obtained by instantiation of it.

Our contribution is to achieve a similar result for specialisations: the existence of a *principal type specialisation*, that is, a specialisation such that every other valid one for the same source term can be obtained by instantiation of it. A first step in this direction is to introduce residual type variables – here denoted by t. Unfortunately, this is not enough, as subtle dependencies between types (such as the relation between $\hat{n}$ and $\hat{n}'$ in the specialisation of Ex. 2.4-1), cannot be expressed. Using type variables, we expect a specialisation of the form $\vdash \lambda^D x.x +^S 1^S : Int^S \to^D Int^S \hookrightarrow \lambda x.\bullet : \forall t, t'.t \to t'$ but with an extra condition relating t and t'. The *theory of qualified types* presents a type framework that allows us to express conditions relating universally quantified variables [13]. In this framework, types are

$$
\begin{array}{lcl}
e' & ::= & x' \mid n \mid e' + e' \mid \bullet \mid \lambda x'.e' \mid e'@e' \mid \\
 & & h \mid v[e'] \mid \Lambda h.e' \mid e'(\!(v)\!) \\
v & ::= & h \mid n \mid C \mid v \circ v \\
C & ::= & [] \mid \Lambda h.C \mid C(\!(v)\!)
\end{array}
$$

$$
\begin{array}{lcl}
\tau' & ::= & t \mid Int \mid \hat{n} \mid \tau' \to \tau' \mid \mathbf{poly}\ \sigma \\
\rho & ::= & \delta \Rightarrow \rho \mid \tau' \\
\sigma & ::= & s \mid \forall s.\sigma \mid \forall t.\sigma \mid \rho \\
\delta & ::= & \mathrm{IsInt}\ \tau' \mid \tau' := \tau' + \tau' \mid \mathrm{IsMG}\ \sigma\ \sigma
\end{array}
$$

Figure 2: Syntax of residual terms and types.

enriched with predicates constraining variables, and type inference, the 'more general' ordering, instantiation, etc. from the Hindley-Milner system, have to be redefined to take the predicates into consideration.

In the example above, we can introduce a predicate expressing the relation between type variables, and thus produce $\vdash \lambda^D x.x +^S 1^S : Int^S \to^D Int^S \hookrightarrow \lambda x.\bullet : \forall t, t'.t' := t + \hat{1} \Rightarrow t \to t'$, in which the predicate $t' := t + \hat{1}$ is *qualifying* the type $t \to t'$ and thus restricting the quantification.

In the theory of qualified types, predicates appearing in types have corresponding information appearing in terms – called *evidence* in [13]. Abstraction and application of evidence are used to ask for or provide the information proving a given predicate, although the system can be presented more concisely without it. But our system depends vitally on evidence; consider the source term

$$e = \lambda^D x.\mathbf{lift}\ (x +^S 1^S) : Int^S \to^D Int^D$$

Its residual type will be $\forall t, t'.t' := t + \hat{1} \Rightarrow t \to Int$, but what should the residual term be? Associated with every predicate, we have evidence that the predicate can be proved, and when that evidence is not known, we use an *evidence variable* abstracted by an *evidence abstraction*. The residual language is changed in order to introduce evidence and predicates – its syntax is presented in Fig. 2 – and a notion of instantiation, $\geq$, is defined following [13]. We have added the syntactic category of evidence, denoted by v, and evidence variables (h), evidence abstraction ($\Lambda h.e'$), and evidence application ($e'(\!(v)\!)$). Type and scheme variables are introduced in the type language – denoted by t and s respectively – and also different kinds of predicates to express the specialisation of different kinds of missing static information; for each kind of predicate, a corresponding kind of evidence is defined. Quantification is treated, as in the Hindley-Milner polymorphic system, in two syntactic levels, with quantifiers appearing on type schemes (σ). The innovation is the use of the new type $\mathbf{poly}\ \sigma$ in order to express first-class polyvariance; this allows type schemes to appear in types – but in a way controlled by the annotations – thus supporting residual programs with a controlled kind of higher-order polymorphism. Scheme variables will be appropriately constrained by predicates of the form IsMG (for "is More General", because it internalizes the relation $\geq$); the evidence for these predicates are *conversions*, denoted by C, which are special constructs that coerce a term of a general type (scheme) into a more specific one by adapting the evidence – remember that types schemes also include constraints in the form of predicates, and that evidence abstraction is used on terms to wait for the corresponding evidence (see Ex. 3.1). An important difference between this work and [13] is that we use conversions as part of the resid-

$$\text{(Fst)}\quad h:\Delta, h':\Delta' \Vdash h:\Delta$$

$$\text{(Snd)}\quad h:\Delta, h':\Delta' \Vdash h':\Delta'$$

$$\text{(Univ)}\quad \frac{h:\Delta \Vdash v':\Delta' \qquad h:\Delta \Vdash v'':\Delta''}{h:\Delta \Vdash v':\Delta', v'':\Delta''}$$

$$\text{(Trans)}\quad \frac{h:\Delta \Vdash v':\Delta' \qquad h':\Delta' \Vdash v'':\Delta''}{h:\Delta \Vdash v''[v'/h']:\Delta''}$$

$$\text{(Close)}\quad \frac{h:\Delta \Vdash v':\Delta'}{h:S\,\Delta \Vdash v':S\,\Delta'}$$

Figure 3: Structural laws satisfied by entailment.

$$\text{(IsInt)}\quad \Delta \Vdash n : \text{IsInt}\ \hat{n}$$

$$\text{(IsOp)}\quad h:\Delta \Vdash n : \hat{n} := \hat{n}_1 + \hat{n}_2 \quad (\textit{whenever } n{=}n_1{+}n_2)$$

$$\text{(IsMG)}\quad \frac{C : (\Delta \mid \sigma') \geq (\Delta \mid \sigma)}{\Delta \Vdash C : \text{IsMG}\ \sigma'\ \sigma}$$

$$\text{(Comp)}\quad \frac{\Delta \Vdash v : \text{IsMG}\ \sigma_1\ \sigma_2 \qquad \Delta \Vdash v' : \text{IsMG}\ \sigma_2\ \sigma_3}{\Delta \Vdash v' \circ v : \text{IsMG}\ \sigma_1\ \sigma_3}$$

Figure 4: Entailment for evidence construction.

ual language (as evidence), while in that work they are used only in the metalanguage to achieve the technical results.

Returning to the example above, the expression e has residual type $\forall t, t'.t' := t{+}\hat{1} \Rightarrow t \to \mathit{Int}$, and as the evidence associated with the predicate is the number represented by the one-point type, the resulting residual term is $\Lambda h.\lambda x'.h$: the evidence variable h in the evidence abstraction waits for the number represented by t' to be known. If we apply e to $42^S : \mathit{Int}^S$, the residual function needs to be applied to $\bullet : \hat{42}$, but first the evidence proving that $\hat{43} := \hat{42}{+}\hat{1}$ should be provided using an evidence application – in this case that evidence is the number 43, resulting in $(\Lambda h.\lambda x'.h)(\!(43)\!)@\bullet$. This last term can be reduced to $(\lambda x'.43)@\bullet$ using a corresponding β-reduction for evidence $((\Lambda h.e')(\!(v)\!) \rhd_{\beta_v} e'[v/h])$. These reductions are performed by a postprocessor before the program is run; this postprocessing is included in the phase of *evidence elimination* (see Sect. 4).

Properties relating lists of predicates (denoted by Δ) and evidence are captured by a relation called *entailment* (denoted by $\Vdash$), which satisfies several structural properties as stated in [13] – see Fig. 3 –, plus some specific ones appearing in Fig. 4. The rules of entailment for IsMG ((IsMG) and (Comp)) capture its internalization of $\geq$– rule (Comp) captures the transitivity of $\geq$, which is important in the proofs.

The notion of instantiation for qualified types is more involved than the corresponding one for normal types: besides replacing quantified type variables by types, it should take into account predicates and evidence. Informally, the relation $\geq$ capturing the notion of instantiation is defined such that $C : (\Delta \mid \sigma_1) \geq (\Delta' \mid \sigma_2)$ when σ_2 is an instance of σ_1 (this means instantiating variables, and proving predicates – see Ex. 3.1, specially item 1), and if $e : \sigma_1$ assuming the predicates in Δ, then $C[e] : \sigma_2$ assuming the predicates in Δ'. Observe that C provides the term e with evidence for the predicates in σ_1. The exact definition is rather technical,

$$\text{(QIN)}\quad \frac{\Delta, h:\delta \mid \Gamma \vdash_{\mathrm{P}} e:\tau \hookrightarrow e':\rho}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e:\tau \hookrightarrow \Lambda h.e' : \delta \Rightarrow \rho}$$

$$\text{(QOUT)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e:\tau \hookrightarrow e' : \delta \Rightarrow \rho \qquad \Delta \Vdash v : \delta}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e:\tau \hookrightarrow e'(\!(v)\!) : \rho}$$

$$\text{(GEN)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e:\tau \hookrightarrow e' : \sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e:\tau \hookrightarrow e' : \forall\alpha.\sigma}\ (\alpha \notin FV(\Delta) \cup FV(\Gamma))$$

$$\text{(INST)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e:\tau \hookrightarrow e' : \forall\alpha.\sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e:\tau \hookrightarrow e' : S\,\sigma}\ (\mathrm{dom}(S)=\alpha)$$

Figure 5: Specialisation rules (part II).

and can be found in the technical report [19].

Example 3.1 Conversions are used to adjust the evidence demanded by different schemes. For all Δ it holds that

1. $[](\!(42)\!) : (\Delta \mid \forall t.\text{IsInt}\ t \Rightarrow t \to \mathit{Int}) \geq (\Delta \mid \hat{42} \to \mathit{Int})$
2. $C : (\Delta \mid \forall t_1, t_2.\text{IsInt}\ t_1, \text{IsInt}\ t_2 \Rightarrow t_1 \to t_2) \geq (\Delta \mid \forall t.\text{IsInt}\ t \Rightarrow t \to t)$ where $C = \Lambda h.[](\!(h)\!)(\!(h)\!)$

For example, the conversion $[](\!(42)\!)$ in Ex. 3.1-1 applies the evidence 42, thus transforming a term of type $(\forall t.\text{IsInt}\ t \Rightarrow t \to \mathit{Int})$ into one of type $(\hat{42} \to \mathit{Int})$ (one example of such a term can be found in Ex.3.2-1). Conversions have several interesting and useful properties, such as reflexivity, transitivity, etc. – see [13].

Judgments are extended with a new environment of predicates and produce residual type schemes, and thus they become $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$; the rules are changed accordingly – Figs. 6 and 5. Observe, for example, in rules (LIFT) and ($+^S$) how the relation $\Vdash$ is used on the premises, allowing residual types to be properly constrained type variables when corresponding static information is missing. The acute reader may have noticed that the assumptions in Γ were slightly modified: instead of allowing a variable to be specialised to any residual expression e', we only allow it to be specialised to a residual variable x'. The original form of assumptions are used to produce unfolding in the case of static lets and static lambdas; we use a different mechanism to obtain it (by extending the language of evidence with a form of application ($@_v$) reducible using a special β-rule : $(\lambda x'.e'_1)@_v e'_2 \rhd_{@_v} e'_1[x'/e'_2]$).

Example 3.2 The predicate IsInt constraints a type such that it can only be a one-point type, and $\_ := \_ + \_$ constraints three types such that they are one-point types, and the first is the result of adding the other two.

1. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x : \mathit{Int}^S \to^D \mathit{Int}^D$
   $\hookrightarrow \Lambda h.\lambda x'.h : \forall t.\text{IsInt}\ t \Rightarrow t \to \mathit{Int}$
2. $\vdash_{\mathrm{P}} \lambda^D x.x +^S 1^S : \mathit{Int}^S \to^D \mathit{Int}^S$
   $\hookrightarrow \Lambda h, h'.\lambda x'.\bullet : \forall t, t'.\text{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \to t'$
3. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x +^D 1^D : \mathit{Int}^S \to^D \mathit{Int}^D$
   $\hookrightarrow \Lambda h.\lambda x'.h + 1 : \forall t.\text{IsInt}\ t \Rightarrow t \to \mathit{Int}$
4. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x +^D \mathbf{lift}\ (x +^S 1^S) : \mathit{Int}^S \to^D \mathit{Int}^D$
   $\hookrightarrow \Lambda h, h'.\lambda x'.h{+}h' : \forall t, t'.\text{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \to \mathit{Int}$

$$\text{(VAR)}\quad \Delta \mid \Gamma, x:\tau \hookrightarrow x':\tau', \Gamma' \vdash_{\mathrm{P}} x:\tau \;\hookrightarrow\; x':\tau'$$

$$\text{(INT}^D\text{)}\quad \Delta \mid \Gamma \vdash_{\mathrm{P}} n^D : \mathit{Int}^D \;\hookrightarrow\; n : \mathit{Int}$$

$$\text{(+}^D\text{)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_i : \mathit{Int}^D \;\hookrightarrow\; e_i' : \mathit{Int}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 +^S e_2 : \mathit{Int}^D \;\hookrightarrow\; e_1' + e_2' : \mathit{Int}}$$

$$\text{(INT}^S\text{)}\quad \Delta \mid \Gamma \vdash_{\mathrm{P}} n^S : \mathit{Int}^S \;\hookrightarrow\; \bullet : \hat{n}$$

$$\text{(+}^S\text{)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_i : \mathit{Int}^S \;\hookrightarrow\; e_i' : \tau_i' \qquad \Delta \Vdash v : \tau' := \tau_1' + \tau_2'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 +^S e_2 : \mathit{Int}^S \;\hookrightarrow\; \bullet : \tau'}$$

$$\text{(LIFT)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \mathit{Int}^S \;\hookrightarrow\; e' : \tau' \qquad \Delta \Vdash v : \mathrm{IsInt}\ \tau'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{lift}\ e : \mathit{Int}^D \;\hookrightarrow\; v : \mathit{Int}}$$

$$\text{(}\lambda^D\text{)}\quad \frac{\Delta \mid \Gamma, x:\tau_2 \hookrightarrow x':\tau_2' \vdash_{\mathrm{P}} e:\tau_1 \;\hookrightarrow\; e':\tau_1' \qquad \Delta \vdash_{\mathrm{SR}} \tau_2 \;\hookrightarrow\; \tau_2' \quad x' \text{ fresh}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x.e : \tau_2 \to^D \tau_1 \;\hookrightarrow\; \lambda x'.e' : \tau_2' \to \tau_1'}$$

$$\text{(@}^D\text{)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 : \tau_2 \to^D \tau_1 \;\hookrightarrow\; e_1' : \tau_2' \to \tau_1' \qquad \Delta \mid \Gamma \vdash_{\mathrm{P}} e_2 : \tau_2 \;\hookrightarrow\; e_2' : \tau_2'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 \,@^D\, e_2 : \tau_1 \;\hookrightarrow\; e_1' @ e_2' : \tau_1'}$$

$$\text{(POLY)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e' : \sigma' \qquad \Delta \Vdash v : \mathrm{IsMG}\ \sigma'\ \sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \;\hookrightarrow\; v[e'] : \mathbf{poly}\ \sigma}$$

$$\text{(SPEC)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \mathbf{poly}\ \tau \;\hookrightarrow\; e' : \mathbf{poly}\ \sigma \qquad \Delta \Vdash v : \mathrm{IsMG}\ \sigma\ \tau' \qquad \Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \tau'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{spec}\ e : \tau \;\hookrightarrow\; v[e'] : \tau'}$$

Figure 6: Specialisation rules (part I).

The residual term $\Lambda h.\lambda x'.h$ in Ex. 3.2-1 can be converted into $\lambda x'.42$ of type $\hat{42} \to \mathit{Int}$ using the conversion in Ex. 3.1-1, and reducing the resulting redexes.

Observe how every predicate appearing in a residual type has a corresponding evidence abstraction on the term level. This is obtained by rules moving predicates from the context into the type, and vice-versa: (QIN) and (QOUT) in Fig. 5.

It should also be observed that the predicate $\_ := \_ + \_$ constrains many variables at once and creates some dependencies between them; for example, in $t' := t + \hat{1}$, the variable t' depends on t. In order to quantify residual types in the right way (that is, either all or none of the variables with dependencies are quantified; the type $(\forall t.\mathrm{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \to t')$ is excluded in Ex. 3.2-2, because as t' depends on t, it cannot be free when t is bounded), a notion of functional dependency should be used in the rules for generalization, exactly as it is done in [15].

We have said that the residual type assigned to the residual variable in the rule for lambda abstraction is not constrained in any way in the original formulation. In that work it was not a problem, because only whole programs were the target for specialisation, so it was expected that every function be applied at least once. But when looking for principal specialisations, this becomes a problem, since certain terms have more specialisations than expected (e.g. $\vdash \lambda^D x.x : \mathit{Int}^S \to^D \mathit{Int}^S \hookrightarrow \lambda x'.x' : \mathit{Bool} \to \mathit{Bool}$ is a valid specialisation), and *every* valid specialisation should be expressed by the principal one. So, we have added a source-residual relationship – τ' is a residual of τ –, expressed by a new kind of judgment: $\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \tau'$. Rules to derive

$$\text{(SR-SINT)}\quad \frac{\Delta \Vdash \mathrm{IsInt}\ \tau'}{\Delta \vdash_{\mathrm{SR}} \mathit{Int}^S \;\hookrightarrow\; \tau'}$$

$$\text{(SR-DINT)}\quad \Delta \vdash_{\mathrm{SR}} \mathit{Int}^D \;\hookrightarrow\; \mathit{Int}$$

$$\text{(SR-DFUN)}\quad \frac{\Delta \vdash_{\mathrm{SR}} \tau_1 \;\hookrightarrow\; \tau_1' \qquad \Delta \vdash_{\mathrm{SR}} \tau_2 \;\hookrightarrow\; \tau_2'}{\Delta \vdash_{\mathrm{SR}} \tau_2 \to^D \tau_1 \;\hookrightarrow\; \tau_2' \to \tau_1'}$$

$$\text{(SR-POLY)}\quad \frac{\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \sigma' \qquad \Delta \Vdash \mathrm{IsMG}\ \sigma'\ \sigma}{\Delta \vdash_{\mathrm{SR}} \mathbf{poly}\ \tau \;\hookrightarrow\; \mathbf{poly}\ \sigma}$$

$$\text{(SR-QIN)}\quad \frac{\Delta, \delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \rho}{\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \delta \Rightarrow \rho}$$

$$\text{(SR-QOUT)}\quad \frac{\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \delta \Rightarrow \rho \qquad \Delta \Vdash \delta}{\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \rho}$$

$$\text{(SR-GEN)}\quad \frac{\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \sigma}{\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \forall\alpha.\sigma}\ (\alpha \notin FV(\Delta))$$

$$\text{(SR-INST)}\quad \frac{\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \forall\alpha.\sigma}{\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; S\,\sigma}\ (\mathrm{dom}(S)=\alpha)$$

Figure 7: Source-residual relationship.

this judgment are more or less straightforward, (SR-SINT) for static integers and (SR-POLY) for polyvariance being the most interesting ones – see Fig. 7. In this way, we cure a simple omission in the original paper, which is necessary to achive our contribution. We use this new judgment in the rule for λ-abstraction, which becomes rule (${\lambda}^D$) (in Fig. 6). It is by combining rules (${\lambda}^D$) and (SR-SINT) that the predicate IsInt appears in all the specialisations of Ex. 3.2.

Type specialising polyvariance in a principal manner is more involved; its treatment is the key notion allowing principality and thus the main contribution of the paper. The basic idea is that the specialisation of a polyvariant expression e should have a scheme as its residual type (instead of a tuple type), and **spec**'s of e provide adequate instances (instead of projections); this is easier to see with an example.

Example 3.3 A specialisation of the expression in Ex. 2.3 using this idea is

$$\begin{aligned}&\vdash_{\mathrm{P}} \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D)\\ &\quad \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 42^S, \mathbf{spec}\ f\ @^D\ 17^S)^D : (\mathit{Int}^D, \mathit{Int}^D)^D\\ &\quad \hookrightarrow \mathbf{let}\ f' = \Lambda h.\lambda x'.h + 1\\ &\qquad \mathbf{in}\ (f'(\!(42)\!)@\bullet, f'(\!(17)\!)@\bullet) : (\mathit{Int}, \mathit{Int})^D\end{aligned}$$

Observe the use of an evidence abstraction corresponding to the use of **poly** and the use of evidence applications corresponding to the use of every **spec** (instead of the previous use of tuples and projections), so no decision about the size and order of the tuple is needed. Unfortunately, this is not enough to achieve principality, because a given source expression may have different residual schemes in different specialisations (e.g. $\lambda^D x.\lambda^D y.\mathbf{lift}\ x : \mathit{Int}^S \to^D \mathit{Int}^S \to^D \mathit{Int}^D$ specialises to $\Lambda h, h'.\lambda x'.\lambda y'.h : \forall t, t'.\mathrm{IsInt}\ t, \mathrm{IsInt}\ t' \Rightarrow t \to t' \to \mathit{Int}$ and also to $\Lambda h.\lambda x'.\lambda y'.h : \forall t.\mathrm{IsInt}\ t \Rightarrow t \to t \to \mathit{Int}$) and the principal one should express both of them. In order to express this, we use the predicate IsMG in the definition of rules (POLY) and (SPEC) in Fig. 6. The type scheme for **poly** e cannot be derived entirely from the type of e, because the context can place further constraints on it, for example by passing the expression to a function which expects

an argument with a more restricted residual type; the use of the predicate IsMG in the rule (POLY) permits expressing the principal specialisation of a **poly** by allowing to abstract over those constraints placed by the context: instead of calculating the residual type directly, a scheme variable s can be introduced and constrained with an upper bound; further constraints can be expressed as additional upper bounds to s. Conversely, the use of IsMG in the rule (SPEC) allows the selection of the proper instance for a **spec**; it introduces a lower bound for s, whose conversion establishes how to instantiate the polyvariant expression to the type needed. The rule (SPEC) also uses the source-residual relation, for just the same reason as it is used in the rule (λ^D). The rule for **poly** types – (SR-POLY) in Fig. 7 – is used when a lambda-bound variable is of a **poly** type – see Ex. 3.4-2.

The principal specialisation for the expression in Ex. 3.3 then is

$$\begin{array}{l} \Lambda h', h_1, h_2.\mathbf{let}\ f' = h'[\Lambda h.\lambda x'.h + 1] \\ \qquad\qquad \mathbf{in}\ (h_1[f']@\bullet, h_2[f']@\bullet) \\ : \ \forall s.\mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int})\ s, \\ \quad \mathrm{IsMG}\ s\ (\hat{4} \to \mathit{Int}), \\ \quad \mathrm{IsMG}\ s\ (\hat{1} \to \mathit{Int}) \\ \quad \Rightarrow (\mathit{Int}, \mathit{Int})^D \end{array}$$

The upper bound (IsMG $(\forall t.\mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int})\ s$) introduced by (POLY) is responsible for the use of h' in the principal specialisation for f, and the lower bounds (IsMG $s\ (\hat{4} \to \mathit{Int})$) and (IsMG $s\ (\hat{1} \to \mathit{Int})$) introduced by (SPEC), for the conversion variables h_1 and h_2, respectively.

Example 3.4 These are the principal specialisations for the expressions in Ex. 2.4.

1. $\vdash_{\mathrm{P}} \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D) : \mathbf{poly}\ (\mathit{Int}^S \to^D \mathit{Int}^D)$
   $\hookrightarrow \Lambda h.h[\Lambda h_x.\lambda^D x'.h_x + 1]$
   $: \forall s.\mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int})\ s \Rightarrow \mathbf{poly}\ s$

2. $\vdash_{\mathrm{P}} \lambda^D f.\mathbf{spec}\ f\ @^D\ 13^S$
   $: \mathbf{poly}\ (\mathit{Int}^S \to^D \mathit{Int}^D) \to^D \mathit{Int}^D$
   $\hookrightarrow \Lambda h, h'.\lambda f'.h'[f']@\bullet$
   $: \forall s.\mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int})\ s,$
   $\mathrm{IsMG}\ s\ (\hat{1} \to \mathit{Int})$
   $\Rightarrow \mathbf{poly}\ s \to \mathit{Int}$

Observe that scheme variables are used when a **poly** appears in the source type, and that the predicates constraining them are upper and lower bounds: the upper bounds come from **poly**'s (h in expression 1), and the lower bounds come from **spec**'s (h' in expression 2). Also observe the upper bound in expression 2: this is one additional constraint that every poly expression used as argument to the function must satisfy.

The main result in this paper is the existence of principal type specialisations.

Theorem 3.5 Let us consider Γ, e, and τ such that $e : \tau$ is specialisable under Γ (i.e. $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ for some Δ, e', and σ). Then, there exist e'_p and σ_p st. $\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'_p : \sigma_p$ and for all Δ'', e'', and σ'' st. $\Delta'' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'' : \sigma''$ there exist a conversion C and a substitution R st. $C : (\mid R\,\sigma_p) \geq (\Delta'' \mid \sigma')$ and $C[e'_p] = e''$.

The residual pair $e'_p : \sigma_p$ is the *principal specialisation* of $e : \tau$, under the assignment Γ. The proof follows the lines of the proof of principality for the theory of qualified types [13], the main difference being the rules for polyvariance and the use of conversions inside the language of evidence. The proof proceeds in two steps: first, we define $\vdash_{\mathrm{S}}$, a syntax directed version of $\vdash_{\mathrm{P}}$, and prove that they are equivalent, and then, we define an algorithm $\vdash_{\mathrm{W}}$, and prove that the $\vdash_{\mathrm{S}}$ system is equivalent to $\vdash_{\mathrm{W}}$. The reason for this separation is that comparing the algorithm against a syntax directed system is easier, and so the proofs are simpler. The algorithm $\vdash_{\mathrm{W}}$ (based on the algorithm W [20]) has two interesting cases: in the rule for polyvariant expressions, and in lambda abstractions when the domain type is polyvariant. In both cases the algorithm introduces a new type scheme variable, and constraints it with an appropriate IsMG predicate. A detailed version of the proof can be obtained from the technical report [19] or [18].

It is important to observe that the proof of principality is constructive, involving an algorithm producing principal specialisations (or failing when none exists). Additionally, some heuristics for the actual implementation can easily be devised. For example, when the principal type of a polyvariant expression is not a type scheme, there is no need to introduce additional constraints to instantiate it.

4. POST-PROCESSING AND IMPLEMENTATION

There are three post-processing phases that can be applied to the output of the algorithm. There is a degree of freedom in choosing none, just the first, the first two, or all of them to be applied; but in order to get optimal code, all are needed.

The first one, *constraint solving*, establishes how to solve sets of constraints, or determines in some cases that the set is ambiguous. It chooses one solution out of many, and thus the result after this phase may no longer be the principal one. When considering this framework for modular specialisation, constraint solving has to be divided among those steps which do not lose principality and those which do, so as much work as possible can be done for each module. Constraint solving proceeds by instantiating type and scheme variables. Constraints for integers can be solved when the type is known to be a one-point type, which corresponds to knowing the static value of an integer computation in the source code. If the information is not available, we can sometimes establish that the specialisation is ambiguous, as in $\vdash_{\mathrm{P}} \mathbf{let}^D\ f = \lambda^D x.\mathbf{lift}\ x\ \mathbf{in}\ 2^D : \mathit{Int}^D \hookrightarrow \Lambda h.\mathbf{let}\ f' = \lambda x'.h\ \mathbf{in}\ 2 : \forall t.\mathrm{IsInt}\ t \Rightarrow \mathit{Int}$. The interesting case arises when solving constraints for polyvariant expressions. A given scheme variable is constrained with upper and lower bounds, and our algorithm calculates the greatest lower bound of all upper bounds, and instantiates the scheme variable to it. When there is more than one solution, it can be shown that after all postprocesses, every solution will produce the same residual term.

The second one, *evidence elimination*, eliminates evidence abstractions and applications coming from **poly** and **spec** annotations in the source code in favour of tuples – see Ex. 3.3 and compare it with Ex. 2.3 – and reduces the β_v redexes. This phase is obtained with a small variation in the algorithm for constraint solving – each conversion for an upper bound should be changed to another one producing a tuple, and each conversion for a lower bound should be changed to another one producing the right projection – thus showing that the specialisations of the original sys-

tem can also be obtained as instances of ours, and then, the property of principality is much more general than which is implied by Theo. 3.5.

And the third one, *arity raising*, eliminates dummy values and static tuples. It was specified in [11], and previously called *void erasure* in [10].

We have implemented a prototype of the specialisation algorithm, including constraint solving and post-processing phases, in the language Haskell. It has been useful to polish the ideas presented and to test different heuristics for constraint solving identifying problems in it; we are currently working in the design of a more powerful constraint solver based on the experience gained. However, the prototype is not strong enough to be applied to more real world situations; it is mandatory the design and implementation of a full-fledged type specialiser. The code of the prototype can be downloaded from

URL: `http://www-lifia.info.unlp.edu.ar/~fidel/`
File: `Works/PTS/PTS.tgz`.

5. ADDING FEATURES TO THE SOURCE LANGUAGE

The language presented here is only a fragment of the one in [10], but rich enough to present the problem of principality. Extensions to treat the full language can be made in an independent way, without affecting our main result.

The first extension that can be made is the addition of booleans and if-then-else. Besides the expected predicates (such as IsBool and those for static operators), we need to introduce conditional predicates ($\tau?\delta$ and $!\tau?\delta$) in order to defer the principal specialisation of both branches but with only one being effectively specialised at the end (a predicate with a guard set to *False* can be solved trivially). The introduction of conditional predicates forces us to introduce predicates to defer unification and failure. Similar extensions are needed to handle sum types and case expressions.

In order to specialise static functions, a predicate expressing the specialisation of the body is introduced, its evidence being a special function that can be reduced by a special reduction rule (similar to β_v). This predicate is similar in nature to the IsMG predicate. Static recursion is handled in a similar way.

Similarly to partial evaluation, dynamic recursion is problematic, and poses the most challenging problem. We can calculate the constraints expressing the principal specialisation of dynamically recursive programs, but as its combination with polyvariance gives rise to IsMG predicates where a scheme variable appears on both sides, constraint solving is more involved. Our solution is, at present, an ad-hoc heuristic handling several interesting examples; however, our approach helps to precisely formulate the problem, and clarifies the way information flows, thus contributing to the finding of solutions for this problem.

One important issue is the size of the generated predicates: they can be quite complex. Simplification [14] and adding local declarations into types to factorize common subexpressions are ways to alleviate this problem. Constraint solving is another way to reduce the size of predicates, although at the cost of principality (because it chooses a solution among many). It is still an open problem to determine if this complexity is inherent to type specialisation, or it is something introduced by our approach.

6. RELATED WORK

As stated in the introduction, partial evaluation works by generalized reduction, while type specialisation works by generalized type inference. We have stressed that type specialisation is a *different* approach to program specialisation than partial evaluation – it is a common misunderstanding, while other techniques for program specialisation, such as deforestation or supercompilation, although more similar in nature to partial evaluation, are never confused with it.

The main difference is that in partial evaluation the overall type of the residual program is *not changed* – types of subexpressions can be changed, but only in limited ways; type specialisation can produce arbitrary types from a given source program. Another difference is that in partial evaluation, annotations are restricted and thus a "best" annotation can be computed; in type specialisation there do not exist something as a best annotation, and thus, annotations are part of the input. This issue has been misunderstood in the past, with several people asking "why to worry about this contrived annotation, while this other will do?": because by deciding the annotations, we decide which specialisations are allowed and which not. In addition, the flexibility in the annotations allows us to specialise more programs: for example, in the following recursive program, the annotation is considered invalid even by a CPS partial evaluator, while type specialisation can produce the residual shown.

Example 6.1 Observe that the result of the recursive function f contains a static part, but it appears under a dynamic recursion, and thus, under a potentially infinite number of distinct dynamic contexts.

$$
\begin{array}{l}
\vdash_{\mathrm{P}} \mathbf{let}^S\ f = \mathbf{fix}^D\ (\lambda^D f.\lambda^D n. \\
\qquad\qquad \mathbf{if}^D\ n ==^D 0^D \\
\qquad\qquad\ \mathbf{then}\ (1^S, 2^D)^D \\
\qquad\qquad\ \mathbf{else\ let}^D\ p = f\ @^D\ (n -^D 1^D) \\
\qquad\qquad\qquad\ \mathbf{in}\ (\mathbf{fst}^D\ p, \mathbf{snd}^D\ p *^D \mathbf{snd}^D\ p)^D\ ) \\
\quad \mathbf{in}\ \lambda^D n.\mathbf{lift}\ (\mathbf{fst}^D\ (f\ @^D\ n)) \\
: Int^D \to^D Int^D \\
\hookrightarrow \lambda n.1 : Int \to Int
\end{array}
$$

But while partial evaluators are very well engineered, type specialisation is currently at prototype stage and thus not widely usable in practice, yet.

Inspired by type specialisation, Peter Thiemann has proposed a partial evaluator with first-class polyvariance and co-arity raising [26]. He shows that these two features are enough to have optimal specialisation of typed interpreters.

Regarding polymorphism and modules, Heldal has written a partial evaluator for a polymorphic source language with modules [9]; he showed how to generate a residual program with different modules from a single source program, but it is not clear that he can generate polymorphic programs from monomorphic ones.

Type-directed partial evaluation [4, 22] is a simple method for implementing powerful partial evaluators that uses reification[3] in a key way. Similarly to our work, residual terms with different types can be obtained from the same source term, but this is achieved by varying the type guiding the reification: the residual type is an *input* to the specialisation process, instead of an output. The benefit of this approach

[3] *Reification* translates from a semantic domain back to the syntactic domain.

over type specialisation is that it is a simpler one, where the symbolic reduction mechanism *is* the operational semantics of the language; in our approach, constraint solving is used for symbolic reduction. Even when the ML implementation of TDPE can handle polymorphism in source terms [22], there is no attempt to take polymorphic types to guide the reification – polymorphic TDPE is still an open problem.

Tag elimination [24, 25] is a transformation that removes type tags as a post-processing phase to traditional partial evaluation. Similarly to TDPE, it uses the desired residual type as input, and performs a type checking of the subject program after the interpretation, removing superfluous tags. The main contribution of tag elimination is that theoretical results about the process can be easily proved: e.g. Jones-optimality for a typed object language, and that performing tag elimination is exactly the same as type-checking the term being interpreted; the only theoretical results about the power of type specialisation are its correctness [12] and the principality established here, although there is also an example showing that optimal interpretation for typed lambda calculus can be achieved [10].

Ohori [21] has developed a framework to efficiently implement a language with polymorphic primitives. His work resembles that of Jones [13] (his *kinds* corresponding to Jones' predicates), and it is very similar in some technical aspects to our work: his transformation of polymorphic primitives into a pair of a low-level operation and a type attribute resembles closely our treatment of polyvariance. The key difference is it is a compilation mechanism, instead of a framework for program generation.

Other approaches to program specialisation include different techniques for staging a program. One of such is multi-stage specialisation, by Glück and Jørgensen [6], which is a partial evaluation technique for which the static data is provided in several stages, and thus every stage except the last one produces a generating extension. Our system performs a kind of multi-staging, because the static data may not be completely known at the same time.

7. CONCLUSIONS

We have presented a new system of rules specifying type specialisation for a small functional language, and proved that it produces principal specialisations.

The technique used consists of introducing residual type variables and predicates to restrict those variables. The main contributions are the following. We proved principality for type specialisation of a functional language with simple static constructions and first-class polyvariance. We defined two phases for specialisation, the first one syntax directed, and the second one based on constraint solving. We express polyvariance using schemes and conversions giving us a better understanding of this feature; the way we use conversions is also new. Additionally, a prototype of the algorithm has been implemented in the language Haskell, including constraint solving and the post-processing phases. The system is a necessary step towards the introduction of parametric polymorphism on both the source and residual languages.

8. REFERENCES

[1] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*, North Holland, 1988. IFIP World Congress Proceedings, Elsevier Science Publishers B.V.

[2] Luis Damas and Robin Milner. Principal type-schemes for functional languages. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, New Mexico, January 1982.

[3] O. Danvy and A. Filinski, editors. *Programs as Data Objects II*, volume 2053 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, May 2001.

[4] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of 23rd ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages (POPL '96)*, pages 242–257, St. Petersburg Beach, Florida, USA, January 1996. ACM Press.

[5] John Gallagher, editor. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation (PEPM '97)*, Amsterdam, The Netherlands, June 1997. ACM.

[6] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialisation. In Manuel V. Hermenegildo and S. Doaitse Swierstra, editors, *Proceedings of 7th International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP '95)*, volume 982 of *Lecture Notes in Computer Science (LNCS)*, Utretch, The Netherlands, September 1995. Springer-Verlag.

[7] Carster K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. In *Journal of Functional Programming*, volume 1 of *1*, pages 21–70, January 1991.

[8] J. Hatcliff, T. Mogensen, and P. Thiemann, editors. *Partial Evaluation - Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science (LNCS)*, Copenhagen, Denmark, June 1998. Springer-Verlag.

[9] R. Heldal. *The Treatment of Polymorphism and Modules in a Partial Evaluator*. PhD thesis, Chalmers and Göteborg Universities, 2001.

[10] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glck, and Peter Thiemann, editors, *Selected papers of the International Seminar "Partial Evaluation"*, volume 1110 of *Lecture Notes in Computer Science*, pages 183–215, Dagstuhl, Germany, February 1996. Springer-Verlag, Heidelberg, Germany.

[11] John Hughes. A type specialisation tutorial. In Hatcliff et al. [8], pages 293–325.

[12] John Hughes. The correctness of type specialisation. In Smolka [23], pages 215–229.

[13] Mark P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[14] Mark P. Jones. Simplifying and improving qualified types. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pages 160–169, June 1995.

[15] Mark P. Jones. Type classes with functional dependencies. In Smolka [23], pages 230–244.

[16] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In Bjørner et al. [1], pages 1–14.

[17] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science, 1993. Available online at URL: `http://www.dina.dk/~sestoft/pebook/pebook.html`.

[18] Pablo E. Martínez López. *Type Specialisation for Polymorphic Languages*. PhD thesis, University of Buenos Aires, 2002. In preparation.

[19] Pablo E. Martínez López and John Hughes. Towards principal type specialisation. Technical report, University of Buenos Aires, 2001. URL: `http://www-lifia.info.unlp.edu.ar/~fidel/` File: `Works/PTS/towardsPTS.dvi.tgz`.

[20] Robin Milner. A theory of type polymorphism in programming. In *Journal of Computer and System Sciences*, volume 17 of *3*, 1978.

[21] Atsushi Ohori. Type-directed specialization of polymorphism. *Information and Computation*, 1999.

[22] Tim Sheard. A type-directed, on-line, partial evaluator for a polymorphic language. In Gallagher [5], pages 22–35.

[23] Gert Smolka, editor. *Proceedings of 9th European Symposium on Programming (ESOP 2000)*, volume 1782 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, March/April 2000.

[24] Walid Taha and Henning Makholm. Tag elimination - or - type specialisation is a type-indexed effect. In *APPSEM Workshop on Subtyping & Dependent Types in Programming*, Ponte de Lima, Portugal, July 2000.

[25] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Danvy and Filinski [3], pages 257–275.

[26] Peter Thiemann. First-class polyvariant functions and co-arity raising, November 2000. Unpublished manuscript. Available from URL: `http://www.informatik.uni-freiburg.de/~thiemann/` File: `papers/fcpcr.ps.gz`.

APPENDIX

A. EXAMPLES

In this appendix we present some bigger examples of the use of type specialisation. We have decided to postpone their presentation to the very end because they use some of the additional features that we have not described, so their inclusion in the main body of the paper will make it non-self-contained – the details of extensions can be obtained from [18]. However, there is a need for more compelling examples, an issue which we address in this section.

Our first example uses an interpreter for lambda calculus with let-constructs, presented in Figure 8. It uses datatype LE^S to represent lambda expressions, and datatype $Value^S$ to represent the result of the interpretation. The interpretation is given by the function $eval : LE^S \to^S Value^S$. The environment is represented as a function $Char^S \to^S MP^S$, which is the type of the first argument in function *preeval*.

It is interesting to observe with some care the annotations. Very few constructs are annotated dynamic: lambda abstractions in the *Lam* branch, application in the *App* branch, the outer **let** in the *Let* branch, and the numbers, which are lifted, in the *Con* branch; all the rest is static. This implies that the residual code will be formed only with these constructs, thus representing lambda terms without tags. In particular, notice that the tags of datatype $Value^S$ are marked static (even when the argument is dynamic!), and thus they will appear in the *residual type*.

An important feature of the example is the way polyvariance is used to make the lambda calculus typed with a Hindley-Milner polymorphic style: every let-bound expression in the object program is made polyvariant (in Let^S branch), and thus stored in the environment, being specialised when it is used (in the Var^S branch). This is only possible because polyvariance is first-class. By removing all the polyvariance from this example we get a simply typed lambda calculus, as the one presented in [10].

We consider the intepretation of the lambda expression

$$\textbf{let } i = \lambda x.x \textbf{ in } (i@i)@0$$

encoded as a LE^S expression. We present both the specialisation using the original formulation and ours, so differences can be appreciated.

Example A.1 The following expression completes the definitions given in Figure 8.

$$\begin{array}{l} meval\ @^S\ (Let^S\ 'i'\ (Lam^S\ 'x'\ (Var^S\ 'x'))\\ \qquad\qquad (App^S\ (App^S\ (Var^S\ 'i')\ (Var^S\ 'i'))\\ \qquad\qquad\quad (Con^S\ 0^S\ ))) \end{array}$$

The specialisation using the original formulation is the following (we show the result before the arity raising, because it simplifies the comparison with the other specialisation).

$$\begin{array}{l} \textbf{let } v = (\lambda v'.v', \lambda v'.v')^S\\ \textbf{ in } ((\textbf{fst}^S\ v@\textbf{snd}^S\ v)@0) : Num\ Int \end{array}$$

Observe the two copies of the identity function in the static pair; they correspond to each of the monomorphic instances used in the code, which can be observed in the use of static projections when using the residual function v. Polyvariance introduces one copy of the residual function for each different residual type – in this case, one for each monomorphic instance of the polymorphic identity: the type of v is

$$\begin{array}{l} (Fun\ (Fun\ (Num\ Int \to Num\ Int)\\ \qquad \to Fun\ (Num\ Int \to Num\ Int)),\\ \ Fun\ (Num\ Int \to Num\ Int))^S \end{array}$$

which shows that the type $Value^S$ has been specialised precisely to the types at which the instances of the identity function were used.

The specialisation of the same expression using the formulation presented in this paper is the following.

$$\begin{array}{l} \textbf{let } v = \Lambda h.\lambda v'.v'\\ \textbf{ in } (v(\!(Fun)\!)@(v(\!(Fun)\!)))@0 : Num\ Int \end{array}$$

Observe how the residual of the let-bound identity function is an evidence abstraction, instead of a tuple; its type is

$$\textbf{poly } (\forall t.\text{IsResidualOf } t\ Value^S \Rightarrow Fun\ (t \to t))$$

– the evidence variable h is waiting for evidence that type t is the residual type of an expression of source type $Value^S$. So, in order to use v, it must first be provided with suitable evidence that the corresponding instance of t satisfy the predicate, which is done by evidence application; the first *Fun* is the evidence that the first component of the residual type of v in the previous specialisation is the residual of $Value^S$, and that of the second one is the evidence

$$
\begin{array}{l}
\textbf{data}\ LE^S\ =\ Var\ Char^S\ |\ Con\ Int^S\ |\ Lam\ Char^S\ LE^S\ |\ App\ LE^S\ LE^S\ |\ Let\ Char^S\ LE^S\ \ LE^S \\
\textbf{data}\ Value^S\ =\ Num\ Int^D\ |\ Fun\ (Value^S \rightarrow^D Value^S)\ |\ Wrong \\
\textbf{data}\ MP^S\ =\ M\ Value^S\ |\ P\ (\textbf{poly}\ Value^S) \\
\textbf{let}^S\ bind = \lambda^S x.\lambda^S v.\lambda^S env.\lambda^S y.\textbf{if}^S\ x == y\ \textbf{then}\ v\ \textbf{else}\ env\ @^S\ y \\
\textbf{in let}^S\ preeval = \textbf{fix}^S\ (\lambda^S eval.\lambda^S env.\lambda^S expr. \\
\qquad \textbf{case}^S\ expr\ \textbf{of} \\
\qquad\quad Var\ x \rightarrow \textbf{case}^S\ (env\ @^S\ x)\ \textbf{of}\ \{M\ v \rightarrow v; P\ \ v \rightarrow \textbf{spec}\ v\} \\
\qquad\quad Con\ n \rightarrow Num^S\ (\textbf{lift}\ n) \\
\qquad\quad Lam\ x\ e \rightarrow Fun^S\ (\lambda^D v.\textbf{let}^S\ env' = bind\ @^S\ x\ @^S\ (M^S\ v)\ @^S\ env \\
\qquad\qquad\qquad\qquad \textbf{in}\ eval\ @^S\ env'\ @^S\ e) \\
\qquad\quad App\ e_1\ e_2 \rightarrow \textbf{case}^S\ (eval\ @^S\ env\ @^S\ e_1)\ \textbf{of} \\
\qquad\qquad\qquad Fun\ f \rightarrow f\ @^D\ (eval\ @^S\ env\ @^S\ e_2) \\
\qquad\quad Let\ x\ e_1\ e_2 \rightarrow \textbf{let}^D\ v = \textbf{poly}\ (eval\ @^S\ env\ @^S\ e_1) \\
\qquad\qquad\qquad\qquad \textbf{in let}^S\ env' = bind\ @^S\ x\ @^S\ (P^S\ s)\ @^S\ env \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ eval\ @^S\ env'\ @^S\ e_2) \\
\textbf{in let}^S\ meval = preeval\ @^S\ (\lambda^S x.M^S\ \ Wrong^S) \\
\textbf{in}\ \langle \ldots \rangle
\end{array}
$$

Figure 8: An interpreter for lambda-calculus, annotated to produce monomorphization.

that the second component of that type is also a residual of $Value^S$. This specialisation can be transformed into the first one by the process of evidence elimination.

Observe that the type of the residual identity function is polymorphic (although annotated and qualified). There is still a difference from a truly parametric polymorphic type – we are working on a variation of evidence elimination that will produce the desired type and code.

Our final example shows the specialisation of an expression that uses a static function producing an infinite static list. It uses the datatype

$$\textbf{data}\ List^S\ t\ =\ Nil\ |\ Cons\ t\ (List^S\ t)$$

of lists with a static spine. In order to observe how static lists are specialised, we present two simple finite lists first.

Example A.2 These examples can be specialised identically with the original formulation.

1. $\vdash_{\mathrm{P}} Nil^S : List^S\ \tau \hookrightarrow ()^S : Nil$
2. $\vdash_{\mathrm{P}} Cons^S\ 42^D\ (Cons^S\ 17^D\ Nil^S) : List^S\ Int^D$
   $\hookrightarrow (42, (17, ()^S)^S)^S : Cons\ Int\ (Cons\ Int\ Nil)$

Observe in the two cases how the static constructors of the lists are moved into the residual type, while the dynamic values are preserved in a static tuple in the residual term – which will be further eliminated by arity raising.

Example A.3 The specialisation shown in Fig. 9 is obtainable with our approach, but there is no specialisation in the original formulation for it. The example shows that we can represent 'infinite' static structures using qualified residual types; the predicate IsFixS will produce the unfolding during constraint solving. Both the original formulation and our present constraint solver will loop if this unfolding is ever attempted; however, we have now the possibility to make a constraint solver taking care of static lazy evaluation.

$$
\begin{array}{l}
\vdash_{\mathrm{P}} \textbf{let}^D\ f = \textbf{fix}^S\ (\lambda^S f.\lambda^S x.Cons^S\ 1^D\ \ (f\ @^S\ x)) \\
\quad \textbf{in case}^S\ f\ @^S\ ()^S\ \textbf{of} \\
\qquad\quad Cons\ x\ xs \rightarrow x \\
\quad : Int^D \\
\quad \hookrightarrow \Lambda h_U, h_L, h_{y_s}.\textbf{let}\ f = \bullet \\
\qquad\qquad\qquad\qquad \textbf{in fst}^S\ (h_L @_v f @_v ()^S) \\
\qquad : \forall t, t_{y_s}, t_e.\mathrm{IsFixS}\ \mathrm{cl}_{x_s}\ t \\
\qquad\qquad\qquad \mathrm{IsFunS}\ t \\
\qquad\qquad\qquad\quad \textbf{clos}(t_e : ()^S \rightarrow Cons\ Int\ t_{y_s}), \\
\qquad\qquad\qquad \mathrm{IsConstrOf}\ (List^S\ \ Int^D)\ t_{y_s}, \\
\qquad\qquad\qquad \Rightarrow Int \\
\text{where} \\
\mathrm{cl}_{x_s} = \\
\quad \textbf{clos}(\Lambda h_f, h_r.\lambda f_s.\lambda f.(f)^S \\
\qquad : \forall t_f, t_r, t_e.\mathrm{IsFunS}\ \mathrm{cl}_f(t_e)\ t_f, \\
\qquad\qquad\qquad \mathrm{IsFunS}\ \mathrm{cl}_r(t_f)\ t_r \\
\qquad\qquad\qquad \Rightarrow t_f \rightarrow t_r) \\
\mathrm{cl}_f(t_e) = \\
\quad \textbf{clos}(t_e : \forall t_{x_s}.\mathrm{IsConstrOf}\ (List^S\ \ Int^D)\ t_{x_s} \\
\qquad\qquad\qquad \Rightarrow ()^S \rightarrow t_{x_s}) \\
\mathrm{cl}_r(t_f) = \\
\quad \textbf{clos}(\Lambda h'_{y_s}, h'_L.\lambda^S f'_s.\lambda^S x.(1, h'_L @_v \pi_{1,1}\ f'_s @_v x)^S \\
\qquad : \forall t'_{y_s}, t'_e.\mathrm{IsConstrOf}\ (List^S\ \ Int^D)\ t'_{y_s} \\
\qquad\qquad\qquad \mathrm{IsFunS}\ t_f\ \textbf{clos}(t'_e : ()^S \rightarrow t'_{y_s}) \\
\qquad\qquad\qquad \Rightarrow ()^S \rightarrow Cons\ Int\ t'_{y_s})
\end{array}
$$

Figure 9: A term with an infinite list.