The Correctness of Type Specialisation

John Hughes

Chalmers University of Technology, S-41296 Göteborg.
rjmh@cs.chalmers.se, www.cs.chalmers/~rjmh.

Abstract. Type specialisation, like partial evaluation, is an approach to specialising programs. But type specialisation works in a very different way, using a form of type inference. Previous articles have described the method and demonstrated its power as a program transformation, but its correctness has not previously been addressed. Indeed, it is not even clear what correctness should mean: type specialisation transforms programs to others with different types, so clearly cannot preserve semantics in the usual sense.
In this paper we explain why finding a correctness proof was difficult, we motivate a correctness condition, and we prove that type specialisation satisfies it. Perhaps unsurprisingly, type-theoretic methods turned out to crack the nut.

1 Introduction

Type specialisation, like partial evaluation, is an approach to specialising programs [13]. While partial evaluation focusses on specialising the control structures of a program, type specialisation focusses on transforming the datatypes. A type specialiser can produce programs operating on quite different types from the source program, and as a result achieve very strong specialisations. Earlier papers contain many illustrations of the power of the method [10,9,12,11,4].

However, these earlier papers do not address the *correctness* of the method: are the programs which type specialisation produces equivalent to those they are derived from? This question is harder to answer for type specialisation than for partial evaluation for two reasons. Firstly, since the type specialiser changes types, it is not even clear what 'equivalent' means. Secondly, for the most part, a partial evaluator applies a sequence of small semantics preserving transformations whose correctness is obvious, but the type specialiser is described by axiomatising the relation between source and residual programs in one go. Thus there is more scope for error. Indeed, it transpires that the type specialiser does *not* preserve semantics, but we are able to prove a weaker result which is 'good enough'.

In this paper, we present our proof of correctness. We shall begin by reviewing type specialisation, and explaining the problems which foiled our earlier attempts to find a proof. Then we explain what we actually prove, which is an analogue of subject reduction. Finally, we will present some of the cases of the proof in detail.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 215–229, 2000.

2 What is Type Specialisation?

Type specialisation transforms a typed source program into a typed residual program, and in constrast to partial evaluation, types play a major rôle during the transformation itself. Both source and residual programs are simply typed, but they are expressed in different languages, and their types are used for different purposes. In Figure 1 we specify the syntax of terms and types for a small language we will study first.

$$\begin{array}{rcl} e & ::= & n \mid e + e \\ & \mid & \underline{\mathbf{lift}}\ e \\ & \mid & x \mid \underline{\lambda} x.e \mid e \underline{@} e \\ & \mid & \underline{\mathbf{fix}}\ e \end{array} \qquad \begin{array}{rcl} e' & ::= & \bullet \\ & \mid & n \\ & \mid & x \mid \lambda x.e' \mid e'\ e' \\ & \mid & \mathbf{fix}\ e' \end{array}$$

$$\begin{array}{rcl} \tau & ::= & \mathbf{int} \\ & \mid & \underline{\mathbf{int}} \\ & \mid & \tau \underline{\rightarrow} \tau \end{array} \qquad \begin{array}{rcl} \tau' & ::= & n \\ & \mid & \mathbf{int} \\ & \mid & \tau' \rightarrow \tau' \end{array}$$

Fig. 1. Source and Residual Languages.

The source language is a form of two-level λ-calculus: constructions may come in two forms, static or dynamic, with the dynamic form indicated by underlining. Similarly, types may be either static or dynamic. In the figure, we consider only static integers (constants or additions), dynamic integers (formed by applying $\underline{\textbf{lift}}$ to static ones), and dynamic functions ($\underline{\lambda}$-expressions, dynamic application, and dynamic **fix**). The typing rules for this fragment are given in Figure 2.

$$\Gamma \vdash n : \mathbf{int} \qquad \frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash \underline{\mathbf{lift}}\ e : \underline{\mathbf{int}}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \underline{\rightarrow} \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \underline{@} e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_i : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \underline{\rightarrow} \tau_2} \qquad \frac{\Gamma \vdash e : \tau \underline{\rightarrow} \tau}{\Gamma \vdash \underline{\mathit{fix}}\ e : \tau}$$

$$\Gamma, x : \tau \vdash x : \tau$$

Fig. 2. Source Typing Rules.

There are two subtleties here, however. Firstly, in contrast to other two-level λ-calculi, we do not restrict the formation of two level types in any way. For example, we allow dynamic functions to take static values as arguments, and return static results, which is forbidden in the context of partial evaluation.

The reason is simply that the type specialiser is able to specialise such programs, while partial evaluators are not. Intuitions from other specialisers lead one astray here therefore: the reason that no restrictions on type formation are stated is not that I have forgotten them, but that there are indeed no restrictions.

Secondly, we interpret the syntax of types co-inductively. That is, types may be *infinite* expressions conforming to this syntax. This is the way in which we handle recursive types: they are represented as their infinite unfolding, and no special construction for type recursion is required. This is particularly useful for residual types, since it allows the specialiser to synthesize recursive types freely. While recursive types correspond only to *regular* infinite types, we need no assumption of regularity in the proofs which follow. Recursive types are of little use in the fragment in the Figure, but when we later extend the language we consider they will of course play their usual useful rôle.

The residual language is also a form of simply typed λ-calculus, but with a rich type system in which types carry static information. Thus there is a residual type n for every integer n; a static integer expression in the source language specialises to a residual expression with such a type. All static information is expressed via residual types, and as a result need not be present in residual terms. This explains the residual term $\bullet$, which stands for 'no value': we can specialise $2+2$ for example to $\bullet$, since the residual type (4) already tells us all we need to know about the result. Type specialisation produces many residual expressions of this sort, but they are easy to remove in a post-processor we call the 'void eraser'. The typing rules for residual terms are given in Figure 3.

$$\Gamma \vdash \bullet : n \qquad\qquad \Gamma \vdash n : \mathbf{int} \qquad\qquad \Gamma, x : \tau' \vdash x : \tau'$$

$$\frac{\Gamma, x : \tau_1' \vdash e : \tau_2'}{\Gamma \vdash \lambda x.e : \tau_1' \to \tau_2'} \qquad \frac{\Gamma \vdash e_1' : \tau_1' \to \tau_2' \quad \Gamma \vdash e_2' : \tau_1'}{\Gamma \vdash e_1'\ e_2' : \tau_2'} \qquad \frac{\Gamma \vdash e' : \tau' \to \tau'}{\Gamma \vdash \mathbf{fix}\ e' : \tau'}$$

Fig. 3. Residual Typing Rules.

Type specialisation is specified via a set of specialisation rules, analogous to typing rules. Specialisation rules let us infer specialisation judgements, of the form

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

meaning that source expression e of type τ specialises to residual expression e' of type τ'. The context Γ contains assumptions about source variables, of the form

$$x : \tau \hookrightarrow e' : \tau'$$

Notice that variables may specialise to any residual expression; they do not have to specialise to variables.

$$\Gamma \vdash n : \textbf{int} \hookrightarrow \bullet : n$$

$$\frac{\Gamma \vdash e_i : \textbf{int} \hookrightarrow e'_i : n_i \qquad n = n_1 + n_2}{\Gamma \vdash e_1 + e_2 : \textbf{int} \hookrightarrow \bullet : n}$$

$$\frac{\Gamma \vdash e : \textbf{int} \hookrightarrow e' : n}{\Gamma \vdash \underline{\textbf{lift}}\ e : \underline{\textbf{int}} \hookrightarrow n : \textbf{int}}$$

$$\Gamma, x : \tau \hookrightarrow e' : \tau' \vdash x : \tau \hookrightarrow e' : \tau'$$

$$\frac{\Gamma, x : \tau_1 \hookrightarrow x' : \tau'_1 \vdash e : \tau_2 \hookrightarrow e' : \tau'_2}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \underline{\to} \tau_2 \hookrightarrow \lambda x'.e' : \tau'_1 \to \tau'_2} \quad x' \notin FV(\Gamma)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \underline{\to} \tau_2 \hookrightarrow e'_1 : \tau'_1 \to \tau'_2 \qquad \Gamma \vdash e_2 : \tau_1 \hookrightarrow e'_2 : \tau'_1}{\Gamma \vdash e_1 \underline{@} e_2 : \tau_2 \hookrightarrow e'_1\ e'_2 : \tau'_2}$$

$$\frac{\Gamma \vdash e : \tau \underline{\to} \tau \hookrightarrow e' : \tau' \to \tau'}{\Gamma \vdash \underline{\textbf{fix}}\ e : \tau \hookrightarrow \textbf{fix}\ e' : \tau'}$$

Fig. 4. Specialisation Rules.

The specialisation rules for the fragment we are considering here are given in Figure 4. Using these rules we can conclude, for example, that

$$\vdash (\underline{\lambda} x.\underline{\textbf{lift}}\ (x+1))\underline{@}2 : \underline{\textbf{int}} \hookrightarrow (\lambda x'.3)\ \bullet : \textbf{int}$$

The $2 : \textbf{int}$ specialises to $\bullet : 2$, which forces the type of x' to be 2. Consequently $x + 1 : \textbf{int}$ specialises to $\bullet : 3$, and the $\underline{\textbf{lift}}$ moves this static information back into the term, specialising to $3 : \textbf{int}$. Void erasure in this case elides both $\bullet$ and $\lambda x'$, resulting in just 3 as the final specialised program.

Note that the residual type system is more restrictive than the source one, so that well-typed source programs may fail to specialise. For example, the term

$$(\underline{\lambda} f.f\underline{@}2 + f\underline{@}3)\underline{@}(\underline{\lambda} x.x + 1)$$

cannot be specialised, because x would need to be assigned both residual types 2 and 3. This is perfectly natural: when we introduce the possibility to specialise types, we also introduce the possibility to do so inconsistently at different points in the program.

Using types to carry static information enables us to specialise more programs than a partial evaluator can. For example,

$$\vdash (\underline{\lambda} f.f\underline{@}2)\underline{@}(\underline{\lambda} x.\underline{\textbf{lift}}\ (x+1)) : \underline{\textbf{int}} \hookrightarrow (\lambda f'.f'\ \bullet)\ (\lambda x'.3) : \textbf{int}$$

where x' must have type 2 to match the call of f', and so the body of f' specialises to 3. Here we can specialise the body of $\underline{\lambda} x.\underline{\textbf{lift}}\ (x+1)$, even though it does not appear in an application. A partial evaluator, such as λ-MIX [6] or Similix [2], would need to contract at least the outer β-redex in order to propagate a static argument to x, but since this is a dynamic β-redex then this is forbidden; this program is not well annotated for partial evaluation, but causes the type specialiser no problems. In larger programs where it is important not to unfold certain function calls, then this capability gives the type specialiser substantially more power.

There is much more to the type specialiser than this, but we will introduce further features later, along with their proofs of correctness.

3 Why is Correctness Difficult?

Of course, we would like to know that specialisation does not change the semantics of programs; residual programs should be equivalent to the source programs they were derived from. Yet we cannot hope to prove this for the type specialiser. The very essence of the type specialiser is that it changes types. The source and residual programs in general have quite different types, and so they lie in different semantic domains: we certainly cannot expect them to be equal. For example, 42 specialises to $\bullet$, and of course these are different.

However, we note that dynamic type constructors always specialise to one-level versions of themselves — in our fragment this refers to $\underline{\mathbf{int}}$ and $\underline{\rightarrow}$. Thus, if the type of an expression involves only these constructors, then it will specialise to a residual expression with an isomorphic type. Thus we might hope to prove equivalence in this case.

Unfortunately, it doesn't hold. Consider the source term $\underline{\mathbf{lift}}\ (\underline{\mathbf{fix}}\ (\underline{\lambda} x.x))$, which clearly denotes $\perp$. If we assume $x : \mathbf{int} \hookrightarrow x' : 42$, then we can specialise $\underline{\lambda} x.x$ to $\lambda x'.x' : 42 \rightarrow 42$, and so specialise the fixpoint to a term with type 42. Now the rule for $\underline{\mathbf{lift}}$ lets us specialise the entire term to $42 : \mathbf{int}$, which is clearly not equivalent to the source expression. In this case the implemented specialiser would not actually choose this specialisation, but we can force it to exhibit similar behaviour by supplying slightly more complex terms. For example,

$$\underline{\mathbf{lift}}\ (\underline{\mathbf{fix}}\ (\underline{\lambda} x.\underline{\mathbf{if}}\ \underline{\mathbf{true}}\ \underline{\mathbf{then}}\ x\ \underline{\mathbf{else}}\ 42))$$

specialises to 42, but denotes $\perp$.

Instead of equivalence, therefore, we will aim to prove that the source term approximates the residual one. That is, the type specialiser may transform non-terminating programs into terminating ones, but it will never transform a terminating program into one which produces a different answer. Many program transformations behave similarly, so we will consider this weaker correctness property to be acceptable.

4 Outline of the Proof

Since type specialisation is modelled closely on type inference, it is perhaps not so surprising that type theoretic methods turn out to be useful. We will prove the correctness of the specialiser by showing a kind of subject reduction result. We will define source and residual reduction relations, both of which we write as $\rightarrow$, and then we will prove

> **Theorem** (Simulation). If $\Gamma \vdash e_1 : \tau \hookrightarrow e_1' : \tau'$ and $e_1 \rightarrow e_2$, then there exists an e_2' such that $\Gamma \vdash e_2 : \tau \hookrightarrow e_2' : \tau'$ and $e_1' \rightarrow^* e_2'$.

By this theorem we know that if e_1 eventually reduces to a value, then e_1' reduces to the specialisation of a value. By

> **Lemma** (Value Specialisation). If $\Gamma \vdash v : \tau \hookrightarrow e' : \tau'$ (where v is a source value), then e' is a residual value.

then the following correctness theorem follows:

> **Theorem** (Correctness). If $\Gamma \vdash e : \tau \hookrightarrow e' : \tau'$ and e reduces to a value, then so does e'.

In order to prove the Simulation theorem, then we will need two lemmata about substitution — two, because we have two kinds of variables, and therefore two kinds of substitution. The lemma for source substitution is

> **Lemma** (Source Substitution). If $\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_1' : \tau_1'$ and $\Gamma, x : \tau_1 \hookrightarrow e_1' : \tau_1' \vdash e_2 : \tau_2 \hookrightarrow e_2' : \tau_2'$, then $\Gamma \vdash e_2[e_1/x] : \tau_2 \hookrightarrow e_2' : \tau_2'$.

No substitution is required into the residual term, because specialisation itself substitutes e_1' for x.

The residual subsitution lemma is even simpler.

> **Lemma** (Residual Substitution). Let θ be a substition of residual terms for the residual variables occurring free in Γ. If $\Gamma \vdash e : \tau \hookrightarrow e' : \tau'$, then $\Gamma\theta \vdash e : \tau \hookrightarrow e'\theta : \tau'$.

We prove both these lemmata, and the Simulation theorem, by induction over the structure of source terms. In the next section we present the proofs for the fragment we are currently consideration, and then in later sections we show the cases for extensions to this fragment.

5 The Correctness of the Fragment

Before we go further we must define reduction relations for the source and target languages. We do so in Figure 5; the reduction relations are the smallest congruences satisfying the stated properties. By a *value* we mean a closed weak head normal form: the values in the source language take the form n, **lift** n, or $\lambda x.e$, while the values in the residual language take the form $\bullet$, n or $\lambda\overline{x}.\overline{e'}$. The $\overline{\text{V}}$alue Specialisation lemma now follows directly, by applying the appropriate specialisation rule to each form of source value. We now prove the substitution lemmata and the Simulation theorem in turn.

$$\begin{array}{lll@{\qquad\qquad}ll}
n_1 + n_2 & \to n & \text{if } n = n_1 + n_2 & & \\
(\underline{\lambda} x.e_1)\underline{@}e_2 & \to e_1[e_2/x] & & (\lambda x.e_1')\ e_2' & \to e_1'[e_2'/x] \\
\underline{\mathbf{fix}}\ e & \to e\underline{@}(\underline{\mathbf{fix}}\ e) & & \mathbf{fix}\ e' & \to e'\ (\mathbf{fix}\ e')
\end{array}$$

Fig. 5. Source and Residual Reduction Rules.

Proof of the Source Substitution Lemma. We are to prove that if $\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_1' : \tau_1'$ and $\Gamma, x : \tau_1 \hookrightarrow e_1' : \tau_1' \vdash e_2 : \tau_2 \hookrightarrow e_2' : \tau_2'$, then $\Gamma \vdash e_2[e_1/x] : \tau_2 \hookrightarrow e_2' : \tau_2'$. The proof is by induction over the syntax of e_2. The only interesting case is that for variables. For the variable x, we must show that

$$\Gamma \vdash x[e_1/x] : \tau_2 \hookrightarrow e_2' : \tau_2'$$

But from the second assumption, we know that

$$\Gamma, x : \tau_1 \hookrightarrow e_1' : \tau_1' \vdash x : \tau_2 \hookrightarrow e_2' : \tau_2'$$

Consulting the specialisation rule for variables, it follows that e_1' and e_2' are the same, as are τ_1 and τ_2, and τ_1' and τ_2'. Since by the first assumption,

$$\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_1' : \tau_1'$$

then the result follows. For other variables, the proof is trivial.

Proof of the Residual Substitution Lemma. We are to prove that if θ is a substition of residual terms for residual variables, and $\Gamma \vdash e : \tau \hookrightarrow e' : \tau'$, then $\Gamma\theta \vdash e : \tau \hookrightarrow e'\theta : \tau'$. Once again the proof is by induction on the syntax of e. We will prove the cases for variables and $\underline{\lambda}$-expressions, since these are the only rules that can introduce residual variables into the residual term.

For a variable x, we assume that $\Gamma \vdash x : \tau \hookrightarrow e' : \tau'$, which by the specialisation rule for variables means that Γ must contain an assumption of the form $x : \tau \hookrightarrow e' : \tau'$. $\Gamma\theta$ therefore contains the assumption $x : \tau \hookrightarrow e'\theta : \tau'$, and it follows that $\Gamma\theta \vdash x : \tau \hookrightarrow e'\theta : \tau'$ as required.

For a $\underline{\lambda}$-expression $\underline{\lambda} x.e$, we know that its specialisation uses the rule

$$\frac{\Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash e : \tau_2 \hookrightarrow e' : \tau_2'}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \underline{\to} \tau_2 \hookrightarrow \lambda x'.e' : \tau_1' \to \tau_2'} \quad x' \notin FV(\Gamma)$$

Since x' is not free in Γ, it cannot be renamed by θ, so we may conclude by the induction hypothesis that

$$\Gamma\theta, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash e : \tau_2 \hookrightarrow e'\theta : \tau_2'$$

Applying the specialisation rule for $\underline{\lambda}$ again, we derive

$$\Gamma\theta \vdash \underline{\lambda} x.e : \tau_1 \underline{\to} \tau_2 \hookrightarrow (\lambda x'.e')\theta : \tau_1' \to \tau_2'$$

as required.

Proof of the Simulation Theorem . We are to prove that if $\Gamma \vdash e_1 : \tau \hookrightarrow e_1' : \tau'$ and $e_1 \rightarrow e_2$, then there exists an e_2' such that $\Gamma \vdash e_2 : \tau \hookrightarrow e_2' : \tau'$ and $e_2 \rightarrow^* e_2'$. This proof is also by induction on the syntax of e_1, and we will present it in some detail.

- Case n. Trivial, since n does not reduce to anything.
- Case $e_1 + e_2$. According to the specialisation rule for +, we have

  $$\frac{\Gamma \vdash e_i : \mathbf{int} \hookrightarrow e_i' : n_i \qquad n = n_1 + n_2}{\Gamma \vdash e_1 + e_2 : \mathbf{int} \hookrightarrow \bullet : n}$$

  Suppose first that e_1 and e_2 are both values. Since

  $$\Gamma \vdash e_1 : \mathbf{int} \hookrightarrow e_1' : n_1$$

  then e_1 must be n_1, and similarly for e_2. It follows that $e_1 + e_2 \rightarrow n$, which specialises to $\bullet : n$. It remains to show that $\bullet \rightarrow^* \bullet$, which it does in zero steps.
  Alternatively, suppose without loss of generality that $e_1 + e_2 \rightarrow e_3 + e_2$ by reducing $e_1 \rightarrow e_3$. Then by the induction hypothesis, there is an e_3' such that $e_1' \rightarrow^* e_3'$ and

  $$\Gamma \vdash e_3 : \mathbf{int} \hookrightarrow e_3' : n_1$$

  Applying the specialisation rule for +, we derive

  $$\Gamma \vdash e_3 + e_2 : \mathbf{int} \hookrightarrow \bullet : n$$

  and it remains only to show $\bullet \rightarrow^* \bullet$ as before.
- Case $\underline{\mathbf{lift}}\ e$. We have $\underline{\mathbf{lift}}\ e \rightarrow \underline{\mathbf{lift}}\ e_0$, and

  $$\frac{\Gamma \vdash e : \mathbf{int} \hookrightarrow e' : n}{\Gamma \vdash \underline{\mathbf{lift}}\ e : \underline{\mathbf{int}} \hookrightarrow n : \mathbf{int}}$$

  We have $e \rightarrow e_0$, and so by the induction hypothesis there is an e_0' such that $e' \rightarrow^* e_0'$ and $\Gamma \vdash e_0 : \mathbf{int} \hookrightarrow e_0' : n$. It follows that

  $$\Gamma \vdash \underline{\mathbf{lift}}\ e_0 : \underline{\mathbf{int}} \hookrightarrow n : \mathbf{int}$$

  and since $n \rightarrow^* n$ then the proof is complete.
- Case x. Trivial since there is no reduction rule for variables.
- Case $\underline{\lambda} x.e$. We have $\underline{\lambda} x.e \rightarrow \underline{\lambda} x.e_0$, and

  $$\frac{\Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash e : \tau_2 \hookrightarrow e' : \tau_2'}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \underline{\rightarrow} \tau_2 \hookrightarrow \lambda x'.e' : \tau_1' \rightarrow \tau_2'} \quad x' \notin FV(\Gamma)$$

  So $e \rightarrow e_0$, and by the induction hypothesis there is an e_0' such that $e' \rightarrow^* e_0'$ and

  $$\Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash e_0 : \tau_2 \hookrightarrow e_0' : \tau_2'$$

  It follows that

  $$\Gamma \vdash \underline{\lambda} x.e_0 : \tau_1 \underline{\rightarrow} \tau_2 \hookrightarrow \lambda x'.e_0' : \tau_1' \rightarrow \tau_2'$$

  and $\lambda x'.e' \rightarrow^* \lambda x'.e_0'$ as required.

- Case $e_1 \underline{@} e_2$. An application can be reduced in three different ways: a reduction may be made inside e_1, or inside e_2, or the application itself may be a β-redex which is reduced. The first two cases are proved in the same way as the $\underline{\lambda}$ case above, so we consider only the third. Suppose therefore that e_1 is $\underline{\lambda} x.e$. Combining the specialisation rules for $\underline{\lambda}$ and $\underline{@}$, we obtain

$$\frac{\dfrac{\Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash e : \tau_2 \hookrightarrow e' : \tau_2'}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \underline{\to} \tau_2 \hookrightarrow \lambda x'.e' : \tau_1' \to \tau_2'} \qquad \Gamma \vdash e_2 : \tau_1 \hookrightarrow e_2' : \tau_1'}{\Gamma \vdash (\underline{\lambda} x.e) \underline{@} e_2 : \tau_2 \hookrightarrow (\lambda x'.e')\ e_2' : \tau_2'}$$

  Substituting e_2' for x' using the Residual Substitution lemma, we know that

$$\Gamma, x : \tau_1 \hookrightarrow e_2' : \tau_1' \vdash e : \tau_2 \hookrightarrow e'[e_2'/x'] : \tau_2'$$

  Now by the Source Substitution lemma, we have

$$\Gamma \vdash e[e_2/x] : \tau_2 \hookrightarrow e'[e_2'/x'] : \tau_2'$$

  Since $(\underline{\lambda} x.e) \underline{@} e_2 \to e[e_2/x]$ and $(\lambda x'.e')\ e_2' \to e'[e_2'/x']$, then the proof of this case is complete.
- Case $\underline{\textbf{fix}}\ e$. This case is similar to application, and is omitted.

This completes the proof of the Simulation theorem for the fragment.

6 Extensions

The tiny language we have considered so far illustrates only the basics of type specialisation: it consists only of dynamic λ-calculus plus one kind of static information. In reality the type specialiser accepts a much richer language. In this section we discuss some of the extensions, and their proofs of correctness.

6.1 Enriching the Dynamic Language

In addition to dynamic function types with dynamic $\underline{\lambda}$-expressions and applications, the type specialiser supports dynamic product types with tuples and selectors, dynamic tagged sum types with constructor application and a $\underline{\textbf{case}}$ expression, dynamic $\underline{\textbf{let}}$ expressions and conditionals. In each case we add a dynamic version of each construct to the source language, and a residual version to the residual language. The new reduction rules in the source and residual language correspond. Each dynamic construct specialises to its corresponding residual construct, with specialised sub-expressions. The substitution lemmata extend easily, and the proofs of the Simulation theorem all take the same form: a reduction in a sub-expression is simulated by reductions in the corresponding residual sub-expression, while a reduction using a new source reduction rule is simulated using the corresponding new residual reduction rule. The proofs are modelled on those for $\underline{\lambda} x.e$ and $e_1 \underline{@} e_2$.

6.2 Static Tagged Sums

One of the most interesting applications of the type specialiser is to remove type tags when specialising an interpreter for a typed language. If such an interpreter represents values using a universal type which is a tagged sum of the differently typed alternatives, then the type specialiser can remove the tags, specialising the universal type to an appropriate representation type at each use. To express this, we must add static tagged sum types to our source language. We extend the syntax of types and expressions as follows, where C is a tag, or 'constructor':

$$\begin{aligned}\tau &::= \Sigma_{i=1}^{n}\ C\ \tau \\ e &::= C\ e \\ &\quad\mid\ \ \mathbf{case}\ e\ \mathbf{of}\{C\ x \to e\}_{i=1}^{n}\ \mathbf{end}\end{aligned}$$

Since the tags are static, the corresponding residual types must record which constructor was actually applied. Thus we extend residual types as follows:

$$\tau' ::= C\ \tau'$$

There is no need to extend the language of residual terms, since application and inspection of static constructors will be specialised away.

The specialisation rule for a constructor application just records the constructor in the residual type,

$$\frac{\Gamma \vdash e : \tau_k \hookrightarrow e' : \tau'_k}{\Gamma \vdash C_k\ e : \Sigma_{i=1}^{n}\ C_i\ \tau_i \hookrightarrow e' : C_k\ \tau'_k}$$

while the rule for a **case** expression uses the statically-known constructor to choose the corresponding branch:

$$\frac{\begin{array}{c}\Gamma \vdash e : \Sigma_{i=1}^{n}\ C_i\ \tau_i \hookrightarrow e' : C_k\ \tau'_k \\ \Gamma, x_k : \tau_k \hookrightarrow e' : \tau'_k \vdash e_k : \tau_0 \hookrightarrow e'_k : \tau'_0\end{array}}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\{C_i\ x_i \to e_i\}_{i=1}^{n}\ \mathbf{end} : \tau_0 \hookrightarrow e'_k : \tau'_0}$$

The Source and Residual substitution lemmata extend easily to these cases.

There is one new source reduction rule, namely

$$\mathbf{case}\ C_k\ e\ \mathbf{of}\{C_i\ x_i \to e_i\}_{i=1}^{n}\ \mathbf{end} \to e_k[e/x_k]$$

and one new form of source value: $C\ v$. Notice that in order to prove the Value Specialisation lemma, we must require the argument of the constructor to be evaluated.

We will prove just the case in the Simulation theorem when the new reduction rule is applied. Thus we must prove that if

$$\Gamma \vdash \mathbf{case}\ C_k\ e\ \mathbf{of}\{C_i\ x_i \to e_i\}_{i=1}^{n}\ \mathbf{end} : \tau_0 \hookrightarrow e'_k : \tau'_0$$

then there is an e'' such that $e'_k \to^* e''$ and

$$\Gamma \vdash e_k[e/x_k] : \tau_0 \hookrightarrow e'' : \tau'_0$$

We shall take e'' to be just e'_k, and argue that from the assumption we know that

$$\Gamma, x_k : \tau_k \hookrightarrow e' : \tau'_k \mid\!\!- e_k : \tau_0 \hookrightarrow e'_k : \tau'_0$$

where

$$\Gamma \mid\!\!- e : \tau_k \hookrightarrow e' : \tau'_k$$

By the Source Substitution lemma, it follows that $\Gamma \mid\!\!- e_k[e/x_k] : \tau_0 \hookrightarrow e'_k : \tau'_0$ as required.

6.3 Polyvariance

All interesting program specialisers are *polyvariant*, that is, they can specialise one expression in the source code multiple times. Polyvariance is provided in the type specialiser by extending the source and residual languages as follows:

$$\begin{array}{ll} e ::= \mathbf{poly}\ e \mid \mathbf{spec}\ e & \quad e' ::= (e', \ldots, e') \mid \pi_k\ e' \\ \tau ::= \mathbf{poly}\ \tau & \quad \tau' ::= (\tau', \ldots, \tau') \end{array}$$

The idea is that **poly** e can be specialised to a tuple of specialisations of e, from which **spec** e chooses an element. The residual type of such a tuple records which specialisations it contains. We add reduction rules

$$\mathbf{spec}\ (\mathbf{poly}\ e) \to e \qquad \pi_k\ (e'_1, \ldots, e'_n) \to e'_k$$

and new source values **poly** e, and residual values $(e'_1, \ldots, e'_n)$.

The specialisation rules for these constructions are:

$$\frac{\Gamma \mid\!\!- e : \tau \hookrightarrow e'_i : \tau'_i, i = 1 \ldots n}{\Gamma \mid\!\!- \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow (e'_1, \ldots, e'_n) : (\tau'_1, \ldots, \tau'_n)}$$

$$\frac{\Gamma \mid\!\!- e : \mathbf{poly}\ \tau \hookrightarrow e' : (\tau'_1, \ldots, \tau'_n)}{\Gamma \mid\!\!- \mathbf{spec}\ e : \tau \hookrightarrow \pi_k\ e' : \tau'_k}$$

The proofs of the substitution lemmata and the Simulation theorem go through easily for this extension. For the Simulation theorem, a reduction **poly** $e_1 \to$ **poly** e_2 by $e_1 \to e_2$ can be simulated by reductions in *each* specialisation, while the reduction **spec** (**poly** e) $\to e$ is simulated by $\pi_k\ (e'_1, \ldots, e'_n) \to e'_k$.

6.4 Static Functions

All interesting specialisers provide *static functions*, that is, functions which are unfolded at specialisation time. So, too, does the type specialiser, via static λ-expressions and static applications. The specialisation rule for static λ given in [10] is

$$\begin{array}{l} \{x_i : \tau_i \hookrightarrow e'_i : \tau'_i\} \mid\!\!- \\ \qquad \lambda x.e : \tau_a \to \tau_b \hookrightarrow (e'_1, \ldots, e'_n) : \mathbf{close}\ \{x_i : \tau_i \hookrightarrow \tau' : i\}\ \mathbf{in}\ \lambda x.e \end{array}$$

That is, a static function is represented in the residual program by a tuple of its free variables, and the residual type carries the information needed to unfold β-redexes.

Unfortunately, this specialisation rule violates the theorems we are trying to prove. The Simulation Theorem fails because a reduction under a static λ changes the residual *type* of the specialisation; since residual reduction does not change types, it is impossible for the result of specialisation before the reduction to reduce to the result of specialisation afterwards. The Source Substitution lemma also fails, because a static λ specialises to a tuple with one element per free variable; substituting for one of these variables changes the size of the tuple. The problem here is that type specialisation essentially performs a closure conversion, and closure conversion is not preserved by substitution.

Our solution is to instead consider specialisation of closure-converted programs. Thus we prove the correctness of a variant of the type specialisation previously described. We extend the source and target languages as follows,

$$\begin{array}{ll} e ::= \mathbf{close}\ \{x = e\}^* \ \mathbf{in}\ \lambda x.e \mid e@e & e' ::= (e', \ldots, e') \mid \pi_k\ e' \\ \tau ::= \tau \to \tau & \tau' ::= \mathbf{close}\ \{x : \tau \hookrightarrow \tau'\}^* \ \mathbf{in}\ \lambda x.e \end{array}$$

with the restriction that all the free variables of $\lambda x.e$ must be bound in the associated definitions. Closures and residual tuples are both values.

We add a β reduction rule to the source language,

$$(\mathbf{close}\ \{x_i = e_i\}\ \mathbf{in}\ \lambda x.e)@e_x \to e[e_i/x_i, e_x/x]$$

and a rule for reducing projections to the residual language. Moreover, we forbid reduction of the body of a static λ – the only reductions of closures take place in the bindings $\{x_i = e_i\}$.

The specialisation rules for static closures and applications are

$$\frac{\Gamma \vdash e_i : \tau_i \hookrightarrow e'_i : \tau'_i, i \in 1 \ldots n}{\begin{array}{l}\Gamma \vdash \mathbf{close}\ \{x_i = e_i\}\ \mathbf{in}\ \lambda x.e : \tau_1 \to \tau_2 \hookrightarrow \\ \qquad (e'_1, \ldots, e'_n) : \mathbf{close}\ \{x_i : \tau_i \hookrightarrow \tau'_i\}\ \mathbf{in}\ \lambda x.e\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_f : \tau_x \to \tau_y \hookrightarrow e'_f : \mathbf{close}\ \{x_i : \tau_i \hookrightarrow \tau'_i\}\ \mathbf{in}\ \lambda x.e \\ \Gamma \vdash e_x : \tau_x \hookrightarrow e'_x : \tau'_x \\ \{x_i : \tau_i \hookrightarrow \pi_i\ e'_f : \tau'_i\}, x : \tau_x \hookrightarrow e'_x : \tau'_x \vdash e : \tau_y \hookrightarrow e' : \tau'_y\end{array}}{\Gamma \vdash e_f@e_x : \tau_y \hookrightarrow e' : \tau'_y}$$

With these definitions, the substitution and value specialisation lemmata are easily proved. To prove the Simulation Theorem we must introduce another (easily proved) lemma:

> **Lemma** (Reduction in Context Lemma) Let Γ_2 be obtained from Γ_1 by making a reduction in one of the residual expressions. If $\Gamma_1 \vdash e : \tau \hookrightarrow e'_1 : \tau'$, then there exists e'_2 such that $e'_1 \to^* e'_2$ and $\Gamma_2 \vdash e : \tau \hookrightarrow e'_2 : \tau'$.

The proof of the Simulation Theorem now goes through.

7 Related Work

In 1991, Gomard and Jones described λ-MIX, the first self-applicable partial evaluator for the λ-calculus, which was so simple that much later work was based on it. Gomard proved the correctness of the partial evaluator, that is, that source and residual programs denote the same values [6]. The proof is based on establishing a logical relation between the denotation of a two-level source term, and the denotation of its one-level erasure. λ-MIX was the first partial evaluator whose binding-time analysis was expressed as a type system, and the logical relation is indexed by binding-time types.

Gomard's original proof was somewhat flawed by an informal treatment of fresh name generation. Moggi pointed this out, and gave a rigorous proof based on an alternative semantics for λ-MIX using functor categories [15]. Using related techniques, Filinksi has proved the soundness and completeness [5] of Danvy's type-directed partial evaluation (TDPE) [3].

These proofs have in common that they are based on establishing logical relations between denotational semantics of source and residual terms. This is essentially the approach we first tried to follow to show the correctness of the type specialiser. But since λ-MIX and TDPE do not transform types, the logical relations are simpler to define, and since neither allows recursive binding-time types, the problems they cause with well-definedness of logical relations do not arise. (Recursive types are not really needed in λ-MIX, since dynamic computations are essentially untyped).

Other recent work on the correctness of partial evaluators has focussed on the correctness of binding-time analysis, rather than on specialisation proper.

A closer analogy can be found with other recent work on type-directed transformations. John Hannan and Patrick Hicks have published a series of papers in which they present such transformations of higher order languages, for example [7,8]. Just like type specialisation, these transformations are specified by inference rules, whose judgements relate a source term, a transformed term, and a type in an extended type language specifying how the former should be transformed into the latter. Proofs of correctness are outlined, and are quite similar to our own: source and target languages are given an operational semantics, and there is an analogue of our Simulation Theorem relating the two. Hannan and Hicks also prove that every well-typed source term can be transformed to a target term, which is of course untrue for type specialisation, and that reductions of target terms can be simulated by the corresponding source terms.

8 Discussion and Conclusions

A first attempt to find a proof was based on giving a denotational semantics to source and target languages, and establishing a logical relation indexed by residual types between them. But this foundered when the relation proved to be ill-defined. The problem is that residual types may involve arbitrary type recursion under function arrows. A recursive type leads to a recursively defined logical relation, which only makes sense if the recursive definition has a least fixed point. But since the formation of logical relations on function types is antimonotonic in the left argument, then the usual monotonicity argument that a least fixed point exists does not apply.

It is possible that this approach might succeed even so. We could try to define a metric on relations, and show that the recursive definitions we are interested in are contractive, just as MacQueen, Plotkin and Sethi did to show that recursive types could be modelled by ideals [14]. But this would at best lead to a very technical proof, dependent on the detailed structure of the underlying semantic domains. Instead, we chose to pursue the more operational approach described in this paper.

The proof we have presented is pleasingly simple, and we have some hope that the proof method will be robust to extensions of the type specialiser, not least since similar methods have been used successfully to prove the correctness of other type-directed transformations. The operational approach, inspired by subject reduction, proved to be much easier to carry through the denotationally-based attempt. And of course, it is pleasing to know that type specialisation actually is correct.

The proof does raise other questions, though. For example, earlier papers were vague on whether the intended semantics of the object language was call-by-value or call-by-name. In this paper we explicitly give it a call-by-name semantics. Is type specialisation correct for a call-by-value language? One would hope that a similar proof would go through, but the most obvious idea of restricting β-reduction to β_v redexes does not seem to work easily. Another interesting idea would be to consider call-by-need reduction rules [1]: perhaps one could show thereby that specialisation (of a suitably restricted language) does not duplicate computations.

We have also focussed here on the relationship between source terms and residual *terms* – the dynamic part of the specialisation. Residual *types* in contrast play only a small rôle here. Yet we might also hope to be able to relate them to the source program. Residual types purport to carry static information about the source term they are derived from: in a sense they can be regarded as properties of source terms. For example, if $\vdash f : \mathbf{int} \underline{\rightarrow} \mathbf{int} \hookrightarrow f' : 42 \rightarrow 44$, then we would expect that f maps 42 to 44. Another interesting avenue would be to assign a semantics to residual types as properties, and prove that specialisation produces properties that really hold.

References

1. Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *22'nd Symposium on Principles of Programming Languages*, San Francisco, California, January 1995. ACM Press.
2. A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Selected papers of ESOP '90, the 3rd European Symposium on Programming.

3. O. Danvy. Type-directed partial evaluation. In *Symposium on Principles of Programming Languages*. ACM, jan 1996.
4. D. Dussart, J. Hughes, and P. Thiemann. Type Specialisation for Imperative Languages. In *International Conference on Functional Programming*, pages 204–216, Amsterdam, June 1997. ACM.
5. Andrzej Filinski. A Semantic Account of Type-Directed Partial Evaluation. In *Principles and Practice of Declarative Programming: International Conference PPDP'99*, volume 1702 of *Lecture Notes in Computer Science*, pages 378–395, Paris, France, September 1999. Springer-Verlag.
6. C.K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, April 1992.
7. John Hannan and Patrick Hicks. Higher-Order Arity Raising. In *Proceedings of 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, Baltimore, Maryland, September 1998.
8. John Hannan and Patrick Hicks. Higher-Order UnCurrying. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–10, San Diego, January 1998.
9. J. Hughes. An Introduction to Program Specialisation by Type Inference. In *Functional Programming*. Glasgow University, July 1996. published electronically.
10. J. Hughes. Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *LNCS*, pages 183–215. Springer-Verlag, February 1996.
11. J. Hughes. A Type Specialisation Tutorial. In *DIKU Summer School on Partial Evaluation*, 1998.
12. J. Hughes. Type Specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *1998 Symposium on Partial Evaluation*, volume 30 of *Computing Surveys*, September 1998.
13. N. D. Jones, , C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
14. David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, October/November 1986.
15. Eugenio Moggi. Functor categories and two-level languages. In *FOSSACS*, volume 1378 of *Lecture Notes in Computer Science*, pages 211–225. Springer-Verlag, 1998.