Simplifying and Solving Qualified Types for Principal Type Specialisation

Pablo E. Martínez López and Hernán Badenes

Laboratorio de Investigación y Formación en Informática Avanzada
Universidad Nacional de La Plata
CC 11, 1900, La Plata, Argentina
{fidel,hernan}@lifia.info.unlp.edu.ar

Abstract. Principal Type Specialisation is an approach to Type Specialisation designed to generate polymorphic residual programs and giving chance of modular specialisation. Principality is obtained by using a system of qualified types (types enriched with predicates, or constraints) to defer some decisions until link-time, when information from the whole program has been gathered.
In order to complete specialisations, it is necessary to provide "solutions" for the predicates, together with evidence that they hold. In this paper we address the problem of *simplifying* and *solving* predicates produced by principal type specialisation. We give a formalization of the simplification and solving problems.
The simplification process (essentially the task of removing redundant predicates) generates substitutions and conversions as tools for soundly modifying terms and types accordingly. We study the basic properties of simplifications in an abstract way, and implement one simplification relation. The process of solving predicates is then defined in terms of simplification and both notions are incorporated to the specialisation process.

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 $\lambda x \to x * (x * x)$, and similarly for other exponents.

There are different approaches to specialisation, *Partial Evaluation* [1] being the most popular, well-known and well-engineered. Partial evaluation specialises programs by a form of generalized evaluation: expressions which depend on known data are replaced by the result of computing them.

A good way to determine the limits of a specialisation method is to specialise a self-interpreter with it for a given object program and compare the residual program with the object 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. For typed interpreters, the residual code will contain type information coming from the representation of programs in the interpreter: optimality is lost. This problem was stated by Neil Jones in 1987 as one of the open problems in the partial evaluation field [2]. Although recently it has been argued that there are ways of reading this situation that avoid the problem [3] we stick to the tradition here.

Type Specialisation [4] 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. The original formulation of type specialisation [4] has some problems, one of them being that there are rules that are not completely syntax directed; 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.

Principal Type Specialisation [5] is an approach to type specialisation which exhibits the principality property –it is possible to derive a 'more general specialisation' for every term– and it was designed to solve some of the problem of classic type specialisation. Further technical details are given in the next section, as they are needed to understand our work.

Our contribution is to formalize some of the postprocessing phases of Principal Type Specialisation and to instantiate them by implementing simple algorithms for simplifying and solving constraints generated (see section 2). In this regard, this paper can be seen as a follow-up of [5].

This paper is structured as follows. In section 2 we make a brief presentation of Principal Type Specialisation, introducing the technical matter needed as background and motivating our work. Then, in section 3 we describe the first half of our work: the simplification of predicates (constraints), specifying what a simplification relation is, implementing a simplification relation for our language, and introducing this concept into the type specialisation process. Section 4 addresses our main problem, the solving of constraints. We specify what a solving is (in terms of simplifications), and describe how to perform it for specialisations. Then we give a simple heuristic for solving the constraints presented in the language we study – it is important to know that it is only an exemplary algorithm; the search for more powerful alternatives is subject of future work. Finally we give some lines of future and related work and the conclusions in sections 5 and 6.

2 Principal Type Specialisation

In this section we mention some essential features of Principal Type Specialisation approach. However, we refer the reader to the original work of Martínez

López and Hughes [5] to get further details, as a strongly recommended introduction to fully understand this work.

The source language specialised is a λ-calculus enriched with arithmetic and boolean constants and operations, tuples, **let-in**, **if-then-else** and **case** constructs. Source expressions are annotated with ^{S} or ^{D} to indicate static (resp. dynamic) expressions, specifying that the information must (resp. must not) be moved from the term into the type. This information is given as input to the specialiser, and might have been provided by the user or by some kind of binding-time analysis.

The language of expressions includes a special annotation for lifting a static value into a dynamic one: **lift**. It also allows the use of polyvariance – i.e. the ability to obtain different versions of the same term in the same specialisation (e.g. for distinct arguments) – by wrapping polyvariant expressions with the construct **poly**, and making different specialisations of them with the construction **spec**, each time they are used. Several extensions to this source language are also studied (as recursion or datatypes), but we limit our work to those mentioned.

The principality property is obtained by defining the specialisation on a system of qualified types [6] as residual type language: residual types are enriched with predicates constraining quantification of type variables. These predicates are generated during specialisation and specify some conditions the terms and types should hold in their final form. Residual terms are typed with schemes (qualified types with quantification for type variables), denoted $\forall\alpha.\Delta \Rightarrow \tau'$, where α is a list of variables, Δ a list of predicates and τ' a residual type. Residual types definition contains, besides the usual constructions, an infinite set of *one-point types*, denoted $\hat{n}$ or $\hat{b}$ for each integer or boolean value, meaning that the term's value is known at specialisation time to be n (resp. b).

During specialisation, *predicates* (denoted δ) are generated; for example: IsInt t (meaning t must resolve to some one-point integer), $t := t' \otimes t''$ (forcing t to be the result of the arithmetic operation with operands t' and t'') and IsMG $\sigma\ \sigma'$ (constraining scheme σ to be 'more general' than σ', according to an ordering $\geq$ defined in terms of instantiation and a relation between predicates called entailment – prescribed by the Theory of Qualified Types). A language of *conversions* C is defined as special terms which transform expressions of one type to expressions of another (less general), denoted $C : \sigma \geq \sigma'$. They are defined in terms of $\geq$, and can be seen as the semantics of IsMG predicates.

Predicates are labelled with *evidence variables*, in the form $h : \delta$. When predicates are proved, those variables are replaced by some *evidence values*, (including integer constants n and conversions C, as well as evidence variables) and this evidence can fill holes or modify the shape of terms.

In specialisation judgements, predicates are stored in the context, as a *predicate assignment*: a list of predicates $h_i : \delta_i$. It is desirable when specialising a term to obtain as final result a program without evidence variables nor predicates, informally meaning that it has been completely specialised.

A word about notation: following [6], we use some abbreviations to improve readability (at least once conventions are understood): for instance, if

$\Delta = \delta_1 \ldots \delta_n$ and $h = h_1 \ldots h_n$, we denote the predicate assignment $h_1 : \delta_1, \ldots, h_n : \delta_n$ by $h : \Delta$. There are a few additional abbreviations, which we hope are intuitive enough to omit here – for the full details refer to [5] or [6].

Specialisation judgements are denoted $h : \Delta \mid \Gamma \vdash_{P} e : \tau \hookrightarrow e' : \sigma$, meaning that the source term e of type τ can be specialised to the residual program e' of type σ, if its free variables specialise as Γ asserts (a list of specialisation assignments of the form $x : \tau \hookrightarrow x' : \tau'$) and if the predicates of Δ hold (proved by unknown evidence, represented by h). Γ is used in specialisations in a similar way as type assignments are in any Hindley-Milner typing procedure. We illustrate some of the concepts mentioned in this section in the following example.

Example 1. The following is a simple specialisation:

$$h : \text{IsInt } t \mid \emptyset \vdash_{P} \lambda^{D} x.\mathbf{lift}\ x : \text{Int}^{S} \rightarrow^{D} \text{Int}^{D} \hookrightarrow \lambda x'.h : t \rightarrow \text{Int}$$

Note the correspondence of static and dynamic annotation between the source term and its type; and the way unknowns are denoted in the residual expression as evidence variables (in the term) and type variables (in its type), and related by the predicate in the context.

The specialisation process is specified by a system of rules, called P (it stands for Principality, in contrast to the Original system O of [4]), later modified to obtain the syntax-directed system S, and finally implemented algorithmically by the system W. In this paper we shall include simplification and solving to the specialisation process by adding new rules to systems P and W.

3 Simplification

As we have said, specialisation generates predicates which restrict the scope of quantification of type variables; and those predicates are labelled with evidence variables. We motivate the need of simplifying them with a simple example.

Example 2. Recall example 1, where the function that lifts static values into dynamic ones is specialised. If this expression were used within a context where the type variable t got unified with some one-point type, say $\hat{3}$, the predicate in the context would become $h : \text{IsInt } \hat{3}$: and then the value 3 could be used to prove this predicate. Using that evidence in the program would give the desired term $\lambda x.3$. But the predicate in the context must still be eliminated; this is one of the tasks simplification must perform.

The rest of this section will be devoted to the formalization of the concept of simplification, and the implementation of a simplification algorithm capable of handling the constraints produced by the principal type specialiser.

3.1 Specifying Simplifications

To formalize the notion of simplification we define it in an abstract way as a relation based on the entailment relation $\Vdash$, with no concern about what simplifications can actually perform.

Definition 3 (Simplification). *A relation $\trianglerighteq$ between two predicate assignments $h : \Delta$ and $h' : \Delta'$, with a substitution S and a conversion C as witnesses, is called a* simplification*, and denoted*

$$S; C \mid h : \Delta \trianglerighteq h' : \Delta'$$

if it holds that: (i) $h' : \Delta' \Vdash v : S\Delta$, (ii) $S\Delta \Vdash \Delta'$, (iii) $C = h \leftarrow v$.

The first condition states that the resulting (simplified) assignment is strong enough to proof (entail) the instantiation by S of the original assignment, constructing some evidence v; the second states that the instantiation of the first predicate assignment can prove the simplified one. In some way, the double entailment ensures that no arbitrary decisions are being made while changing predicates or instantiating them by S; the only changes are those that do not narrow the world of possible specialisations. The conversion C (which converts terms using the variables of assignment Δ, to terms using that – usually fewer – of Δ') must provide the evidence v whenever evidence variables h appear.

We shall use the notation $h \leftarrow v$ as just an abbreviation for $(\Lambda h.[])((v))$, where the variable h is abstracted and then the evidence v is applied. This notation will be used to emphasize that evidence variables are being replaced by evidence values; while abstraction and application are just the tool to do it, they can be confused with those generated by the moving of predicates from the context to the type and back, using standard rules (QIN) and (QOUT) of Qualified Types Theory. The equality of conversions used in the rule is not syntactic but semantic, that is, $C = C'$ if for $C[e'] = C'[e']$ for every term e', as defined by [5] in terms of β_v- and η_v-reduction relations for conversions.

Each relation satisfying definition 3 is suitable to be used as a simplification relation, while in the practice we would require additional properties, like some kind of reduction (to actually simplify), confluence of simplifications, etc. – but we allow any simplification (e.g. the identity) to fit in the definition. In the next section we shall implement a simplification for our language.

Viewing a simplification as a reduction relation between predicate assignments (with additional 'witnesses' like the substitution and conversion), it should be interesting to show that the reduction is stable under substitutions. Unfortunately, this is not true for definition 3, due to the substitution involved in the entailment. A weaker condition is still true, and will be enough for our purposes. We say two substitutions S and T are *compatible* for a predicate assignment Δ, denoted $S \leftrightarrow_{\Delta} T$, if $S\Delta = T\Delta$. Using this definition, the following property holds:

Proposition 4. *If $S \leftrightarrow_{\Delta} T$ and $T; C \mid \Delta \trianglerighteq \Delta'$ then $T; C \mid S\Delta \trianglerighteq S\Delta'$.*

$$\text{(SimEntl)}\quad \frac{h : \Delta \Vdash v_\delta : \delta}{\mathrm{Id}; h_\delta \leftarrow v_\delta \mid h : \Delta, h_\delta : \delta \trianglerighteq h : \Delta}$$

$$\text{(SimComp)}\quad \frac{S; C \mid h : \Delta \trianglerighteq h' : \Delta' \qquad S'; C' \mid h' : \Delta' \trianglerighteq h'' : \Delta''}{S'S; C' \circ C \mid h : \Delta \trianglerighteq h'' : \Delta''}$$

$$\text{(SimCtx)}\quad \frac{S; C \mid h_1 : \Delta_1 \trianglerighteq h_2 : \Delta_2}{S; C \mid h_1 : \Delta_1, h' : \Delta' \trianglerighteq h_2 : \Delta_2, h' : S\Delta'}$$

$$\text{(SimPerm)}\quad \frac{S; h_1, h_2 \leftarrow v_1, v_2 \mid h_1 : \Delta_1, h_2 : \Delta_2 \trianglerighteq h_1' : \Delta_1', h_2' : \Delta_2'}{S; h_2, h_1 \leftarrow v_2, v_1[h_2/v_2] \mid h_2 : \Delta_2, h_1 : \Delta_1 \trianglerighteq h_2' : \Delta_2', h_1' : \Delta_1'}$$

$$\text{(SimCHAM)}\quad \frac{\Delta_1' \equiv \Delta_1 \qquad S; C \mid \Delta_1 \trianglerighteq \Delta_2 \qquad \Delta_2 \equiv \Delta_2'}{S; C \mid \Delta, \Delta_1' \trianglerighteq \Delta, \Delta_2'}$$

Fig. 1. Structural simplification rules

This is an essential property to ensure that sequential steps of our algorithm give a sound solution. (This can be seen as going against principality, but it can be shown that only unsatisfiable models are left out.) It is interesting to note that, as defined, simplification works looking only at the information of the context: it does not matter what the specialised expression is.

3.2 A Simplification Relation

In section 3.1 we defined when a relation between predicate assignments can be called a simplification. In this section we implement a particular simplification relation, to deal with the predicates produced in the specialisation process.

We give an implementation of $\trianglerighteq$ defining it by means of a system of rules. Part of it is shown in figure 1. These are called *structural rules*, since they are not restricted to any particular construct in source or residual languages.

The first rule makes it possible to use the entailment relation to eliminate predicates, when they are implied by the rest of the context – in this way internalising the entailment relation $\Vdash$ on which the theory of qualified types is based.

The second one makes the transitive closure of the relation, choosing suitable substitution and conversion as compositions of the original ones.

The third one extends simplification to work on bigger sets of predicates; it is important to note the use of the substitution S in the right hand side to cancel variables that may have been simplified.

The fourth one allows the treatment of the lists of predicates as if they have no order, by closing the simplification with respect to permutations.

These two rules are complementary to each other, and usually used together. For that reason, we define a derived rule, (SimCHAM), to expand the relation to be used in larger contexts, following the *Chemical Abstract Machine* approach of [7], where the equivalence $\equiv$ is defined as the smallest relation containing $\Delta, \delta, \delta', \Delta' \equiv \Delta, \delta', \delta, \Delta'$, allowing to conveniently permutate predicates (reflexivity of $\trianglerighteq$ is given by this rule, when no changes are made to predicates). It is

$$\text{(SimOp}_{\text{res}}\text{)}\quad \frac{t \sim^S \hat{n}}{S; h_\delta \leftarrow n \mid h_\delta : t := \hat{n_1} \otimes \hat{n_2} \trianglerighteq \emptyset}\ (n = n_1 \otimes n_2)$$

$$\text{(SimMG}_{\text{U}}\text{)}\quad \frac{C : \sigma_2 \geq \sigma_1}{\text{Id}; h_2 \leftarrow (h_1 \circ C) \mid h_1 : \text{IsMG}\ \sigma_1\ s, h_2 : \text{IsMG}\ \sigma_2\ s \trianglerighteq h_1 : \text{IsMG}\ \sigma_1\ s}$$

Fig. 2. Language-dependent Simplification rules

important to note that the order of predicates can be changed only when they are still labelled with evidence variables in a predicate assignment ($h : \delta$); after they are introduced in a type with the (QIN) rule of qualified types theory, the link from the variables to their predicates is only given by the order in which they appear in the expressions ($\Lambda h.\_$ in terms and $\delta \Rightarrow \_$ in types).

To complete the relation defined by the structural rules, we give in figure 2 a (partial) set of rules to deal with some constructs of our language.

The rule (SimOp$_{\text{res}}$) allows to simplify a predicate representing a static operation, when both operands are known to be numeric one-point types. In this case, the predicate is eliminated while the conversion provides the necessary evidence (the result) to the term. With this rule the standard constant folding procedure is performed, solving the constraints of the form $h : t := t' \oplus t''$ generated by the specialiser.

Example 5. Consider this principal specialisation, derived in system P:

$$\begin{array}{l} \Delta \vdash (\lambda^D x.(\mathbf{lift}\ x, x +^S 3^S)^D) \mathbin{@^D} (2^S +^S 1^S) : (\text{Int}^D, \text{Int}^S) \\ \quad \hookrightarrow (\lambda x'.(h, \bullet))@\bullet : (\text{Int}, t') \end{array}$$

with the following predicates as the context:

$$\Delta = h : t := \hat{1} + \hat{2}, h' : t' := t + \hat{3}$$

Removing the predicates by applying (SimOp$_{\text{res}}$) twice, the term becomes

$$(\Lambda h'.(\Lambda h.(\lambda x'.(h, \bullet))@\bullet)((3)))((6)) : (\text{Int}, \hat{6})$$

from which, after β_v-reduction steps (we shall omit them for readability in next examples; since we work under the equivalence between conversions, with include β_v- and η_v-reductions) is converted to

$$(\lambda x'.(3, \bullet))@\bullet : (\text{Int}, \hat{6})$$

This illustrates the way constant folding is performed by type specialisation, and also how the information can flow from the argument to the body of a function, using the information carried by type and evidence variables.

The second rule, (SimMG$_{\text{U}}$), allows to eliminate a redundant IsMG predicate: when two upper bounds of a scheme variable are comparable to each other, then the greatest bound does not provide any further information than the least one.

It can be removed, and the conversion is constructed as the composition of the conversion going from the greatest type to the least one and the conversion of the remaining predicate, abstracted by the evidence variable h.

There can be a variety of rules to handle different types of predicates. For instance, one can think of simplifying variables in arithmetic predicates when there is only one possible solution, or dealing with **case** constructs and patterns to unify and simplify unknowns – we limit to those mentioned for reasons of space.

Example 6. Our simplification relation can simplify $S; C \mid \Delta_1 \trianglerighteq \Delta_2$, where

$$\begin{aligned}
\Delta_1 &= h_1 : \text{IsInt}\ \hat{9}, h_2 : \text{IsInt}\ t''', h_3 : t := \hat{1} + \hat{2}, h_4 : t' := t + \hat{3}, h_5 : t''' := t'' + t' \\
\Delta_2 &= h_5 : t''' := t'' + 6
\end{aligned}$$

using (SimEntl) once to remove IsInt $\hat{9}$, (SimEntl) once again to remove IsInt t''' (entailed by the predicated labelled by h_5), and then ($\text{SimOp}_{\text{res}}$) twice to solve sequentially $t := \hat{1} + \hat{2}$ and $t' := t + \hat{3}$. Composing the applications with (SimComp) and expanding the assignment with (SimCHAM) when needed, the substitution $S = [t/\hat{3}][t'/\hat{6}]$ and conversion $C = (h_1, h_2, h_3, h_4) \leftarrow (9, h_5, 3, 6)$ are obtained.

The following theorem justifies the use of the relation defined above as a simplification, according to definition 3.

Theorem 7. *The rules in figures 1 and 2 define a simplification relation.*

Example 8. It would look intuitive to include a counterpart to (SimMG_{U}), to simplify lower bounds, like:

$$\frac{C : \sigma_2 \geq \sigma_1}{\text{Id}; h_1 \leftarrow C \circ h_2 \mid h_1 : \text{IsMG}\ s\ \sigma_1, h_2 : \text{IsMG}\ s\ \sigma_2 \trianglerighteq h_2 : \text{IsMG}\ \sigma_2\ s}$$

However, there are reasons for not including it as a simplification. For example, lower bounds generated by the **spec** annotation never contain type variables; and these ground schemes can never be comparable each other by $\geq$.

This is an example of how ad-hoc a simplification system is, taking into account not only the specific predicates it must simplify but also the way we expect to have the result, or which things we do not want to get simplified.

3.3 Type Specialisation

The next step in defining simplification is to incorporate the notion into the specialisation process. As a first approach, a new rule is added to specialisation system P, allowing to simplify predicates at any stage of the process:

$$\text{(SIMP)}\ \frac{h : \Delta \mid \Gamma \vdash_P e : \tau \hookrightarrow e' : \sigma \qquad S; C \mid h : \Delta \trianglerighteq h' : \Delta'}{h' : \Delta' \mid S\Gamma \vdash_P e : \tau \hookrightarrow C[e'] : S\sigma}$$

Its use in the system P is justified by the following consistency theorem:

Theorem 9. *If* $h : \Delta \mid \Gamma \vdash_P e : \tau \hookrightarrow e' : \sigma$ *and* $S; C \mid h : \Delta \trianglerighteq h' : \Delta'$ *then* $h' : \Delta' \mid S\Gamma \vdash_P e : \tau \hookrightarrow C[e'] : S\sigma$*.*

In practice, we would need an algorithm to perform simplification, and to decide when it is convenient to actually do it. Provided we define a function *simplify* that returns a 'maximally simplified' (up to some criterion) assignment $h' : \Delta'$, a substitution S and a conversion C, given a non-simplified assignment $h : \Delta$, we should decide where to incorporate it to the specialisation algorithm. The most convenient place is before wrapping types into **poly** type schemes, so they are already simplified (to the extent allowed by the information available at the wrapping) when they are unwrapped for use, saving computational work. In this way we change the rule (W-POLY) of system W, resulting in:

$$\text{(W-POLY)} \quad \frac{h : \Delta \mid S\Gamma \vdash_W e : \tau \hookrightarrow e' : \tau'}{h'' : \text{IsMG } \sigma\ s \mid TS\Gamma \vdash_W \textbf{poly}\ e : \textbf{poly}\ \tau \hookrightarrow e'' : \textbf{poly}\ s}$$

where, being s and h'' fresh variables:

$$\begin{array}{l} e'' = h''[\Lambda h'.C[e']] \\ (h' : \Delta', T, C) = \mathit{simplify}(h : \Delta) \\ \sigma = \text{Gen}_{TS\Gamma}(\Delta' \Rightarrow T\tau') \end{array}$$

This rule simplifies the assignment $h : \Delta$ using the function *simplify* to obtain a new assignment $h' : \Delta'$, a substitution T and a conversion C. It then builds the term $C[e']$ of type $T\tau'$ under assignment Δ', and empties the context simulating the application of (QIN), thus obtaining the term $\Lambda h'.C[e']$ of type $\Delta' \Rightarrow T\tau'$. It finally fills in the context with a new predicate h'' : IsMG $\sigma\ s$ with a fresh variable, using the evidence variable to specify the new conversion at top level of the term. The type used in the IsMG predicate is the generalization over every free variable (not free in Γ), and the predicates are in that way simplified before being wrapped with the **poly** construct. This modification to W is the only one needed to perform simplifications in type specialisation.

There are several aspects of simplification that are not mentioned here. For instance, an analogy of the difference between the concepts of simplification and improving in Jones' work [8] can be found in this model, or the analysis of the flow of data during the specialisation process giving place to distinct characterization of rules – they can be found in the full version of this work [9].

4 Solving Predicates

The motivation of this work is to define the constraint solving process. In presence of **poly** and **spec** annotations, the specialiser does not decide the final form of polyvariant expressions, but abstracts it with evidence variables (put in every definition point **poly** and use point **spec**) until all the information can be gathered. These variables are a key component of principality, as they abstract the different instances of a term, and will be replaced by conversions when the values of scheme variables are decided by constraint solving.

It is not until solving these predicates that the residual expressions can take their final form. This stage is performed after specialisation, and only when there is enough information in the context to allow it – they are related in the same way compiling and linking are.

This kind of static analysis allows to separately *specify* (specialise) and *implement* (solve) the problem, as was pointed out by Aiken [10] for set constraint-based analysis. In this way specialisation can be seen as a static program analysis, and can be done locally (principality of specialisations). It allows modular specialisation, and the conjunction of the analyses of subprograms gives the constraints required for the whole process to finish.

4.1 Specifying Solutions

To begin with, we define when a substitution mapping scheme variables to type schemes can be called a *solution*, when it can be performed, and which other components are needed.

Definition 10 (Solving). *A* solving *from a predicate assignment Δ_1 to Δ_2, requiring the predicates of Δ', is a relation*

$$S\!: T; C \mid \Delta_1 + \Delta' \triangleright_V \Delta_2$$

where S (called the solution*) and T are substitutions, C a conversion and V a set of type variables, such that*

$$\begin{array}{l}\text{(i) } T; C \mid S\Delta_1, \Delta' \trianglerighteq \Delta_2 \\ \text{(ii) } \mathrm{dom}(S) \cap (V \cup \mathrm{FTV}(\Delta')) = \emptyset\end{array}$$

While solving may appear similar to simplifying at a first glance, its consequences are stronger. The substitution S, the solution, may decide the values of some scheme variables and the new (solved) predicate assignment it is not asked to entail the original one (in contrast to simplifying, where both predicate assignments are equivalent in some sense). This is why, when trying to solve a variable, we must be more careful than with simplification: some extra conditions must be observed to prevent taking unsound or premature decisions, as we shall see in the next sections.

4.2 Solving Specialisations

In this section we study how solving can be performed during specialisation, by incorporating it to system P. In contrast to simplifying, some cautions have to be taken to avoid unsound results: if a variable is decided when some of the information affecting its set of possible values is missing (which can happen if a scheme variable occurs anywhere in the residual type or in the type assignment Γ) then it should not be solved. This situation is presented in the following example:

Example 11. Specialising the term e defined as:

$$\begin{array}{l}\textbf{let}^D\ f = \textbf{poly}\ (\lambda^D x.\lambda^D y.(\textbf{lift}\ x, \textbf{lift}\ y))\ \textbf{in}\ (\textbf{spec}\ f\ @^D\ 1^S\ \ @^D\ 2^S\ , f)\\ : ((\mathrm{Int}^D\ , \mathrm{Int}^D), \textbf{poly}\ (\mathrm{Int}^S \to \mathrm{Int}^S \to (\mathrm{Int}^D\ , \mathrm{Int}^D)))\end{array}$$

gives the residual term and types

$$\begin{array}{l}\Lambda h^u_f, h^\ell_f.\textbf{let}\ f = h^u_f[\Lambda h_x.\Lambda h_y.\lambda x.\lambda y.(h_x, h_y)]\ \textbf{in}\ (h^\ell_f[f]@\bullet @\bullet, f)\\ : \forall s.\ \mathrm{IsMG}\ (\forall t, t'\mathrm{IsInt}\ t, \mathrm{IsInt}\ t' \Rightarrow t \to t' \to (\mathrm{Int}, \mathrm{Int}))\ s,\\ \qquad \mathrm{IsMG}\ s\ (\hat{1} \to \hat{2} \to (\mathrm{Int}, \mathrm{Int})) \Rightarrow ((\mathrm{Int}, \mathrm{Int}), \textbf{poly}\ s)\end{array}$$

Had the variable s be solved before moving the predicates from the context into the type, a suitable solution would be to choose $\sigma = \forall t, t'.\mathrm{IsInt}\ t, \mathrm{IsInt}\ t' \Rightarrow t \to t' \to (\mathrm{Int}, \mathrm{Int})$, giving

$$\textbf{let}\ f = \Lambda h_x\, h_y.\lambda x\, y.(h_x, h_y)\ \textbf{in}\ (f(\!(1)\!)(\!(2)\!)@\bullet @\bullet, f) : ((\mathrm{Int}, \mathrm{Int}), \textbf{poly}\ \sigma)$$

In this term, the second component of the tuple can be used for any type instance of $t \to t' \to (\mathrm{Int}, \mathrm{Int})$. This is apparently a good solution. But, if the original term e appears in the following program, and specialised in a monolithic fashion:

$$\begin{array}{l}\textbf{let}^D\ e = \ldots\ \textbf{in}\ \textbf{let}^D\ id = \lambda^D x.x\ \textbf{in}\\ \textbf{let}^D\ g = \textbf{poly}\ (\lambda^D x.\lambda^D y.(\textbf{lift}\ (id\ @^D\ x), \textbf{lift}\ (id\ @^D\ y)))\ \textbf{in}\\ \textbf{spec}\ (\textbf{if}\ \mathrm{True}\ \textbf{then}\ g\ \textbf{else}\ \textbf{snd}\ e)\ @^D\ 4^S\ \ @^D\ 4^S \qquad\qquad : (\mathrm{Int}, \mathrm{Int})\end{array}$$

it can be seen (as a good exercise of type specialisation, for which there is not enough space here) that the specialisation fails, as the two predicates

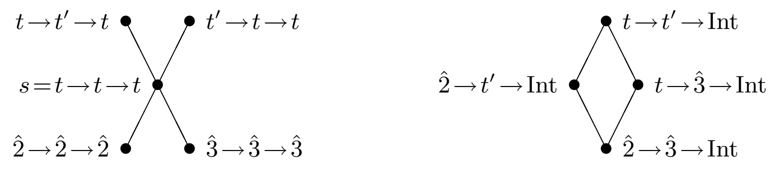
$$\begin{array}{l}h^u_g : \mathrm{IsMG}\ (\mathrm{IsInt}\ t \Rightarrow t \to t \to (\mathrm{Int}, \mathrm{Int}))\ s,\\ h^\ell_f : \mathrm{IsMG}\ s\ (\hat{1} \to \hat{2} \to (\mathrm{Int}, \mathrm{Int}))\end{array}$$

can not be satisfied sharing the same context. This shows that we must be careful when introducing solving into the specialisation process, particularly looking at which scheme variables appears out of the context (in the type).

With the motivation of the previous example, we add constraint solving to the specialisation process in a similar way we did with simplification, with the additional condition about free variables of not appearing in the type scheme nor in the type assignment. The rule (SOLV) is added to system P:

$$(\text{SOLV})\quad \frac{\Delta_1 \mid \Gamma \vdash_P e : \tau\ \hookrightarrow\ e' : \sigma \qquad S{:}T;C \mid \Delta_1 + \Delta' \rhd_{\mathrm{FTV}(\Gamma,\sigma)} \Delta_2}{\Delta_2 \mid TS\Gamma \vdash_P e : \tau\ \hookrightarrow\ C[e'] : TS\sigma}$$

In this point it is possible to illustrate the reason of the occurrence of the predicate assignment Δ' in the solving relation. It stands for expressing the need of forcing the creation of new predicates to make two type schemes comparable by $\geq$: when an IsMG predicate is to be satisfied, some conditions are forced to be true, as shown in the next example.



$t\to t'\to t$
$t'\to t\to t$
$s = t\to t\to t$
$\hat{2}\to\hat{2}\to\hat{2}$
$\hat{3}\to\hat{3}\to\hat{3}$
$t\to t'\to \mathrm{Int}$
$\hat{2}\to t'\to \mathrm{Int}$
$t\to \hat{3}\to \mathrm{Int}$
$\hat{2}\to \hat{3}\to \mathrm{Int}$

IsMG $(\forall t, t'.\mathrm{IsInt}\ t, \mathrm{IsInt}\ t' \Rightarrow t\to t'\to t)\ s$
IsMG $(\forall t, t'.\mathrm{IsInt}\ t, \mathrm{IsInt}\ t' \Rightarrow t'\to t\to t)\ s$
IsMG $s\ (\hat{2}\to\hat{2}\to\hat{2})$
IsMG $s\ (\hat{3}\to\hat{3}\to\hat{3})$
(a) Unique solution

IsMG $(\forall t, t'.\mathrm{IsInt}\ t, \mathrm{IsInt}\ t' \Rightarrow$
$t \to t' \to \mathrm{Int})\ s$
IsMG $s\ (\hat{2}\to\hat{3}\to \mathrm{Int})$
(b) Multiple solutions

Fig. 3. Lattices of solutions

Example 12. Consider the following principal type specialisation:

$$\begin{array}{l}\Delta \vdash \mathbf{let}^D\ \ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ (x +^S 2^S)) \\ \quad \mathbf{in\ let}^D\ \ g = \mathbf{poly}\ (\lambda^D y.\mathbf{spec}\ f\ @^D\ (y +^S 1^S))\ \mathbf{in\ spec}\ g\ @^D\ 2^S : \mathrm{Int}^D \\ \hookrightarrow \mathbf{let}\ f = h^u_f[\Lambda h_x.\Lambda h_{x''}.\lambda x.h_{x''}] \\ \quad \mathbf{in\ let}\ g = h^u_g[\Lambda h_y.\Lambda h^\ell_f.\Lambda h_{y'}.\lambda y.h^\ell_f[f]@\bullet]\ \mathbf{in}\ h^\ell_g[g]@\bullet : \mathrm{Int}\end{array}$$

with Δ being:

$$\begin{array}{l}h^u_g : \mathrm{IsMG}\ (\forall t_y, t_{y'}.\mathrm{IsInt}\ t_y, \mathrm{IsMG}\ s_f\ (t_{y'} \to \mathrm{Int}), t_{y'} := t_y + \hat{1} \Rightarrow t_y \to \mathrm{Int})\ s_g, \\ h^\ell_g : \mathrm{IsMG}\ s_g\ (\hat{2} \to \mathrm{Int}), \\ h^u_f : \mathrm{IsMG}\ (\forall t_x, t_{x''}.\mathrm{IsInt}\ t_x, t_{x''} := t_x + \hat{2} \Rightarrow t_x \to \mathrm{Int})\ s_f,\end{array}$$

If type scheme variables s_g y s_f are about to be solved (which is possible w.r.t. the condition shown in example 11), it can be seen that s_g is the first choice, and according to the next sections it has one upper and one lower bound. Choosing the upper bound as solution, the IsMG including s_f must be added to the context to ensure that the upper bound is actually more general than the lower bound. This solving gives

$$\begin{array}{l}\mathbf{let}\ f = h^u_f[\Lambda h_x, h_{x''}.\lambda x.h_{x''}] \\ \ \mathbf{in\ let}\ g = \Lambda h_y, h^\ell_f, h_{y'}.\lambda y.h^\ell_f[f]@\bullet\ \mathbf{in}\ g(\!(2)\!)(\!(h^*)\!)(\!(3)\!)@\bullet : \mathrm{Int}\end{array}$$

(where h^* is a fresh evidence variable) and the assignment

$$\begin{array}{l}h^u_f : \mathrm{IsMG}\ (\forall t_x, t_{x''}.\mathrm{IsInt}\ t_x, t_{x''} := t_x + \hat{2} \Rightarrow t_x \to \mathrm{Int})\ s_f, \\ h^* : \mathrm{IsMG}\ s_f\ (3 \to \mathrm{Int})\end{array}$$

The last predicate, labelled with the new variable h^*, constitute the predicate assignment Δ' of (SOLV) rule, whose introduction we wished to illustrate. We finish the example solving (and then simplifying) s_f by choosing the value given by its upper bound. The final result is (the context is now empty):

$$\begin{array}{l}\mathbf{let}\ f = \Lambda h_x, h_{x''}.\lambda x.h_{x''} \\ \ \mathbf{in\ let}\ g = \Lambda h_y, h^\ell_f, h_{y'}.\lambda y.h^\ell_f[f]@\bullet\ \mathbf{in}\ g(\!(2)\!)(\!([\,](\!(3)\!)(\!(5)\!))\!)(\!(3)\!)@\bullet : \mathrm{Int}\end{array}$$

It is interesting to observe that, knowing all the definition and use places of f and g, an equivalent term could be constructed:

$$\mathbf{let}\ f = \lambda x.5\ \mathbf{in}\ \mathbf{let}\ g = \lambda y.f@\bullet\ \mathbf{in}\ g@\bullet : \mathrm{Int}$$

This term is much simpler than the previous one, where evidence abstraction and application produces a substantial overhead, had it to be executed. It has also the advantage of belonging to the same residual language of the original type specialisation of [4] (furthermore, it is exactly the term there obtained).

Like the case of simplification, solving is consistent w.r.t. system P:

Theorem 13. *If* $\Delta_1 \mid \Gamma \vdash_P e : \tau \hookrightarrow e' : \sigma$ *and* $S{:}T; C \mid \Delta_1 + \Delta' \rhd_{\mathrm{FTV}(\Gamma,\sigma)} \Delta_2$ *then* $\Delta_2 \mid TS\Gamma \vdash_P e : \tau \hookrightarrow C[e'] : TS\sigma$*.*

4.3 Finding Solutions

In this section we develop an algorithm to perform constraint solving for the language presented. It will be an heuristic, that will try to eliminate as many predicates as possible. Our goal is just to make a very simple algorithm so as to illustrate the concepts studied, but not to make a real constraint solving algorithm – that job is outside the limits of this work. Some conditions to be considered are studied in the next subsections, finishing with the implementation of the algorithm in section 4.3.

The search space While simplification works on any kind of predicate, the solving process have to decide the values of variables appearing in specific predicates, which represent decisions deferred at specialisation time. In our language they are symbolized exclusively by IsMG predicates, generated by the use of (POLY) and (SPEC) specialisation rules. (Extensions of the language like static functions produces similar predicates w.r.t. solving)

Working specifically for the residual expressions coming from specialising our source language, it results that the only IsMG predicates generated are of the form IsMG $\sigma\ s$ or IsMG $s\ \sigma$, where $s \notin \mathrm{FV}(\sigma)$.

As the ordering defined by conversions on type schemes can be viewed as a lattice, the problem of solving IsMG predicates (which are just the extension of $\geq$ to type scheme variables) can take advantage of this feature. An example of the representation of type schemes in the lattice and the distinct possible solutions for IsMG predicates, is found in figure 3.

In this way we shall talk, when trying to find the solution for a scheme variable, about *upper* and *lower bounds*. Our heuristic for constraint solving will search for the values that are inside the given bounds (e.g., the greatest lower bound of upper bounds), for which a necessary condition will be to have found all the possible bounds for the variable being solved (example 11 shows what could happen if this condition were not observed).

Multiple solutions In general, there can be many possible solutions for the same variable. In presence of several predicates of the form IsMG σ^u s and IsMG s σ^ℓ, any type scheme less than every upper bound σ^u and greater than the lower bounds σ^ℓ can be used as a solution (provided s does not appear neither in the type or the assignment – as shown in example 11 – nor in any other kind of predicate).

We will choose the greatest possible solutions. It remains to be studied how this choice affects the final result. We think that the different solutions will produce the same term, after applying a transformation to eliminate as much evidence as possible.

Selection of variables Another decision to make when solving a predicate set is which variables are chosen to be solved. If there are solutions to all variables, solving then at once would be enough, but practical implementations will be based on picking them one by one, or by groups. The following lemma allows us to calculate solutions sequentially, and then compose them:

Lemma 14. *Solutions sequentially obtained can be composed. That is, provided that* $S_2 \leftrightarrow_{S\Delta_1,\Delta'} T_1$ *then* $S_1{:}T_1;C_1 \mid \Delta_1 + \Delta' \rhd_V \Delta_2$ *and* $S_2{:}T_2;C_2 \mid \Delta_2 + \Delta'' \rhd_V \Delta_3$ *implies* $S_2S_1{:}T_2T_1;C_2 \circ C_1 \mid \Delta_1 + (S_2\Delta', \Delta'') \rhd_V \Delta_3$

The compatibility condition may seem too strong, but as we shall see, it holds when we are sequentially solving variables with the algorithm described in the next section. So our algorithm will solve single variables in each step.

Algorithm As described in 4.3, type schemes can be viewed in a lattice sorted by the 'more general' relation $\geq$. As stated in 4.3, we know that any value inside the bounds given for a scheme variable s can be selected as a solution.

In this way we shall describe an heuristic for solving constraints produced by specialisations of our source language based on this ordering.

Given a predicate assignment Δ, which can be partitioned in $\Delta^u, \Delta^\ell, \Delta_s$, where Δ^u are all upper bounds of the form h_i^u : IsMG σ_i^u s, Δ^ℓ the lower bounds h_j^ℓ : IsMG σ_j^ℓ s, and $s \notin \mathrm{FTV}(\sigma_i^u, \sigma_j^\ell, \Delta_s)$ (i.e. s is ready to be solved), we proceed one step of the algorithm in the following way:

1. Let $\sigma = \mathrm{glb}(\sigma_i^u)$ be the greatest lower bound of upper bounds (there is no solution if it is not defined), and C_i^u conversions such that $C_i^u : \sigma_i^u \geq \sigma$ (all of them must exist, because of the way we are chosing σ).
2. Compute a conversion C_j^ℓ and a predicate assignment Δ_j^ℓ for every lower bound, such that $C_j^\ell : \sigma \geq (\Delta_j^\ell \mid \sigma_j^\ell)$. The predicates Δ_j^ℓ are needed since we must enforce σ to be greater than any lower bound, and this is can be done constraining them with extra conditions, taken from the glb.
3. Let Δ_f be the union of all Δ_j^ℓ (the forced predicates), and S the substitution $[\sigma/s]$. Simplify the result using the function defined in section 3.3: $(\Delta', T, C) = \textit{simplify}(\Delta_s, \Delta_f)$.

4. The result is the tuple $([\sigma/s], T, C \circ (h_i^u \leftarrow C_i'^u) \circ (h_j^\ell \leftarrow C_j'^\ell), \Delta_f, \Delta')$, returning the solution, the simplifying solution, a conversion made of the composition of the evidence replacements for every upper and lower bounds with the simplifying conversion, the forced ("new") predicates and the solved and simplified predicate assignment.

Correctness of this algorithm w.r.t. the specification of solving relations is shown in the following lemma:

Lemma 15. *If one step of the algorithm obtains $(S, T, C, \Delta_f, \Delta')$ for an assignment Δ and scheme variable s (with $s \notin V$) then $S : T; C \mid \Delta + \Delta_f \rhd_V \Delta'$.*

We can see that the variable s is removed from the term, and no variables are created in the process. This guarantees termination of the full algorithm, defined as just the iteration of the step described above. Moreover, since compositions of solutions are also solutions (provided compatibility of substitution, which holds for those picked in the algorithm), the algorithm gives a solution for as many variables as can be solved with this criterion.

This algorithm solves only those constraints generated for the simple, pruned language we work with. It fails (it does nothing) if given an assignment which can not be split in the way described by step 1; this is a strong condition: for instance, specialising recursive definitions produces IsMG predicates where the same variable appears on both bounds, so they can not be solved by this method. Solving those assignments is subject of future work.

5 Related and Future Work

This work is entirely based on a fairly reduced source language. We think that this language is still appropriate, introducing the problems we aimed to solve.

The main line of future work is to develop heuristics to deal with the results of specialising more interesting constructs like recursion or static functions. That work is not just an instantiation of the algorithms to our formalization, but it is a complex task by itself, given by the fact that constraint solving is the point where principal type specialisation can hide the source of non-termination of traditional partial evaluators. This is why we talk about heuristics instead of algorithms, since it is not sure to obtain a procedure for effectively solving every possible set of predicates. However, this task can bring surprisingly innovative ideas to the field: for example, the specialisation of lazy functional languages can be represented, as argued in [11].

Another interesting task is the study of the relation between the terms resulting from solving predicates (full of evidence expressions like evidence abstraction and application) and those of the original specialisation of Hughes [4]. We are studying an implementation of the 'evidence elimination' postprocessing phase (the task of eliminating evidence abstraction and applications from specialised terms) for a selected class of terms, as an instance of constraint solving. This can be done by carefully choosing the solutions of variables, taking into account every upper and lower bounds present in the term. This yields, in the best case, the same terms obtained by the original specialiser.

6 Conclusions

We presented a formalization of the processes of simplification and constraint solving as part of the principal type specialisation process, a novel approach to type specialisation. The main contribution is to provide a formal language in which to study the behavior of types and terms under this processes, giving a basis to the formal study of constraint solving of specialising new, and more interesting, features, such as recursion and polymorphism.

We also gave an instantiation of our abstract study by specifying a simplification relation with a system of rules, proved soundness w.r.t. their requirements, and a simple implementation of an heuristic for constraint solving that is relatively weak but strong enough to solve the cases generated as specialisations of the constructs in our source language. This can be seen as a starting place to implement more powerful constraint solving algorithms, for which ours is just a minimalist example.

References

1. Jones, N.D., Gomard, C.K., Sestoft, P.: 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`.
2. Jones, N.D.: Challenging problems in partial evaluation and mixed computation, North Holland, IFIP World Congress Proceedings, Elsevier Science Publishers B.V. (1988) 1–14
3. Danvy, O., Martínez López, P.E.: Tagging, encoding and Jones optimality. In Degano, ed.: Programming Languages and Systems. 12th European Symposium on Programming, ESOP 2003. Volume 2618 of Lecture Notes in Computer Science. (2003) 335–347
4. Hughes, J.: Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Danvy, O., Glück, R., Thiemann, P., eds.: Selected papers of the International Seminar “Partial Evaluation”. Volume 1110 of Lecture Notes in Computer Science., Dagstuhl, Germany, Springer-Verlag, Heidelberg, Germany (1996) 183–215
5. Martínez López, P.E., Hughes, J.: Principal type specialisation. In: Proceedings of Asian Symposium on Partial Evaluation and Semantic-Based Program Manipulation (ASIA-PEPM), ACM Press (2002)
6. Jones, M.P.: Qualified Types: Theory and Practice. Distinguished Dissertations in Computer Science. Cambridge University Press (1994)
7. Berry, G., Boudol, G.: The chemical abstract machine. In: Theoretical Computer Science. Volume 96. (1992) 217–248
8. Jones, M.P.: Simplifying and improving qualified types. Technical report, Yale University (1994) YALEU/DCS/RR-1040.
9. Badenes, H.: Cómo eliminar evidencia resolviendo restricciones para producir automáticamente programas tipados. (2003) Undergraduate thesis, Universidad Nacional de La Plata (In preparation).
10. Aiken, A.: Introduction to set constraint-based program analysis. In: Science of Computer Programming. Volume 35. (1999) 79–111
11. Martínez López, P.E.: Type Specialisation of Polymorphic Languages. PhD thesis, University of Buenos Aires (2003) (In preparation).