DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Principal Type Specialization for Polyvariant Sum Types

Thesis to obtain the degree of Licentiate in Computer Science

November 24, 2006

Laura Carolina Lowenthal Quastler
laulowen@dc.uba.ar

Director
Dr. Pablo E. Martínez López
fidel@sol.info.unlp.edu.ar

Facultad de Cs. Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Especialización Principal de Tipos para Sumas Polivariantes

Tesis para acceder al grado de Licenciada en Ciencias de la Computación

24 de Noviembre de 2006

Laura Carolina Lowenthal Quastler
laulowen@dc.uba.ar

Director
Dr. Pablo E. Martínez López
fidel@sol.info.unlp.edu.ar

Facultad de Cs. Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Abstract

Program specialization is a form automatic program generation that produces different versions of a given general source program, each of them specialized to particular known data. For example, the recursive function *power*, if the exponent is known to be 3, can be specialized to a more efficient (non-recursive) function $\lambda x.x \cdot x \cdot x$, and similarly for other exponents.

Type specialization [Hughes, 1996b; Hughes, 1996a; Hughes, 1998] is a form of program specialization based on type inference. Both the source program and its type are specialized to a residual program and a residual type. Principal type specialization [Martínez López and Hughes, 2002; Martínez López, 2005] is a detailed formulation to this system based on the theory of Qualified Types [Jones, 1994]. It has the property of producing principal specializations: for each specializable source expression and type, a residual expression and type can be generated such that they are more general than any other valid specialization, and all of them can be obtained from it by a notion of instantiation.

An important notion in any specialization system is that of *polyvariance*, a feature allowing a single source expression to be specialized to many residual results. Polyvariance can be achieved in more than one way; in particular, the original type specialization system [Hughes, 1996b] includes constructs for polyvariant products (where an expression e is specialized to a tuple of expressions $(e'_1, \ldots, e'_n)$) and polyvariant sums (where a tagged expression $In\ e$ is specialized to many tagged expressions $In_1\ e'_1, \ldots, In_n\ e'_n$), the latter also known as *constructor specialization* [Mogensen, 1993].

Principal type specialization was formulated only for a subset of the language presented originally; in particular, polyvariant sum types were not considered. In this thesis, we extend the system with new constructs and rules to specialize polyvariant sums. We prove that our contribution preserves all the properties of the original system, including that of principality, and we incorporate our extension to *PTS*, a prototype implementation of the specializer.

Resumen

La especialización de programas es una forma de generar programas automáticamente, que consiste en producir distintas versiones de un programa fuente general, cada una especializada según datos particulares conocidos. Por ejemplo, la función recursiva *potencia*, sabiendo que el exponente será igual a 3, puede especializarse a una versión residual más eficiente (no recursiva) $\lambda x.x \cdot x \cdot x$, y en forma similar para otros exponentes.

La especialización de tipos [Hughes, 1996b; Hughes, 1996a; Hughes, 1998] es una forma de especialización de programas basada en inferencia de tipos. Tanto el programa fuente como su tipo son especializados a un programa y un tipo residuales. La especialización principal de tipos [Martínez López and Hughes, 2002; Martínez López, 2005] es una formulación detallada de este sistema basada en la teoría de tipos calificados [Jones, 1994]. Tiene la propiedad de generar especializaciones principales: para cada expresión y tipo fuente especializables, se puede producir una expresión y un tipo residuales que son más generales que cualquier otra especialización válida, y tales que todas ellas pueden obtenerse a partir de la primera mediante de una noción de instanciación.

Un concepto importante en todo sistema de especialización es el de *polivarianza*, una característica que permite que una única expresión fuente pueda ser especializada a más de una expresión residual. La polivarianza puede obtenerse de distintas formas; en particular, el sistema de especialización de tipos original [Hughes, 1996b] incluye construcciones para productos polivariantes (donde una única expresión fuente e especializa a una tupla de expresiones $(e'_1, \ldots, e'_n)$) y para sumas polivariantes (donde una expresión etiquetada con un constructor In e especializa a varias expresiones con varios constructores $In_1\ e'_1, \ldots, In_n\ e'_n$). Esta última se conoce también como especialización de constructores [Mogensen, 1993].

La especialización principal de tipos fue formulada sólo para un subconjunto del lenguaje presentado originalmente; en particular, las sumas polivariantes no fueron consideradas. En esta tesis extendemos el sistema con nuevas construcciones y reglas para especializar sumas polivariantes. Demostramos que nuestra contribución preserva todas las propiedades del sistema orignal, incluyendo la de principalidad, e incorporamos nuestra extensión a *PTS*, un prototipo de implementación de este sistema.

Agradecimientos

Gracias Fidel, por proponerme esta tesis, guiarme, acompañarme y aconsejarme durante todo su desarrollo, y especialmente por el esfuerzo e interés puesto en el crecimiento de este trabajo y en el mío como estudiante.

Gracias también a los jurados, Ariel y Eduardo, por el tiempo dedicado a leer y corregir este trabajo.

A mi familia, en especial a mis papás y a Sonia, por estar conmigo durante toda la carrera, alegrarse con el más mínimo de mis éxitos y apoyarme en los fracasos. Por participar de distintas formas, desde el intento de aprenderse el nombre de las materias que cursaba hasta la preocupación por la enorme cantidad de horas de sueño cedidas al estudio.

A todos los amigos que me acompañaron durante estos años, los más viejos y los más nuevos, por escucharme, distraerme, y dejarme compartir este pedazo de mi vida con ellos. A mis compañeros, que pasaron conmigo tantas horas de cursada y trabajo haciéndolas divertidas y más fáciles. Y también a mis amigos que, siendo más avanzados que yo en la carrera, me ofrecieron siempre su orientación y sus consejos, y que en mayor o menor medida influyeron en mí para ser mejor docente y alumna.

Finalmente, agradezco especialmente a Daniel, que vivió conmigo toda la evolución de esta tesis, disfrutándola y sufriéndola como lo hice yo. Gracias por interesarte en los detalles, por darme ánimo en las partes difíciles y por permitirme mis momentos obsesivos de trabajo... pero sólo moderadamente.

Resumen Extendido

Introducción

La especialización de programas es una forma de generar programas automáticamente, que consiste en producir distintas versiones de un programa fuente general, cada una especializada según datos particulares conocidos. Un ejemplo clásico es el da la función *potencia*, que computa x^n

$$\begin{array}{l} \mathit{potencia}\ n\ x\ =\ \textbf{if}\ n\ ==\ 1\ \textbf{then} \\ \qquad\qquad\qquad\qquad x \\ \qquad\qquad\qquad \textbf{else} \\ \qquad\qquad\qquad\qquad x\ *\ \mathit{potencia}\ (n-1)\ x \end{array}$$

El cómputo de esta función involucra comparaciones y llamados recursivos, pero cuando el parámetro n es conocido, puede especializarse a una función no recursiva. Por ejemplo, si sabemos que n vale 3, la función

$$\mathit{potencia}_3\ x\ =\ x\ *\ (x\ *\ x)$$

sería una especialización apropiada, claramente más eficiente que la versión original para calcular cubos. Un resultado similar se puede obtener para distintos exponentes, a partir de la misma función. Al programa original se lo llama *programa fuente*, y a las versiones especializadas, *programas residuales*.

La especialización de programas ha sido encarada de distintas formas, entre las cuales la más difundida es la de *evaluación parcial* [Jones *et al.*, 1993; Consel and Danvy, 1993]. Esta consiste en producir programas residuales a través de reducciones: las subexpresiones con argumentos conocidos se reemplazan por el resultado de su evaluación y se combinan con los cómputos que no pueden hacerse. Es decir, la evaluación parcial trabaja con el *texto* del programa fijando algunos datos de entrada y combinando el cómputo con generación de código para producir un nuevo programa. Los programas generados, cuando se corren con los datos restantes, arrojan el mismo resultado que el programa original corrido con todos los datos.

La especialización de tipos [Hughes, 1996b; Hughes, 1996a; Hughes, 1998] es una forma de especialización de programas basada en inferencia de tipos. Tanto el programa fuente como su tipo son especializados a un programa y un tipo residuales.

En todos los lenguajes tipados, los tipos proveen información acerca de las expresiones. Por ejemplo, cuando una expresión es de tipo *Int*, sabemos que si su evaluación termina arrojará un número entero. Pero si sabemos que la expresión es la constante 11, podemos tener información más refinada con un *tipo que represente la propiedad de ser el entero 11*:

llamemos a este tipo $\hat{1}\hat{1}$. Teniendo toda la información en el tipo, ya no hay necesidad de ejecutar el programa, así que la constante entera puede ser reemplazada por un valor único de tipo $\hat{1}\hat{1}$. En otras palabras, la expresión fuente $11 : \mathit{Int}$ puede especializarse a $\bullet : \hat{1}\hat{1}$.

Llamamos a los tipos y operaciones conocidas *estáticos*, y a los que no lo son, *dinámicos*. Cada subexpresión del programa fuente está etiquetada con los superíndices ^{S} o ^{D} respectivamente. El problema de especializar un programa f con parámetros $x_1, \ldots, x_n$ donde $x_1, \ldots x_k$ son conocidos puede expresarse como la especialización de la expresión

$$f \mathbin{@^S} x_1 \mathbin{@^S} \ldots \mathbin{@^S} x_k \mathbin{@^D} x_{k+1} \mathbin{@^D} \ldots \mathbin{@^D} x_n$$

donde f es una función y $f@x$ representa la aplicación.

Pero la especialización de tipos es un enfoque más general, que permite combinaciones mucho más flexibles de anotaciones estáticas y dinámicas. Los tipos residuales pueden expresar información "parcialmente estática", de manera tal que cierta información estática puede ser asociada con valores dinámicos y propagada con mecanismos análogos a los de inferencia de tipos.

Ejemplo 1 Consideremos la expresión

$$\left(\lambda^D f.\mathbf{lift}\ \left(f \mathbin{@^D} 3^S\right)\right) \mathbin{@^D} \left(\lambda^D x.x +^S 1^S\right) : \mathit{Int}^D$$

Tenemos expresiones lambda y aplicaciones dinámicas, que se transformarán en expresiones lambda y aplicaciones residuales en el programa especializado. El operador **lift** convierte una expresión de tipo entero estático en su valor dinámico: si un entero estático e tiene tipo residual $\hat{n}$, entonces **lift** e especializa a $n : \mathit{Int}$.

Para especializar esta expresión, inferimos el tipo residual de cada una de sus partes. Para empezar, f se aplica a un argumento con tipo residual $\hat{3}$, así que debe tener tipo residual $\hat{3} \to \tau$ para algún τ. Ahora bien, este también debe ser el tipo de $\left(\lambda^D x.x +^S 1^S\right)$, con lo que x debe ser de tipo residual $\hat{3}$. Entonces $\left(x +^S 1^S\right)$ debe tener tipo $\hat{4}$; conlcuimos que τ es $\hat{4}$, f es de tipo $\hat{3} \to \hat{4}$ y $f \mathbin{@^D} 3$ es de tipo $\hat{4}$. Finalmente, la operación **lift** puede especializarse a 4, y el resultado es:

$$(\lambda f.4) \mathbin{@} (\lambda x.\bullet) : \mathit{Int}$$

Hemos obtenido el valor 4 en el código residual sin desdoblar ninguna de las funciones. $\diamond$

Hasta aquí hemos descripto brevemente la especialización de tipos tal como fue presentada originalmente por Hughes. La especialización principal de tipos [Martínez López and Hughes, 2002; Martínez López, 2005] es una formulación detallada de este sistema basada en la teoría de tipos calificados [Jones, 1994]. Tiene la propiedad de generar especializaciones principales: para cada expresión y tipo fuente especializables, se pueden producir una expresión y un tipo residuales que son más generales que cualquier otra especialización válida, y tales que todas ellas pueden obtenerse a partir de la primera mediante de una noción de instanciación.

Un concepto importante en todo sistema de especialización es el de *polivarianza*, una característica que permite que una única expresión fuente pueda ser especializada a más de una expresión residual. La polivarianza puede obtenerse de distintas formas; en particular, el sistema de especialización de tipos original [Hughes, 1996b] incluye construcciones para productos polivariantes (donde una única expresión fuente e especializa a una tupla de expresiones $(e'_1, \ldots, e'_n)$) y para sumas polivariantes (donde una expresión etiquetada con un

constructor *In e* especializa a varias expresiones con varios constructores $In_1\ e'_1, \ldots, In_n\ e'_n$). Esta última se conoce también como especialización de constructores [Mogensen, 1993].

La especialización principal de tipos fue formulada sólo para un subconjunto del lenguaje presentado originalmente; en particular, las sumas polivariantes no fueron consideradas. En esta tesis extendemos el sistema con nuevas construcciones y reglas para especializar sumas polivariantes. Demostramos que nuestra contribución preserva todas las propiedades del sistema orignal, incluyendo la de principalidad, e incorporamos nuestra extensión a *PTS*, un prototipo de implementación de este sistema.

Especialización Principal de Tipos

En esta sección describimos con ejemplos las características más importantes del sistema de especialización principal de tipos, con énfasis en las partes mayormente invoulcradas con nuestro trabajo. Trabajamos únicamente con un subconjunto del lenguaje propuesto por Martínez López [2005], suficiente para ilustrar los elementos característicos del sistema.

La clave para obtener especialización de tipos es la riqueza del sistema de tipado residual. En él se pretende capturar toda la información estática proveniente de la expresión fuente, y más aún, especializar en forma general cualquier subexpresión. Motivado por esto último, el sistema está basado en la teoría de tipos calificados de Mark Jones [1994], que provee un nivel intermedio entre el tipado monomórfico y el polimórifco introduciendo *predicados* que restringen tipos. Por ejemplo, si $P(t)$ es un predicado sobre tipos, entonces se usan esquemas de la forma $\forall t.P(t) \Rightarrow f(t)$ para representar el conjunto

$$\{f(\tau) | \tau \text{ es un tipo tal que vale } P(\tau)\}$$

Recordemos que la especialización en este sistema se especifica a través de *reglas de especialización*, de la misma forma que un sistema de tipos se especializa con reglas de inferencia. Los juicios del sistema son de la forma

$$\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \ \hookrightarrow\ e' : \tau'$$

expresando que el término e de tipo τ especializa a la expresión residual e' de tipo τ' bajo los contextos Γ (donde se asume cómo especializan las variables libres de e) y Δ (donde se asumen predicados sobre los tipos residuales).

Esta forma de especificación para un especializador tiene la ventaja de ser modular: pueden agregarse nuevas construcciones al lenguaje fuente simplemente agregando nuevas reglas, sin necesidad de cambiar el resto del sistema.

Ejemplo 2 Las siguientes son especializaciones válidas en el contexto vacío. Se observa cómo cada expresión anotada como dinámica aparece en el término residual, mientras que la información de expresiones estáticas es trasladada al tipo residual.

1. $\vdash 11^D : Int^D \hookrightarrow 11 : Int$
2. $\vdash 11^S : Int^S \hookrightarrow \bullet : \hat{11}$
3. $\vdash (2^D\ +^D 1^D\ ) +^D 1^D : Int^D \hookrightarrow (2+1)+1 : Int$
4. $\vdash (2^S\ +^S 1^S\ ) +^S 1^S : Int^S \hookrightarrow \bullet : \hat{4}$

$\diamond$

Predicados y evidencia

Consideremos la especialización de la función que toma un entero estático y lo convierte en uno dinámico:

$$\lambda^D x.\textbf{lift}\ x : Int^S \to^D Int^D$$

Siendo una función dinámica, esperamos que se convierta en una función residual, cuyo resultado, especialización de una expresión **lift**, debe ser un entero residual. Una especialización posible, por ejemplo, sería la que parte de asumir que el argumento es 3^S:

$$\lambda x'.3 : \hat{3} \to^D Int$$

Otra, la que asume que el argumento es 8^S:

$$\lambda x'.8 : \hat{8} \to^D Int$$

y de la misma forma para cualquier n. Para obtener una especialización general de esta función, se hace uso de variables de tipo, ligadas con el operador $\forall$. Una primera aproximación sería darle tipo residual $\forall t.t \to Int$. Sin embargo, este describe las funciones que toman argumento de *cualquier tipo* y devuelven un entero. En este caso, la función no puede recibir "cualquier tipo", sino sólo uno de la forma $\hat{n}$. Usamos entonces un predicado que describe la propiedad de ser de esta forma: el predicado IsInt. El tipo residual de la función es entonces

$$\forall t.\text{IsInt}\ t \Rightarrow Int$$

Ahora bien, resta definir nuestra expresión residual, o más precisamente, a qué valor entero debe especializar el cuerpo de la función. Hemos visto que para una expresión de tipo $\hat{n}$, el operador **lift** sobre ella debe especializar al entero n.

Para modelar esto, usamos la noción de *evidencia* introducida por Jones. La idea básica es que un objeto de tipo $\Delta \Rightarrow \tau$ sólo puede usarse si además tenemos una evidencia apropiada de que los predicados de Δ efectivamente valen. La evidencia se define junto con cada predicado como un término ad-hoc, y se nota $v : \delta$ para decir que v es evidencia de δ. Además, se cuenta con un conjunto numerable de *variables de evidencia* h, y con las operaciones de *abstracción y aplicación de evidencia*, $\Lambda h.e$ y $e(\!(v)\!)$ respectivamente. La abstracción de evidencia en una expresión indica que se está asumiendo que vale cierto predicado, que estará presente en el tipo de la expresión. Las reglas de especialización fuerzan a que exista una abstracción de evidencia en la expresión por cada predicado en su tipo. La aplicación es la operación inversa, que permite eliminar un predicado del tipo por haberlo "demostrado", esto es, habiendo construido evidencia para él.

En el caso del predicado IsInt, la evidencia es el número del tipo: en otras palabras, decimos que n : IsInt $\hat{n}$. Ahora bien, en nuestro caso no tenemos directamente un tipo $\hat{n}$ sino una variable que lo representa: usamos entonces una h que nos permite asumir que vale IsInt t. Esta variable aparecerá abstraída en la expresión, y es justamente la que usamos para el cuerpo de la función.

$$\vdash_{\text{P}} \lambda^D x.\textbf{lift}\ x : Int^S \to^D Int^D \ \hookrightarrow\ \Lambda h.\lambda x'.h : \forall t.\text{IsInt}\ t \Rightarrow t \to Int$$

Polivarianza

Hasta ahora, hemos mostrado ejemplos de especialización *monovariante*, es decir, aquella en que las variables estáticas pueden tomar sólo un valor estático. Por ejemplo, la expresión

$$\mathbf{let}^D\ f = \lambda^D x.\mathbf{lift}\ x\ \mathbf{in}\ f\ @^D\ 11^S\ : \mathit{Int}^D$$

puede especializarse a

$$\mathbf{let}\ f' = \lambda x'.11\ \mathbf{in}\ f'@\bullet : \mathit{Int}$$

Pero esta otra expresión, similar,

$$\begin{array}{l}\mathbf{let}^D\ f = \lambda^D x.\mathbf{lift}\ x\\ \quad\mathbf{in}\ \left(f\ @^D\ 11^S, f\ @^D\ 6^S\right)^D : \left(\mathit{Int}^D, \mathit{Int}^D\right)^D\end{array}$$

no se puede especializar, porque f' no puede tener tipos $\hat{1}1 \to \mathit{Int}$ y $\hat{6} \to \mathit{Int}$ simultáneamente. Cualquier especializador útil necesita especialización polivariante, generando en este caso al menos dos versiones de f': una por cada argumento estático.

Introducimos el operador **poly** para producir especializaciones polivariantes. Una expresión encapsulada con la palabra **poly** especializa a un tipo residual general (también envuelto en una anotación **poly**) que se puede instanciar con cada uso. El operador **spec** produce una instanciación adecuada.

Con estos operadores, la función f de arriba puede hacerse polivariante, para poder ser aplicada a más de un argumento estático. Como ya no tendrá tipo función, no podrá ser aplicada directamente, sino que tendrá que llevar un **spec** antes.

$$\begin{array}{l}\mathbf{let}^D\ f = \mathbf{poly}\ \lambda^D x.\mathbf{lift}\ x\\ \quad\mathbf{in}\ \left(\mathbf{spec}\ f\ @^D\ 11^S, \mathbf{spec}\ f\ @^D\ 6^S\right)^D : \left(\mathit{Int}^D, \mathit{Int}^D\right)^D\end{array}$$

Ahora f especializa a la expresión general que ya hemos visto para esta función: $\Lambda h.\lambda x'.h$: **poly** $(\forall t.\mathrm{IsInt}\ t \Rightarrow t)$. Llamemos $f' = \Lambda h.\lambda x'.h$ y $\sigma = \forall t.\mathrm{IsInt}\ t \Rightarrow t$. Veamos cómo generar cada instanciación.

Como **spec** f aparece aplicada a 11^S, sabemos que debe tener tipo residual $\hat{1}1 \to \mathit{Int}$. Si debemos ver este tipo como una instancia del tipo de f', es decir como una instancia de σ, podemos entender que efectivamente lo es, en el sentido de que $\hat{1}1$ es un *caso particular* de t. Este concepto se modela en el sistema a través de una relación llamada *más general*, notada con el símbolo $\geq$. Informalmente, decimos que σ_1 es más general que σ_2 si toda vez que se requiere un objeto de tipo σ_2 puede usarse uno de tipo σ_1 en su lugar. En nuestro caso, vale que

$$\forall t.\mathrm{IsInt}\ t \Rightarrow t \geq \hat{1}1 \to \mathit{Int}$$

Ahora bien, esta definición no está completamente refinada. En realidad, las reglas de tipado no permiten, en general, usar un término de tipo σ_1 donde se requiere uno de tipo σ_2. Por ejemplo, en nuestro caso, no podemos usar f' directamente como especialización de **spec** f, ya que deberíamos aplicarla a $\bullet : \hat{1}1$. Para aplicar algo a esta expresión, las reglas de especialización (siguiendo la línea de reglas de tipado) exigen que se trate exactamente de una función $\hat{1}1 \to \mathit{Int}$ (sin predicados ni variables de tipo).

Lo que falta es un modo de adaptar f' de manera tal que efectivamente tenga el tipo que necesitamos. Observemos que en $\Lambda h.\lambda x'.h$ hay una abstracción de evidencia, necesaria para asumir el predicado IsInt t. En algo de tipo $\hat{1}1 \to \mathit{Int}$, no necesitamos dicha abstracción,

ya que $\hat{1}1$ ocupa el lugar de t y sabemos que 11 es la evidencia que corresponde. En otras palabras, estaríamos *demostrando* que vale IsInt t al asociar t con $\hat{1}1$ y construir evidencia 11. Como mencionamos antes, la operación de eliminar predicados y demostrarlos se refleja en la expresión a través de la *aplicación de evidencia*.

$$(\Lambda h.\lambda x'.h)(\!|(11)|\!) \triangleright \lambda x'.11$$

Esta operación de aplicar la evidencia 11 no sirve sólo para nuestra expresión en particular, sino para convertir cualquier expresión de tipo σ en una de tipo $\hat{1}1 \to \mathit{Int}$. Usamos una forma de contexto, llamada *conversión*, para expresar esta operación en términos abstractos. Finalmente, la relación "más general" tiene en realidad tres partes: un tipo general σ_1, una instancia σ_2, y una conversión que transforma expresiones del primer tipo en el segundo.

$$[](\!|(11)|\!) : \forall t.\text{IsInt } t \Rightarrow t \geq \hat{1}1 \to \mathit{Int}$$

Teniendo esto, sólo queda aplicar la evidencia 11 a f' antes de utilizarla como función. La instanciación correspondiente a **spec** 6 es análoga. Nuestra expresión puede especializarse finalmente a

$$\begin{array}{l}\textbf{let } f' = \Lambda h.\lambda x'.h \\ \quad \textbf{in } (f'(\!|(11)|\!)@\bullet, f'(\!|(6)|\!)@\bullet) : (\mathit{Int}, \mathit{Int})\end{array}$$

Principalidad

La propiedad más importante de este sistema es la existencia de especializaciones *principales*: toda subexpresión especializable tiene una especialización que es más general que cualquier otra válida, y tal que todas ellas pueden obtenerse a partir de la primera según una noción apropiada de instanciación.

Sumas Polivariantes

La formulación original de Martínez López fue hecha sólo para un pequeño subconjunto de un lenguaje fuente completo. En particular, las sumas dinámicas no fueron consideradas allí sino en una extensión propuesta por Alejandro Russo [2004].

Las *sumas* son tipos definidos por el programador, como una serie de constructores aplicados a un argumento. Por ejemplo, a partir de la definición

$$\mathit{data}\ \mathit{EitherSD}^D\ = \mathit{Sta}^D\ \mathit{Int}^S\ \mid \mathit{Dyn}^D\ \mathit{Int}^D$$

podemos construir la expresión

$$\left(\mathit{Sta}^D\ 4^S\,, \mathit{Sta}^D\ 9^S\right)^D : \left(\mathit{EitherSD}^D\,, \mathit{EitherSD}^D\right)^D$$

Tratándose de sumas *dinámicas*, esperamos que las construcciones asociadas con ellas permanezcan en el código residual: la declaración del tipo, las expresiones acompañadas de un constructor y las operaciones de pattern matching. La expresión anterior especializa al siguiente código residual:

$$\begin{array}{l}\mathit{data}\ \mathit{EitherSD}^1\ = \mathit{Sta}^1\ \hat{4} \\ \mathit{data}\ \mathit{EitherSD}^2\ = \mathit{Sta}^2\ \hat{9} \\ \\ (\mathit{Sta}^1\ \bullet, \mathit{Sta}^2\ \bullet) : (\mathit{EitherSD}^1, \mathit{EitherSD}^2)\end{array}$$

que ilustra varias de las características de la especialización de sumas dinámicas:

- Se pueden generar varias declaraciones de tipos residuales a partir de una única declaración fuente. En este caso, no hay necesidad de que los dos argumentos de la tupla especialicen a algo del mismo tipo. Entonces, han sido generados dos tipos residuales distintos, distinguidos por superíndices, y los constructores han sido numerados en forma acorde.

- No todo constructor que aparece en la declaración fuente debe aparecer en las declaraciones residuales. En este caso, como *Dyn* no es utilizado para construir ninguna expresión, no es necesario que forme parte del tipo.

- Los constructores aparecen en el código residual, pero sus argumentos son especializados normalmente, y en caso de haber información estática, esta pasa a la declaración del tipo. En el ejemplo, los argumentos son enteros estáticos, con lo cual especializan a la constante $\bullet$, y su valor pasa al tipo de cada constructor residual: $Sta^1\,\hat{4}$ y $Sta^2\,\hat{9}$ respectivamente.

A pesar de poder generar varias copias de un mismo tipo fuente, dentro de cada copia, puede haber al menos un constructor residual por cada constructor en la definición fuente. Por ejemplo, la siguiente es una expresión similar a la anterior

$$\begin{array}{l}\textbf{let}^D\ \ id = \lambda^D x.x\ \textbf{in}\\ \quad \left(id\ @^D\ (Sta^D\ 4^S), id\ @^D\ (Sta^D\ 9^S)\right)^D : \left(EitherSD^D, EitherSD^D\right)^D\end{array}$$

pero no puede ser especializada. Como las dos expresiones con constructores son argumento de una misma función, deben tener el mismo tipo residual $EitherSD'$, ¡pero este tipo no puede tener los constructores $Sta\ \hat{4}$ y $Sta\ \hat{9}$ al mismo tiempo!

Las sumas dinámicas *polivariantes*, la contribución principal de nuestro trabajo, permiten generar múltiples copias de un mismo constructor dentro de la misma declaración residual. Declaramos una suma polivariante con la palabra **polydata**, obteniendo la siguiente especialización:

$$\begin{array}{l}data\ EitherSD^D\ = Sta^1_1\ \ \hat{4} \mid Sta^1_2\ \ \hat{9}\\ \textbf{let}\ id' = \lambda^D x'.x'\ \textbf{in}\\ \quad (id'@(Sta^1_2\ \ \bullet), id'@(Sta^1_2\ \ \bullet)) : (EitherSD^1, EitherSD^1)\end{array}$$

donde se ha generado una única declaración residual con dos constructores *Sta*, distinguidos con subíndices, cada uno con el argumento adecuado.

Las sumas polivariantes son una forma alternativa para obtener polivarianza. En el ejemplo anterior, el constructor *Sta* se aplica a dos argumentos con distinto tipo residual — $\hat{4}$ y $\hat{9}$ — para producir expresiones con el mismo: $EitherSD^1$. De esta manera, permiten también pasar distintos argumentos estáticos a la misma función, *id* en este caso. Volviendo a la expresión que no podíamos especializar sin el uso de anotaciones **poly** y **spec**

$$\begin{array}{l}\textbf{let}^D\ \ f = \lambda^D x.\textbf{lift}\ x\\ \quad \textbf{in}\ \left(f\ @^D\ 11^S, f\ @^D\ 6^S\right)^D : \left(Int^D, Int^D\right)^D\end{array}$$

ahora podemos, en lugar de dar polivarianza a la función f, dársela a sus argumentos, encapsulándonos en un constructor de una suma polivariante, que llamamos *Poly*, y usando pattern

matching para desencapsularlo.

$$\begin{array}{l}\textbf{polydata}\ P^D\ =\ Poly^D\ Int^S\\ \textbf{let}^D\ f = \lambda^D px.\textbf{case}^D\ px\ \textbf{of}\\ \qquad\qquad Poly^D\ x\ \rightarrow\ \textbf{lift}\ x\\ \quad \textbf{in}\ \left(f\ @^D\ (Poly^D\ 11^S), f\ @^D\ (Poly^D\ 6^S)\right)^D : \left(Int^D, Int^D\right)^D\end{array}$$

Esta expresión especializa a

$$\begin{array}{l}data\ P^1 = Poly_1^1\ \hat{1}1 \mid Poly_2^1\ \hat{6}\\ \textbf{let}\ f' = \lambda px'.\textbf{case}\ px'\ \textbf{of}\\ \qquad\qquad Poly_1^1\ x\ \rightarrow\ 11\\ \qquad\qquad Poly_2^1\ x\ \rightarrow\ 6\\ \quad \textbf{in}\ (f@(Poly_1^1\ \bullet), f@(Poly_2^1\ \bullet)) : (Int, Int)\end{array}$$

lo cual ilustra la característica más importante de las sumas polivariantes: la capacidad de replicar las ramas de una expresión de pattern matching tantas veces como copias de los constructores se hayan generado, y de especializar cada una de ellas según el argumento de la copia que le corresponde.

Predicados y evidencia para sumas polivariantes

La definición de las reglas de especialización para sumas polivariantes sigue el esquema propuesto para todo el sistema, compuesto de dos fases:

1. La *especialización* propiamente dicha, donde se construye una descripción del problema: tipos restringidos con predicados y expresiones residuales intermedias que contienen la evidencia necesaria (abstraída en forma de variables) para construir el término residual final.

2. La *resolución* o *constraint solving*, donde se encuentra una solución al problema construido. Se obtiene la evidencia de cada predicado y se la usa en la expresión intermedia, que una vez que cuenta con toda la información, se reduce al término residual final.

En nuestro caso, durante la fase de especialización, cada suma residual está representada por una variable de tipo, y un conjunto de predicados describe cómo debe estar formado: qué constructores debe tener, con qué argumentos, etc. Sólo durante la fase de resolución se construye efectivamente la definición de una suma residual, de manera tal que satisfaga todos los predicados reunidos en la especialización.

El siguiente ejemplo muestra una especialización de la primera fase. A partir de él, describimos brevemente los predicados y formas de evidencia introducidos para especializar sumas polivariantes, y cómo esta última participa de la construcción de las expresiones finales.

Ejemplo 3 La expresión

$$\textbf{polydata}\ EitherSD^D\ =\ Sta^D\ Int^S\ \mid\ Dyn^D\ Int^D$$

$$\begin{array}{l}\textbf{let}^D\ f = \lambda^D x.\textbf{case}^D\ x\ \textbf{of}\\ \qquad\qquad Sta^D\ y\ \rightarrow\ \textbf{lift}\ y\\ \qquad\qquad Dyn^D\ y\ \rightarrow\ 4^D\\ \quad \textbf{in}\ \left(f\ @^D\ (Sta^D\ 11^S), f\ @^D\ (Sta^D\ 6^S)\right)^D : \left(Int^D, Int^D\right)^D\end{array}$$

especializa a

$$
\begin{array}{l}
\Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7. \\
\textbf{let } f' = \lambda x'.\textbf{polycase}_v\ x'\ \textbf{with}\ h_1\ \textbf{and}\ (h_6, h_7)\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Sta}\ \rightarrow\ \Lambda h.\lambda y'.h \\
\qquad\qquad\qquad\qquad \mathit{Dyn}\ \rightarrow\ \lambda y'.4 \\
\quad \textbf{in}\ (f@(\mathit{Sta}^{h_4}\ \bullet), f@(\mathit{Sta}^{h_5}\ \bullet)) \\
: \forall t.\ h_1 : \text{IsPolySum}\ t, \\
\qquad h_2 : \text{HasMGC}\ t\ \ \mathit{Sta}\ \ (\forall t'.\text{IsInt}\ t' \Rightarrow t'), \\
\qquad h_3 : \text{HasMGC}\ t\ \ \mathit{Dyn}\ \ \mathit{Int}, \\
\qquad h_4 : \text{HasPolyC}\ t\ \ \mathit{Sta}\ \ \hat{1}1, \\
\qquad h_5 : \text{HasPolyC}\ t\ \ \mathit{Sta}\ \ \hat{6}, \\
\qquad h_6 : \text{HasMGBr}\ t\ \ \mathit{Sta}\ \ (\forall t'.\text{IsInt}\ t' \Rightarrow t' \rightarrow \mathit{Int})\ \ \mathit{Int}, \\
\qquad h_7 : \text{HasMGBr}\ t\ \ \mathit{Dyn}\ \ (\mathit{Int} \rightarrow \mathit{Int})\ \ \mathit{Int} \Rightarrow (\mathit{Int}, \mathit{Int})
\end{array}
$$

$\diamond$

La sola presencia de una suma polivariante en el tipo fuente genera una variable de tipo t con el predicado IsPolySum t, que indica simplemente que t representa una suma residual. Esta suma, una vez resuelta, puede estar definida de distintas formas, tener todos los constructores (*Sta* y *Dyn* en este caso) o no, y distinto número de copias con distintos argumentos. Sin embargo, no cualquier suma residual sería un resultado razonable. Por ejemplo, la declaración en el programa fuente indica que el argumento de *Sta* es un entero estático. Entonces, una suma residual donde dicho constructor aparezca, por ejemplo, con un argumento de tipo función, no debería valer, ya que de ningún modo un entero estático puede especializar a una función. Motivado por esta observación, el predicado HasMGC introduce una *cota superior* en todo posible argumento de cada constructor. Por ejemplo, el predicado de evidencia h_2 indica que cada argumento de *Sta* en la definición residual de t debe ser una instancia de $\forall t'.\text{IsInt}\ t' \Rightarrow t'$, en el sentido de la relación "más general" ya introducida.

La construcción de las expresiones $\mathit{Sta}^D\ 11^S$ y $\mathit{Sta}^D\ 6^S$ en el programa fuente, ambas argumento de f, impone más restricciones sobre t. Necesariamente deberá incluir al menos dos copias del constructor *Sta*, uno con argumento $\hat{1}1$ y otro con argumento $\hat{6}$. Para expresar esto, se usa el predicado HasPolyC, que impone *requerimientos* sobre la presencia de constructores y sus argumentos en el tipo. La evidencia para estos predicados representa el número de declaración a la que corresponde el argumento (recordemos que podría generarse más de una declaración residual) y el número de copia del constructor, de manera tal que con h_4 y h_5 se generarán los índices adecuados para *Sta* en la expresión residual.

La especialización de la estructura de pattern matching es la más compleja. Como no se conoce a priori exactamente qué constructores tendrá el tipo residual, no puede saberse qué ramas debe tener el **case** residual. Resolvemos esta especialización con una estrategia análoga a la de las funciones polivariantes: especializamos cada rama una única vez, como una función general, y cada replicación es una instanciación de dicha rama.

La expresión intermedia $\textbf{polycase}_v$ especializa cada rama vista como una función que toma la variable de pattern matching y devuelve el lado derecho de la rama. Por ejemplo, la rama $\mathit{Sta}^D\ y\ \rightarrow\ \textbf{lift}\ y$ puede verse como la función $\lambda^D y.\textbf{lift}\ y$, que especializa a la función general $\Lambda h.\lambda y'.h : \forall t.\text{IsInt}\ t \Rightarrow t \rightarrow \mathit{Int}$. Ahora toda posible rama sobre el constructor *Sta* deberá ser una instancia de esta función, donde el argumento estará especializado. En otras palabras, para todo posible argumento residual τ del constructor, la función de tipo

$\tau \to \mathit{Int}$ debe ser una instancia de $\forall t.\text{IsInt } t \Rightarrow t \to \mathit{Int}$. El predicado HasMGBr expresa esta restricción. El caso del constructor *Dyn* es análogo: la rama especializa a la función general $\lambda y'.4 : \mathit{Int} \to \mathit{Int}$ y el predicado HasMGBr restringe todas las posibles ramas poniendo este tipo como cota superior.

Veamos ahora cómo se resolvería esta expresión. Asociamos t con los únicos constructores requeridos, $\mathit{Sta}_1^1\ \hat{1}1$ y $\mathit{Sta}_2^1\ \hat{6}$. La evidencia h_1 de que t es una suma residual es el conjunto de sus constructores, $\{\mathit{Sta}_1^1, \mathit{Sta}_2^1\}$. De esto se deduce que *Dyn* no aparece en el tipo (notemos que en el programa fuente no se utilizó para construir ninguna expresión), y entonces la rama correspondiente no va a aparecer en el **case** residual. Por este motivo, la evidencia h_7 no se usa.

La evidencia h_6 es la clave para generar las ramas replicadas del constructor *Sta*. Esta evidencia representa el hecho de que todos las ramas son una instancia de la general; en otras palabras, una vez resuelta, contiene las *conversiones* que asocian cada rama instanciada con el tipo general. En particular en nuestro caso:

$$[](\!(11)\!) : (\forall t'.\text{IsInt } t' \Rightarrow t' \to \mathit{Int}) \geq (\hat{1}1 \to \mathit{Int})$$
$$[](\!(6)\!) : (\forall t'.\text{IsInt } t' \Rightarrow t' \to \mathit{Int}) \geq (\hat{6} \to \mathit{Int})$$

Además, tenemos que:

$$(\Lambda h.\lambda y'.h)(\!(11)\!) \triangleright \lambda y'.11$$
$$(\Lambda h.\lambda y'.h)(\!(6)\!) \triangleright \lambda y'.6$$

¡y con esto, las funciones resultantes pueden volver a verse como ramas! De $\lambda y'.11$ podemos pasar nuevamente a $\mathit{Sta}_1^1\ y' \to 11$, y de la misma forma para la otra rama:

$$\mathit{data}\ \mathit{EitherSD}^1 = \mathit{Sta}_1^1\ \hat{1}1 \mid \mathit{Sta}_2^1\ \hat{6}$$

$$\begin{array}{l}\textbf{let } f' = \lambda x'.\textbf{case } x' \textbf{ of} \\ \qquad\qquad\qquad\qquad \mathit{Sta}_1^1\ y' \to 11 \\ \qquad\qquad\qquad\qquad \mathit{Sta}_2^1\ y' \to 6 \\ \quad \textbf{in } (f@(\mathit{Sta}_1^1\ \bullet), f@(\mathit{Sta}_2^1\ \bullet)) : (\mathit{Int}, \mathit{Int})\end{array}$$

Resumen y trabajo futuro

En esta tesis, extendimos el sistema de especializón principal de tipos para especializar sumas polivariantes. Incorporamos nuevas construcciones al lenguaje de términos y tipos fuente, términos y tipos residuales, predicados y evidencia, y agregamos las reglas de especialización necesarias. Además, demostramos que nuestra extensión preserva las propiedades del sistema, en particular, la existencia de especializaciones principales. Extendimos el algoritmo para computar especializaciones principales, dimos una heurística para constraint solving y extendimos la implementación de ambos en un prototipo escrito en Haskell.

Dentro de las posibilidades de trabajo futuro se encuentra formalizar y mejorar la fase de constraint solving para sumas dinámicas. Existe una formalización de este proceso formulado únicamente para un pequeño conjunto de predicados; incorporar las sumas dinámicas ayudaría a definir este mecanismo más precisamente y encontrar posibilidades de mejora.

En cuanto al sistema en general, el lenguaje fuente todavía necesita ser extendido. De las características que todavía no posee la más importante es sin dudas la especialización de funciones recursivas dinámicas. Además, puede considerarse la generación de sumas dinámicas paramétricas y recursivas (lo cual permitiría modelar listas, árboles, etc.).

Contents

Chapter 1

Introduction

1.1 Program Specialization

Automatic program production is to programming as a weaving machine is to cloth making. Some repetitive, error-prone or time-consuming tasks no longer need to be performed by a human being, leaving room for more sophisticated activities such as planning and designing, and improving the discipline to levels originally limited by human ability and physical or mental state. We can find many examples of programs generating programs, most of them restricted to a specific domain, such as parser generators.

Automatic program generation studies this problem from a general point of view. There are several different ways to automatically produce a program; *program specialization* is perhaps the most successful. When solving a set of similar problems, a programmer can choose between writing many small efficient programs and writing a bigger, less efficient program which solves any of them depending on the data. Program specialization takes the best of both worlds: given a general program, it produces one or more versions of it, each specialized to particular data. The program used as input is called the *source program*, and those produced as output are called the *residual programs*.

A classic example is the *power* function calculating x^n

$$\begin{aligned} power\ n\ x\ =\ & \textbf{if}\ n\ ==\ 1\ \textbf{then} \\ & \quad x \\ & \textbf{else} \\ & \quad x\ *\ power\ (n-1)\ x \end{aligned}$$

whose computation involves several comparisons and recursive calls, but when the input parameter n is known, it can be specialized to a non-recursive residual version. For example, if n is known to be 3, the function

$$power_3\ x\ =\ x\ *\ (x\ *\ x)$$

would be a proper specialization, clearly much more efficient than the source version when computing cubes.

1.1.1 Partial evaluation

Program specialization has been studied from several different approaches; *partial evaluation* [Jones *et al.*, 1993; Consel and Danvy, 1993] is by far the most popular and well-known.

Partial evaluation produces residual programs by using a generalized form of reduction: subexpressions with known arguments are replaced by the result of their evaluation and combined with the computations that cannot be performed. That is, a partial evaluator works with the *text* of the source program by fixing some of the input data (the *static data*) and performing a mixture of computation and code generation to produce a new program. The programs produced, when run on the remaining data (the *dynamic data*), yield the same result as the original program run on all the data.

Partial evaluation may sound like a sophisticated form of constant folding, but in fact a wide variety of powerful techniques are needed to do it successfully, and these may completely transform the structure of the original program.

An area where partial evaluation is particularly successful is the automatic production of compilers: compilation is obtained by specializing an interpreter for a language to a given object program [Futamura, 1971; Jones *et al.*, 1985; Jones *et al.*, 1989; Wand, 1982; Hannan and Miller, 1992]. Let us suppose we have an interpreter for language B written in language A, and we specialize it taking an object program P_B as static data. The residual program, P'_A, run by itself behaves the same way as running P_B on the interpreter. So P'_A is a compiled version of P_B to language A!

Another layer of complexity can be added when the partial evaluator is written in the language it specializes: self-application becomes possible, and thus compilers can be generated as well. The (code of the) partial evaluator is the source program and the interpreter is the static data; the resulting residual program performs specializations of the interpreter mentioned above. Now the residual program expects an object program as input and produces a compiled version of it: a compiler! This is very useful in the area of domain-specific languages [Thibault *et al.*, 1998], where the cost of generating a compiler must be kept to a minimum.

Other areas where partial evaluation has been applied successfully include software architectures [Marlet *et al.*, 1999], networking [Muller *et al.*, 1998], hardware design and verification [Hogg, 1996; Au *et al.*, 1991], virtual worlds [Beshers and Feiner, 1997], numerical computation [Lawall, 1998] and aircraft crew planning [Augustsson, 1997].

1.1.2 Optimality

An important notion in the program specialization approach is that of *optimality*. Intuitively, optimal specializations are those that leave no unnecessary *traces* of the source program in their results. Neil Jones [1988] defines a notion of optimality (also called Jones optimality) by specializing a self-interpreter and comparing the source program with its residual version: if they are essentially the same, we say that the specialization was optimal. Robert Glück showed that Jones optimality plays an important role in binding time improvements [Glück, 2002].

Partial evaluation alone can only obtain optimality for self-interpreters written in untyped languages, but it cannot if they are written in a typed language. As partial evaluation works by reduction, the type of the residual program is constrained by that of the source one. In particular, the residual code contains tagging and untagging operations coming from the representation of programs in the interpreter, that is, it contains *traces* of the source program. That means the source and residual programs are not "essentially the same", so optimality is not achieved. This problem was stated by Neil Jones as one of the open problems in the partial evaluation field [Jones, 1988].

1.2 Type Specialization

Type specialization is an approach to program specialization proposed by John Hughes [1996b; 1996a; 1998]. His main motivation was to provide optimal specialization for interpreters written in typed languages, which was later also achieved by applying different techniques [Thiemann, 1999; Thiemann, 2000; Taha *et al.*, 2001], but it proved to be in itself a very interesting framework for program specialization. The key idea is to specialize a source expression together with its type, to obtain a residual program *with a residual type*. Whereas partial evaluation is based on a generalized form of reduction, type specialization is based on a generalized form of type inference — in this sense, it has introduced a new paradigm for program specialization.

In all typed languages, types provide information about expressions. For example, when an expression is of type *Int*, we know that, if its evaluation terminates, it will yield an integer. But if the expression is known to be the constant 11, for example, a more refined approximation is possible by having a *type representing the property* of being the integer 11 — let's call this type $\hat{11}$. Having all the information in the type, there is no need to execute the program anymore, so the integer constant can be replaced by a dummy value having type $\hat{11}$ — that is, the source expression $11 : \mathit{Int}$ can be specialized to $\bullet : \hat{11}$.

Type specialization extends residual types to give as much information about expressions as possible, and works by propagating static information in the source code to residual types. This involves a more powerful residual type system, which is the key fact allowing optimal specialization for typed interpreters.

Like many partial evaluators, type specialization processes a two-level language [Gomard and Jones, 1991]; that is, each construct in the source program is labelled either static or dynamic. For example, the number 3 can appear either statically ($3^S : \mathit{Int}^S$) or dynamically ($3^D : \mathit{Int}^D$). We will denote static and dynamic constructs with ^{S} and ^{D} superscripts respectively.

Just as a type checker can be specified by a set of type inference rules, a type specializer is specified by a set of *specialization rules*. Judgments inferred by these rules are of the form

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

meaning expression e of type τ specializes to expression e' of type τ' under context Γ (containing assumptions on the specialization of free variables).

Specification of a type specializer by a set of inference rules has the advantage of being modular — new constructs can be added to the source language just by adding rules for their specialization, without changing the rest of the system.

Example 1.1 The following are all valid specializations under the empty context. Observe how every expression annotated as dynamic appears in the residual term, whereas information from static expressions is moved into the residual type.

1. $\vdash 11^D : \mathit{Int}^D \hookrightarrow 11 : \mathit{Int}$
2. $\vdash 11^S : \mathit{Int}^S \hookrightarrow \bullet : \hat{11}$
3. $\vdash (2^D +^D 1^D) +^D 1^D : \mathit{Int}^D \hookrightarrow (2+1)+1 : \mathit{Int}$
4. $\vdash (2^S +^S 1^S) +^S 1^S : \mathit{Int}^S \hookrightarrow \bullet : \hat{4}$

◇

The problem of specializing a program f with parameters $x_1, \ldots, x_n$ where $x_1, \ldots x_k$ are known can be expressed as specializing the expression

$$f \ @^S x_1 \ @^S \ \ldots \ @^S x_k \ @^D x_{k+1} \ @^D \ \ldots \ @^D x_n$$

where f is a function and $f@x$ represents application. But type specialization is a more general approach, allowing much more flexible combinations of static and dynamic annotations. Residual types can express 'partially static' information — $(\hat{4}, \mathit{Int})$ represents the type of pairs whose first component is statically 4, for example. Static information can then be associated with dynamic values and propagated by type inference.

Example 1.2 Consider the expression

$$\left(\lambda^D f.\mathbf{lift} \ \left(f \ @^D \ 3^S\right)\right) \ @^D \ \left(\lambda^D x.x +^S 1^S\right) : \mathit{Int}^D$$

Here, we have dynamic λ-expressions and dynamic application, so they will be transformed into residual λ-expressions and applications in the specialized program. The **lift** operator converts an expression of a static integer type to its dynamic value — if a static integer expression e has residual type $\hat{n}$, then **lift** e specializes to $n : \mathit{Int}$.

To specialize this expression, we infer the residual type of each subexpression. Firstly, f is applied to an argument with residual type $\hat{3}$, so it must have residual type $\hat{3} \to \tau$ for some τ. Now this must also be the type of $\left(\lambda^D x.x +^S 1^S\right)$, so x must have residual type $\hat{3}$. Then $\left(x +^S 1^S\right)$ must have type $\hat{4}$ — we can conclude τ is $\hat{4}$, f has type $\hat{3} \to \hat{4}$ and $f \ @^D 3$ has type $\hat{4}$. Finally, the **lift** operation can be specialized to 4, so the dynamic application of f does not appear in the specialized program — this is the only kind of situation where a dynamic construct is removed from the residual code. The final result is:

$$(\lambda f.4) \, @ \, (\lambda x.\bullet) : \mathit{Int}$$

Observe how the value 4 in the residual code has been obtained without actually unfolding any of the functions. ◇

In Hughes's formulation, both the source and the residual type systems are monomorphic. In addition, some of the rules are not completely syntax-directed, so for some source expressions, different *unrelated* specializations can be produced. This is comparable to the problem posed in the simply-typed λ-calculus when typing expressions like $\lambda x.x$: the type of x depends on the context of use, and no relation between the different types is expressible in the system. The solution to this last problem is a polymorphic type system, where a *principal type* expresses all possible ways to type a given term. Hughes's type specialization formulation lacks the property of *principality*.

The lack of principality has some serious consequences. Firstly, extending the system to produce polymorphic residual code or to treat polymorphic source code is very difficult. Secondly, a specialization algorithm dealing with a valid source expression would have to fail or choose an arbitrary (and potentially erroneous) specialization if some of the context information happened to be missing. And thirdly, even if all the context information could be obtained, it is too restrictive for the whole specialization process to depend on the whole

context — it makes it virtually impossible to achieve specialization of program modules, for instance.

Pablo E. Martínez López presented a new system for type specialization which solves the problem of lack of principality [Martínez López and Hughes, 2002]. The system, called *principal type specialization*, is based on Mark Jones's theory of qualified types [Jones, 1994], briefly described below.

1.3 Qualified Types

Mark Jones's theory of qualified types [Jones, 1994] develops a general approach to constrained type systems providing an intermediate level between monomorphic and polymorphic typing disciplines. For example, if $P(t)$ is a predicate on types, then we can use a type scheme of the form $\forall t.P(t) \Rightarrow f(t)$ to represent the set of types

$$\{f(\tau) | \tau \text{ is a type such that } P(\tau) \text{ holds}\}$$

Jones describes an extension of Milner's polymorphic type system [Milner, 1978] which includes support for overloading based on the use of qualified types and parameterized by an arbitrary system of predicates. He defines an ordering on the set of type schemes and shows there is a type inference algorithm calculating principal types, that is, greatest possible types with respect to this ordering.

The theory of qualified types has a number of applications — in particular, it provides an elegant formalization of Haskell's type class system [Wadler and Blott, 1989; Peterson and Jones, 1993; Augustsson, 1993] — and is the main framework for the principal type specialization system. In the following lines, we will describe its main features with special emphasis on what is thoroughly used in Martínez López's and our work.

Predicates, type schemes and terms The key feature of this system is the use of a language of *predicates* to describe sets of types, or more generally, relations between types. The exact set may vary from one application to another; only a few properties on the predicate language are expressed in the form of an *entailment relation* to satisfy a few simple laws. For example, in Haskell's type system, each class such as *Eq a*, *Ord a*, etc. is modelled as a predicate.

Following the definition of types and type schemes in ML [Milner, 1978], a structured language of types is defined in three syntactic categories: types τ, qualified types ρ on top of them, and then type schemes σ:

$$\begin{aligned} \tau &::= t \mid \tau \to \tau \\ \rho &::= \delta \Rightarrow \tau \mid \tau \\ \sigma &::= \forall \alpha.\rho \mid \rho \end{aligned}$$

Here t and α range over a countably infinite set of type variables, and δ ranges over a finite set of predicates. Other type constructors such as type constants, tuples or lists can be easily added.

The term language is based on simple untyped λ-calculus with the addition of a *let* construct to enable the definition and use of polymorphic terms [Milner, 1978; Damas and Milner, 1982; Clément *et al.*, 1986]. That is, expressions can be either variables, λ-abstractions, applications or *let* constructs.

Evidence In order to discuss the semantics and evaluation of terms with qualified types, Jones introduces the concept of *evidence*. The essential idea is that an object of type $\Delta \Rightarrow \tau$ can only be used if we are also supplied with suitable evidence that the predicates in Δ indeed hold.

The treatment of evidence can be ignored in the basic typing algorithm — in fact, Jones's presentation begins by developing a language without evidence, and only then defines a new language which includes evidence expressions as a resource for describing overloading more precisely. However, in the principal type specialization system, evidence has a fundamental role as a part of the residual language, so we will directly describe the use of terms and predicates with evidence.

The set of terms is extended with a language of *evidence expressions* v denoting evidence values, including a countably infinite set of *evidence variables* h, disjoint from the sets of term and type variables. $EV(v)$ represents the set of free evidence variables in v, and $e[v/h]$ is the substitution of all evidence variables h in e for v. These are naturally extended to sets of variables $\bar{h}$, sets of expressions $\bar{v}$ and simultaneous substitutions $e[\bar{v}/\bar{h}]$.

The language must include the constructs needed for each particular application, plus *evidence abstraction* ($\Lambda h.e$) and *evidence application* ($e(\!(v)\!)$), whose use will be explained below. We write $v : \delta$ to express that v is evidence for predicate δ.

The following abbreviations are useful and will be used extensively throughout this work.

Object	Expression	Abbreviation
Evidence assignment	$v_1 : \delta_1, \ldots, v_n : \delta_n$	$\bar{v} : \Delta$
Qualified type	$\delta_1 \Rightarrow \ldots \Rightarrow \delta_n \Rightarrow \tau$	$\Delta \Rightarrow \tau$
Type scheme	$\forall\alpha_1 \ldots . \forall\alpha_n . \rho$	$\forall\bar{\alpha}.\rho$
Evidence abstraction	$\Lambda h_1 \ldots . \Lambda h_n . e$	$\Lambda\bar{h}.e$
Evidence application	$((e(\!(v_1)\!))\ldots)\,(\!(v_n)\!)$	$e(\!(\bar{v})\!)$

Entailment As we mentioned above, the set of predicates varies with the different applications, as do the evidence expressions that prove them. Properties of predicates are captured by an entailment relation $\Vdash$between finite lists of predicates. An entailment of the form $\bar{h} : \Delta \Vdash \bar{v} : \Delta'$ indicates that evidence $\bar{v}$ can be constructed for the predicates in Δ' assuming evidence $\bar{h}$ for Δ. In a context where evidence is irrelevant, we can also write $\Delta \Vdash \Delta'$.

In order for a particular predicate system to be suitable for this theory, it must satisfy a few rules, which are given in figure 1.1. Rule (Close) is needed to ensure that the system of predicates is compatible with the use of parametric polymorphism; here S denotes a substitution on the type variables appearing in the predicates in Δ. Rules (Evars) and (Rename) express properties on the evidence language, similar to those relating terms and types in a classic typing system.

Type inference Jones's extends the Hindley-Milner type inference system to include qualified types. Judgments are of the form

$$\bar{h} : \Delta \mid \Gamma \vdash e : \sigma$$

representing the fact that, when the predicates in Δ are satisfied with evidence $\bar{h}$ and the types of the free variables in e are specified in Γ, then e has type σ.

$$\text{(Id)}\quad \bar{h} : \Delta \Vdash \bar{h} : \Delta$$

$$\text{(Term)}\quad \bar{h} : \Delta \Vdash \emptyset$$

$$\text{(Fst)}\quad \bar{h}_1 : \Delta_1, \bar{h}_2 : \Delta_2 \Vdash \bar{h}_1 : \Delta_1$$

$$\text{(Snd)}\quad \bar{h}_1 : \Delta_1, \bar{h}_2 : \Delta_2 \Vdash \bar{h}_2 : \Delta_2$$

$$\text{(Univ)}\quad \frac{\bar{h}_1 : \Delta_1 \Vdash \bar{v}_2 : \Delta_2 \quad \bar{h}_1 : \Delta_1 \Vdash \bar{v}_3 : \Delta_3}{\bar{h}_1 : \Delta_1 \Vdash \bar{v}_2 : \Delta_2, \bar{v}_3 : \Delta_3}$$

$$\text{(Trans)}\quad \frac{\bar{h}_1 : \Delta_1 \Vdash \bar{v}_2 : \Delta_2 \quad \bar{h}_2 : \Delta_2 \Vdash \bar{v}_3 : \Delta_3}{\bar{h}_1 : \Delta_1 \Vdash \bar{v}_3[\bar{v}_2/\bar{h}_2] : \Delta_3}$$

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

$$\text{(Evars)}\quad \frac{\bar{h} : \Delta \Vdash \bar{v} : \Delta'}{EV(\bar{v}) \subseteq \bar{h}}$$

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

$$\text{(Dist)}\quad \frac{\bar{h}_1 : \Delta_1 \Vdash \bar{v}_1 : \Delta_1' \quad \bar{h}_2 : \Delta_2 \Vdash \bar{v}_2 : \Delta_2'}{\bar{h}_1 : \Delta_1, \bar{h}_2 : \Delta_2 \Vdash \bar{v}_1 : \Delta_1', \bar{v}_2 : \Delta_2'}$$

$$\text{(Cut)}\quad \frac{\bar{h}_1 : \Delta_1 \Vdash \bar{v}_2 : \Delta_2 \quad \bar{h}_1 : \Delta_1, \bar{h}_2 : \Delta_2 \Vdash \bar{v}_3 : \Delta_3}{\bar{h}_1 : \Delta_1 \Vdash \bar{v}_3[\bar{v}_2/\bar{h}_2] : \Delta_3}$$

Figure 1.1: Rules for predicate entailment

$$\Rightarrow intr \quad \frac{\Delta, h : \delta, \Delta' \mid \Gamma \vdash e : \rho}{\Delta, \Delta' \mid \Gamma \vdash \Lambda h.e : \delta \Rightarrow \rho}$$

$$\Rightarrow elim \quad \frac{\Delta \mid \Gamma \vdash e : \delta \Rightarrow \rho \quad \Delta \Vdash v : \delta}{\Delta \mid \Gamma \vdash e((v)) : \rho}$$

Figure 1.2: Rules relating predicates with types in the type inference system

Figure 1.2 shows the rules to construct and deconstruct qualified types — all the other rules correspond to the Hindley-Milner polymorphic typing system. Observe how evidence abstraction corresponds to adding a predicate to the qualified type, and evidence application to removing it.

Ordering on type schemes Motivated by the need to describe all of the ways in which a particular term e can be used with a given type assignment, Jones defines a preorder ($\geq$) (read *more general*). Essentially, a statement of the form $(\Delta' \mid \sigma') \geq (\Delta \mid \sigma)$ means it is possible to use an object of type σ in an environment satisfying the predicates in Δ whenever an object of type σ' is required in an environment satisfying the predicates in Δ'. He calls the pair $(\Delta \mid \sigma)$ a *constrained type scheme*.

For example, if P is a predicate on types and $\emptyset \Vdash P(Int)$, then it is true that

$$(\emptyset \mid \forall t.P(t) \Rightarrow t \to t \to Bool) \;\geq\; (\emptyset \mid Int \to Int \to Bool)$$

usually abbreviated as

$$\forall t.P(t) \Rightarrow t \to t \to Bool \;\geq\; Int \to Int \to Bool$$

This should mean that an object of type $\forall t.P(t) \Rightarrow t \to t \to Bool$ could be used in any context where an object of type $Int \to Int \to Bool$ was required. However, the typing rules in figure 1.2 make sure that evidence is abstracted whenever a predicate is introduced into the type, and applied when it is removed. So by virtue of these rules, an object of type $Int \to Int \to Bool$ should have a different structure than one of type $\forall t.P(t) \Rightarrow t \to t \to Bool$ — namely they would differ in the presence of evidence abstraction and application.

The observation above motivates the introduction of a third component into the $\geq$ relation: a *conversion*. We write $C : (\Delta' \mid \sigma') \geq (\Delta \mid \sigma)$ when any object of type σ' (in an environment constrained by Δ') can be *converted* to an object of type σ (in an environment constrained by Δ) by adding or removing evidence abstractions and applications. Jones gives a precise definition of conversions as a particular set of terms of the language. Martínez López's formulation involves the use of the $\geq$ relation and all the concepts mentioned here, but conversions are defined differently — namely as a particular set of contexts.

1.4 Principal Type Specialization

Principal type specialization is a new formulation to type specialization presented by Pablo E. Martínez López [2002; 2005]. Based on Jones's theory of qualified types, the system can

produce *principal* type specializations — for any source term and type, a specialization exists such that any other can be obtained from it by a suitable notion of instantiation.

Specialization is separated into two independent phases: constraint generation and constraint solving. The first phase works with *information flow* as it was described in section 1.2; when some of the information depends on the context of use, constraints play a crucial role. In the second phase, when there is enough information from the contexts, all the constraints are gathered and a suitable residual program is calculated [Martínez López and Badenes, 2003]. This separation provides a better understanding of the information flow during specialization, and enables the application of different heuristics to the process of calculating the right residual program. It also makes it possible to define modular specialization: for each module, a principal specialization can be computed independently, and when linking each residual code to the residual main program, the right instantiation can be produced.

An important notion in any specialization system is that of *polyvariance*. We say a specialization is monovariant when static variables can take only one static value. Monovariant specializations are seriously limited — for example, if $f : Int^S \to Int^D$ is applied to both 3^S and 4^S in the same source program, specialization cannot be achieved, because f cannot be assigned both residual types $\hat{3} \to Int$ and $\hat{4} \to Int$. A polyvariant system, in contrast, allows static variables to take more than one value.

Polyvariance can be achieved in more than one way; in particular, Hughes's system includes constructs for polyvariant products and polyvariant sums [Hughes, 1996b], the latter also known as *constructor specialization* [Mogensen, 1993].

Principal type specialization was originally formulated for a subset of the language presented by John Hughes. In particular, dynamic sum types were not considered. Later, Alejandro Russo [2004] extended the system to manipulate dynamic sum types — a set of *tagged* values in the form of constructors applied to arguments. Russo's extension incorporates named data types without recursion and provides monovariant specialization of expressions involving sum types, preserving the property of principality.

1.5 Contribution of this work

The main contribution of this work is to extend Martínez López's system [Martínez López and Hughes, 2002; Martínez López, 2005] and Russo's additions [Russo, 2004] to include polyvariant sum types. Expressions of the form $L\ x$, where L is a constructor and x is a static variable, can now be specialized to more than one value, generating copies of the constructor L_1, L_2, etc, one for each different use of x. As in Russo's extension, recursive data types are not considered.

We also give an alternative formulation to one of Russo's rules that allows better use of the static information involving the sum's definition. Our extension to polyvariant sums is based on this new formulation. We extend the source and residual languages, all the formal system rules and proofs to manipulate the new constructs, and we prove that the property of principality is preserved by our addition.

Constructor specialization is a useful feature on any specializer. The addition of polyvariant sums to the system is a small step toward making principal type specialization more powerful and closer to a real programming language.

1.6 Overview

This thesis is organized in six chapters. Chapter 2 introduces the principal type specialization system as formulated by Martínez López and extended by Russo. An alternative formulation to one of Russo's original rules for dynamic sum types is analyzed and explained in chapter 3. In chapter 4, we introduce polyvariant sum types to the system, and in chapter 5, we extend the algorithm computing type specializations and prove that the property of principality is preserved by our extensions. Finally, in chapter 6 we discuss related and future work and conclude.

Two appendixes are given so that the main reading is not interrupted. The first one contains some technical definitions and auxiliary systems. The second consists of the proofs of all the properties stated in chapters 4 and 5.

Together with this thesis, we present a prototype of a principal type specializer. Written in Haskell, it is an extension to the type specializer introduced by Martínez López [2005, chapter 10] that handles all our new constructs.

Chapter 2

Principal Type Specialization

In this chapter, we present the principal type specialization system. We explain the concepts behind type specialization in general and this system in particular, with emphasis on the elements that are most involved with our work.

The material in this chapter is based completely on Martínez López's work [2005, chapter 6 and section 9.6] and Russo's extensions [2004, chapter 3].

2.1 Source Language

The source language we consider is a λ-calculus enriched with local definitions, tuples and arithmetic constants and operations. Expressions are annotated as either static or dynamic, with superscripts ^{S} and ^{D} respectively.

Definition 2.1 Let x denote a *source term variable* from a countably infinite set of variables, and let n denote an integer number. A *source term*, denoted by e, is an element of the language defined by the following grammar:

$$
\begin{array}{rlll}
e ::= & x & \mid n^S & \mid n^D \\
\mid & e +^S e & \mid e +^D e & \mid \textbf{lift}\ e \\
\mid & \lambda^D x.e & \mid e\ @^D\ e & \mid \textbf{let}^D\ x = e\ \textbf{in}\ e \\
\mid & (e, \ldots, e)^D & \mid \pi^D_{n,n}\ e & \\
\mid & \textbf{poly}\ e & \mid \textbf{spec}\ e &
\end{array}
$$

where $(e_1, \ldots, e_n)^D$ is a finite tuple of expressions for every possible arity n. The projections $\pi^D_{1,2}\ e$ and $\pi^D_{2,2}\ e$ may be abbreviated $\textbf{fst}^D\ e$ and $\textbf{snd}^D\ e$ respectively.

Annotation ^{S} is interpreted as the requirement to remove an expression from the source program, by computing it and moving its result into the residual type, and annotation ^{D} as the requirement to keep the expression in the residual code. The **lift** operator casts a static expression into a dynamic value. The **poly** and **spec** annotations express polyvariance: the former allows a single expression to produce several different residual results in the specialized program, and the latter chooses an appropriate one among all that can occur — this is further explained in section 2.3.1.

Source types also reflect the static or dynamic nature of expressions. For example, the constant 42^D has type Int^D and the constant 42^S has type Int^S. Also as an example, a

dynamic function can only be dynamically applied, that is, in an expression like $(\lambda^D x.x)\ @^D\ y$, both $\_^D$'s correspond to each other. Additionally, expressions annotated with **poly** have a corresponding **poly** in their type.

Definition 2.2 A *source type*, denoted by τ, is an element of the language defined by the following grammar:

$$\tau ::= Int^D \mid Int^S \mid (\tau, \ldots, \tau)^D \mid \tau \rightarrow^D \tau \mid \textbf{poly}\ \tau$$

the type $(\tau_1, \ldots, \tau_n)^D$ being a finite tuple for every possible arity n.

This language is a small subset of the language of Hughes's type specializer [Hughes, 1996b], but contains enough constructs to illustrate the basic notions. We introduce tagged sum types in section 2.5.

2.2 Residual Language

The residual language has constructs and types corresponding to all the dynamic constructs and types in the source language, plus additional ones used to express the result of specializing static expressions.

2.2.1 Residual types

Based on the theory of qualified types presented in section 1.3, the residual type language includes predicates to express restrictions imposed by the source expressions and their context.

Definition 2.3 Let t denote a *type variable* from a countably infinite set of variables, and s a *type scheme variable* from another countably infinite set of variables, both disjoint with any other set of variables already used. A *residual type*, denoted by τ', is an element of the language given by the grammar

$$\begin{aligned}
\tau' &::= t \mid Int \mid \hat{n} \mid \tau' \rightarrow \tau' \mid (\tau', \ldots, \tau') \mid \textbf{poly}\ \sigma \\
\rho &::= \delta \Rightarrow \rho \mid \tau' \\
\sigma &::= s \mid \forall s.\sigma \mid \forall t.\sigma \mid \rho \\
\delta &::= \text{IsInt}\ \tau' \mid \tau' := \tau' + \tau' \mid \text{IsMG}\ \sigma\ \sigma
\end{aligned}$$

The intuition for predicate IsInt is that its argument is a one-point type, $\hat{n}$ — that is, a type with a single value, $\bullet : \hat{n}$. That of $\tau := \tau_1 + \tau_2$ is that the arguments are three one-point types $\hat{n}$, $\hat{n_1}$ and $\hat{n_2}$ such that $n = n_1 + n_2$. Predicate IsMG internalizes the "more general" relation corresponding to the principal type specialization theory — see section 2.2.4.

Free variables and substitutions are defined in the obvious way — with $\forall$ being the only binder — on both the set of type variables and type scheme variables. With α we represent an element of any of these sets.

2.2.2 Residual terms

Just as the residual types involve predicates, the term language has constructs to manipulate evidence. These include the structural components taken from the theory of qualified types itself — namely evidence variables, abstraction and application — and other constructs needed

$$(\beta_v) \quad (\Lambda h.e'_1)(\!(v)\!) \ \triangleright \ e'_1[h/v]$$

$$(\eta_v) \quad \Lambda h.e'_1(\!(h)\!) \ \triangleright \ e'_1 \quad (h \notin EV(e'_1))$$

$$(\mathbf{let}_v) \quad \mathbf{let}_v \ x = e'_1 \ \mathbf{in} \ e'_2 \ \triangleright \ e'_2[x/e'_1]$$

$$(\circ_v) \quad (v_1 \circ v_2)[e'] \ \triangleright \ v_1[v_2[e']]$$

Figure 2.1: Reduction for residual terms

to express specialization features. Evidence is very important in this formulation because it abstracts the differences among the possible specializations of a given source term, and is one of the cornerstones of the principality result.

Definition 2.4 A *residual term*, denoted by e', is an element of the language defined by the following grammar:

$$\begin{array}{rcllllll}
e' & ::= & x' & | \ n & | \ e' + e' & | \ \bullet & & \\
 & | & \lambda x'.e' & | \ e'@e' & | \ \mathbf{let} \ x' = e' \ \mathbf{in} \ e' & & & \\
 & | & (e'_1, \ldots, e'_n) & | \ \pi_{n,n} \, e' & & & & \\
 & | & h & | \ v[e'] & | \ \Lambda h.e' & | \ e'(\!(v)\!) & | \ \mathbf{let}_v \ x = e' \ \mathbf{in} \ e' \\
v & ::= & h & | \ n & | \ C & | \ v \circ v & & \\
C & ::= & [] & | \ \Lambda h.C & | \ C(\!(v)\!) & | \ \mathbf{let}_v \ x = C \ \mathbf{in} \ e' & &
\end{array}$$

We use the constant $\bullet$ as the (only) value of one-point types. Although it is written the same way for every type, it can be seen as a family of values $\bullet_{\tau'}$, a different one for each type τ'.

As presented in section 1.3, h represents evidence variables, $\Lambda h.e'$ represents abstraction and $e'(\!(v)\!)$ represents application.

Two particular kinds of evidence are used: numbers, as evidence for predicates of the form IsInt and $\_ := \_ + \_$, and conversions, as evidence for predicates of the form IsMG. Conversions, denoted by C, are defined as contexts, separately from other elements in the language; the particular forms $v \circ v$ and $\mathbf{let}_v \ x = e' \ \mathbf{in} \ e'$ are used for composition of conversions, necessary for technical reasons.

We work under an equivalence ($=$) relation on residual terms, defined as the smallest congruence containing α-conversions for both λ and Λ-abstractions and the reduction rules appearing in figure 2.1. The spirit of this definition is that operations involving evidence are meant to be solved during specialization (as opposed to regular applications, which are meant to remain in the residual code).

2.2.3 Predicates and entailment relation

The properties relating predicates and evidence are captured by an entailment relation, as described in section 1.3, which satisfies the structural properties established in figure 1.1. The meaning of these predicates is defined by completing the relation with rules that are particular to the system; these are presented in figure 2.2.

The predicate IsInt is provable when the type is a one-point type representing a number, and the evidence is the value of that number. Similarly, the predicate $\_ := \_ + \_$ is provable when

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

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

$$\text{(IsOpIsInt)} \quad \Delta, h : \tau' := \tau'_1 + \tau'_2, \Delta' \Vdash h : \text{IsInt } \tau'$$

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

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

Figure 2.2: Entailment rules for evidence construction

the three arguments are one-point types with the corresponding numbers related by addition, and the evidence is the number corresponding to the result of the addition. The predicate IsMG internalizes the ordering $\geq$ (see section 2.2.4), and the evidence is the corresponding conversion; rule (Comp) captures the transitivity of $\geq$.

2.2.4 Ordering between residual types

As described in section 1.3, the comparison between different types and type schemes can be done by using a "more general" ordering, with a third component: a conversion.

Definition 2.5 Let $\sigma = \forall \alpha_i.\Delta_\tau \Rightarrow \tau$ and $\sigma' = \forall \beta_i.\Delta'_\tau \Rightarrow \tau'$ be two type schemes, and suppose that none of β_i appears free in σ, $\bar{h} : \Delta$, or $\bar{h}' : \Delta'$. A term C is called a *conversion* from $(\Delta \mid \sigma)$ to $(\Delta' \mid \sigma')$, written $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, if and only if there are types τ_i, evidence variables $\bar{h}_\tau$ and $\bar{h}'_\tau$, and evidence expressions $\bar{v}$ and $\bar{v}'$ such that:

- $\tau' = \tau[\alpha_i/\tau_i]$
- $\bar{h}' : \Delta', \bar{h}'_\tau : \Delta'_\tau \Vdash \bar{v} : \Delta, \bar{v}' : \Delta_\tau[\alpha_i/\tau_i]$, and
- $C = (\mathbf{let}_v \; x = \Lambda \bar{h}.[] \; \mathbf{in} \; \Lambda \bar{h}'_\tau.x(\!(\bar{v})\!)(\!(\bar{v}')\!))$

By this definition, if $\Delta = \emptyset$, we would have $C = \mathbf{let}_v \; x = [] \; \mathbf{in} \; \Lambda \bar{h}'_\tau.x(\!(\bar{v}')\!)$, which is equivalent to $\Lambda \bar{h}'_\tau.[](\!(\bar{v}')\!)$

Equivalence is defined for conversions based on the equivalence defined for residual types, stating that $C = C'$ if for all residual expressions e', $C[e'] = C'[e']$.

The most important property of conversions is that they can be used to transform an object e' of type σ under a predicate assignment Δ into an element of type σ' under a predicate assignment Δ', changing only the evidence that appears at the top level of e'.

Example 2.6 Conversions are used to adjust the evidence demanded by different type schemes. For all Δ it holds that

1. $[](\!(11)\!) : (\Delta \mid \forall t.\text{IsInt } t \Rightarrow t \to \mathit{Int}) \geq (\Delta \mid \hat{1}\hat{1} \to \mathit{Int})$

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

$\diamond$

The following propositions are useful for some of our proofs; see Martínez López's presentation ([2005], propositions 6.7 and 6.9) for their proofs.

Proposition 2.7 *The following assertions hold when $\sigma, \sigma', \sigma''$ are not scheme variables:*

1. $[] : (\Delta \mid \sigma) \geq (\Delta \mid \sigma)$
2. *if* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *and* $C' : (\Delta' \mid \sigma') \geq (\Delta'' \mid \sigma'')$ *then*

$$C' \circ C : (\Delta \mid \sigma) \geq (\Delta'' \mid \sigma'')$$

Proposition 2.8 *For any qualified type ρ and predicate assignments $\bar{h} : \Delta$ and $\bar{h}' : \Delta'$,*

1. $\Lambda \bar{h}'.[] : (\Delta, \bar{h}' : \Delta' \mid \rho) \geq (\Delta \mid \Delta' \Rightarrow \rho)$
2. $[](\!(\bar{h}')\!) : (\Delta \mid \Delta' \Rightarrow \rho) \geq (\Delta, \bar{h}' : \Delta' \mid \rho)$
3. *if* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *and* $\bar{h}''' : \Delta''' \Vdash \bar{v}'' : \Delta''$, *then* $C' : (\Delta, \Delta'' \mid \sigma) \geq (\Delta', \Delta''' \mid \sigma')$ *where* $C' = (\mathbf{let}_v\ x = \Lambda \bar{h}'''.C[]\ \mathbf{in}\ x(\!(\bar{v}'')\!))$
4. *if* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *and* $\alpha \notin FV(\Delta, \Delta' \Rightarrow \sigma)$, *then* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \forall \alpha.\sigma')$

2.3 Specifying Principal Specialization

2.3.1 The specialization system: P

System P (for *principal* type specialization) specifies how source terms and types are specialized. Judgments are of the form

$$\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \ \hookrightarrow\ e' : \sigma'$$

meaning source expression e of type τ specializes to residual expression e' with type σ' under the given context. Here Δ is a predicate context, and $\Gamma = \{x_i : \tau_i \hookrightarrow x_i' : \sigma_i \mid i = 1, \ldots, n\}$ is a context mapping source variables and types to residual variables and their type schemes. If any of the contexts is empty, we usually omit it in the specialization judgment.

The rest of this section is dedicated to introducing the rules of the system, illustrating them with examples.

Base types

Specialization of variables, dynamic constants and operators is straightforward — the residual expression is essentially the same, with the annotations removed.

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

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

$$\text{(D+)}\quad \frac{(\Delta \mid \Gamma \vdash_{\mathrm{P}} e_i : \mathit{Int}^D \;\hookrightarrow\; e_i' : \mathit{Int})_{i=1,2}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 +^D e_2 : \mathit{Int}^D \;\hookrightarrow\; e_1' + e_2' : \mathit{Int}}$$

Static constants and operations are meant to be removed from the code into the residual type. Notice the use of predicate $\_ := \_ + \_$ in rule (S+) to make sure τ' is the correct residual type.

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

$$\text{(S+)}\quad \frac{(\Delta \mid \Gamma \vdash_{\mathrm{P}} e_i : \mathit{Int}^S \;\hookrightarrow\; e_i' : \tau_i')_{i=1,2} \quad \Delta \Vdash v : \tau' := \tau_1' + \tau_2'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 +^S e_2 : \mathit{Int}^S \;\hookrightarrow\; \bullet : \tau'}$$

The **lift** operator casts a static expression of integer type into its residual numeric value. Here, the predicate context must entail predicate IsInt, and the evidence is used to get the correct residual value.

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

Rules for dynamic tupling and projection are also straightforward, and analogous to their counterparts in a usual type inference system.

$$\text{(DTUPLE)}\quad \frac{(\Delta \mid \Gamma \vdash_{\mathrm{P}} e_i : \tau_i \;\hookrightarrow\; e_i' : \tau_i')_{i=1,..,n}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} (e_1, \ldots, e_n)^D : (\tau_1, \ldots, \tau_n)^D \;\hookrightarrow\; (e_1', \ldots, e_n') : (\tau_1', \ldots, \tau_n')}$$

$$\text{(DPRJ)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : (\tau_1, \ldots, \tau_n)^D \;\hookrightarrow\; e' : (\tau_1', \ldots, \tau_n')}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \pi^D_{i,n}\, e : \tau_i \;\hookrightarrow\; \pi_{i,n}\, e' : \tau_i'}$$

Notice that even though tuples are dynamic, they can have either static or dynamic components — see below for an example.

Example 2.9 The following are all valid specializations:

1. $\vdash_{\mathrm{P}} 11^D : \mathit{Int}^D \;\hookrightarrow\; 11 : \mathit{Int}$
2. $\vdash_{\mathrm{P}} 11^S : \mathit{Int}^S \;\hookrightarrow\; \bullet : \hat{11}$
3. $\vdash_{\mathrm{P}} \left(2^D +^D 1^D\right) +^D 1^D : \mathit{Int}^D \;\hookrightarrow\; (2+1)+1 : \mathit{Int}$
4. $\vdash_{\mathrm{P}} \left(2^S +^S 1^S\right) +^S 1^S : \mathit{Int}^S \;\hookrightarrow\; \bullet : \hat{4}$
5. $\vdash_{\mathrm{P}} \mathbf{lift}\ \left(2^S +^S 1^S\right) +^D 1^D : \mathit{Int}^D \;\hookrightarrow\; 3+1 : \mathit{Int}$
6. $\left\{x : \mathit{Int}^D \hookrightarrow x' : \mathit{Int}\right\} \vdash_{\mathrm{P}} \left(2^S +^S 1^S, x\right)^D : \left(\mathit{Int}^S, \mathit{Int}^D\right)^D \;\hookrightarrow\; (\bullet, x') : (\hat{3}, \mathit{Int})$

◇

Functions and let

Dynamic λ-expressions are specialized as λ-expressions binding a fresh residual variable to the specialized body.

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

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

Rule (DAPP) is a straightforward extension of the usual typing rule for function application — dynamic applications remain in the residual code as regular applications of the specialized arguments.

Rule (DLAM) has two premises. The first one expresses the specialization of the body assuming the specialization of the bound variable — this is a natural extension of a typing rule for functions. Now if it only had this premise, we would have specializations as the following one:

$$\frac{\left\{x : \mathit{Int}^D \hookrightarrow x' : \mathit{Bool}\right\} \vdash_{\mathrm{P}} x : \mathit{Int}^D \;\hookrightarrow\; x' : \mathit{Bool}}{\vdash_{\mathrm{P}} \lambda^D x.x : \mathit{Int}^D \to^D \mathit{Int}^D \;\hookrightarrow\; \lambda x'.x' : \mathit{Bool} \to \mathit{Bool}}\ (x' \textit{ fresh})$$

That is, we would have the identity function on integers specializing to the identity function on booleans. Clearly this should not be a valid specialization! The problem here is the assumption that a variable of type Int^D can be specialized to a variable of type *Bool*.

The second premise of rule (DLAM) restricts the type τ_2' to be a *reasonable* type for a source variable of type τ_2. The judgment $\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \tau'$ expresses a source-residual relation between two types in a given predicate context, which essentially means it is reasonable to find type τ' in the result of specializing a source expression with type τ. System SR is specified separately — see section 2.3.2 for details.

The rule for specializing dynamic let expressions is also straightforward.

$$\text{(DLET)}\quad \frac{\begin{array}{c}\Delta \mid \Gamma \vdash_{\mathrm{P}} e_2 : \tau_2 \;\hookrightarrow\; e_2' : \tau_2' \\ \Delta \mid \Gamma, x : \tau_2 \hookrightarrow x' : \tau_2' \vdash_{\mathrm{P}} e_1 : \tau_1 \;\hookrightarrow\; e_1' : \tau_1'\end{array}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{let}^D\ x = e_2\ \mathbf{in}\ e_1 : \tau_1 \;\hookrightarrow\; \mathbf{let}\ x' = e_2'\ \mathbf{in}\ e_1' : \tau_1'}\ (x' \textit{ fresh})$$

Here there is no need to restrict types τ_2 and τ_2' individually, since they are related by the specialization of e_2 to e_2'.

Example 2.10 We revisit example 1.2 to show how the residual expression can be derived. The $\vdash_{\mathrm{SR}}$ judgments needed to apply rule (DLAM) are easily verified — see section 2.3.2.

1. $\left\{f : \mathit{Int}^S \to^D \mathit{Int}^S \hookrightarrow f' : \hat{3} \to \hat{4}\right\} \vdash_{\mathrm{P}} f \mathbin{@^D} 3^S : \mathit{Int}^S \;\hookrightarrow\; f' @ \bullet : \hat{4}$
2. $\left\{f : \mathit{Int}^S \to^D \mathit{Int}^S \hookrightarrow f' : \hat{3} \to \hat{4}\right\} \vdash_{\mathrm{P}} \mathbf{lift}\ \left(f \mathbin{@^D} 3^S\right) : \mathit{Int}^D \;\hookrightarrow\; 4 : \mathit{Int}$
3. $\vdash_{\mathrm{P}} \lambda^D f.\mathbf{lift}\ \left(f \mathbin{@^D} 3^S\right) : \left(\mathit{Int}^S \to^D \mathit{Int}^S\right) \to^D \mathit{Int}^D \;\hookrightarrow\; \lambda^D f'.4 : \left(\hat{3} \to \hat{4}\right) \to \mathit{Int}$

4. $\{x : Int^S \hookrightarrow x' : \hat{3}\} \vdash_{\mathrm{P}} x +^S 1^S : Int^S \hookrightarrow \bullet : \hat{4}$
5. $\vdash_{\mathrm{P}} \lambda^D x.x +^S 1^S : Int^S \to^D Int^S \hookrightarrow \lambda x'.\bullet : \hat{3} \to \hat{4}$
6. $\vdash_{\mathrm{P}} (\lambda^D f.\mathbf{lift}\ (f\ @^D\ 3^S))\ @^D\ (\lambda^D x.x +^S 1^S) : Int^D \hookrightarrow (\lambda f'.4)\,@\,(\lambda x'.\bullet) : Int$

◇

Example 2.11 Terms can have more than one valid specialization.

1. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x : Int^S \to^D Int^D \hookrightarrow \lambda x'.3 : \hat{3} \to Int$
2. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x : Int^S \to^D Int^D \hookrightarrow \lambda x'.n : \hat{n} \to Int \quad \forall n \in \mathbf{Z}$
3. $\vdash_{\mathrm{P}} \lambda^D x.x +^S 1^S : Int^S \to^D Int^S \hookrightarrow \lambda x'.\bullet : \hat{1}\hat{1} \to \hat{1}\hat{2}$
4. $\vdash_{\mathrm{P}} \lambda^D x.x +^S 1^S : Int^S \to^D Int^S \hookrightarrow \lambda x'.\bullet : \hat{n} \to n \hat{+} 1 \quad \forall n \in \mathbf{Z}$

◇

Qualified types and type schemes

Just as the system of qualified types [Jones, 1994], system P includes structural rules to move predicates from the context into the residual type, and conversely, to eliminate them from the type if they can be proved by the context. The structure of the residual term changes accordingly by means of evidence abstraction and application.

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

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

Generalization and instantiation of type schemes is specified in the same way as in a type inference system.

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

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

Example 2.12 By the rules above, properly constrained type variables can be introduced to obtain a principal specialization when more than one is possible. Evidence for predicate IsInt is used to obtain the correct residual code (see rule (LIFT)) and then abstracted to form the qualified type. Instances of a general type scheme can be obtained by rule (INST) and (QOUT), with the corresponding evidence application.

1. $\{h_t : \mathrm{IsInt}\ t\} \vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x : Int^S \to^D Int^D \hookrightarrow \lambda x'.h_t : t \to Int$

2. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x : \mathit{Int}^S \to^D \mathit{Int}^D \ \hookrightarrow\ \Lambda h_t.\lambda x'.h_t : \mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int}$
3. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x : \mathit{Int}^S \to^D \mathit{Int}^D \ \hookrightarrow\ \Lambda h_t.\lambda x'.h_t : \forall t.\mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int}$
4. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x : \mathit{Int}^S \to^D \mathit{Int}^D \ \hookrightarrow\ \Lambda h_t.\lambda x'.h_t : \mathrm{IsInt}\ \hat{1}1 \Rightarrow \hat{1}1 \to \mathit{Int}$
5. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x : \mathit{Int}^S \to^D \mathit{Int}^D \ \hookrightarrow\ (\Lambda h_t.\lambda x'.h_t)\,(\!(11)\!) : \hat{1}1 \to \mathit{Int}$
6. $\vdash_{\mathrm{P}} \lambda^D x.\mathbf{lift}\ x : \mathit{Int}^S \to^D \mathit{Int}^D \ \hookrightarrow\ \lambda x'.11 : \hat{1}1 \to \mathit{Int}$

The residual terms of items 5 and 6 are equivalent based on the definition in section 2.2.2. $\diamond$

Polyvariance

So far, the rules specify a *monovariant* form of specialization — that is, one in which static variables can take only one static value. For example, the expression

$$\mathbf{let}^D\ \ f = \lambda^D x.\mathbf{lift}\ x\ \mathbf{in}\ f\ @^D\, 11^S\ : \mathit{Int}^D$$

can be specialized to

$$\mathbf{let}\ f' = \lambda x'.11\ \mathbf{in}\ f'@\bullet : \mathit{Int}$$

But the similar expression

$$\mathbf{let}^D\ \ f = \lambda^D x.\mathbf{lift}\ x\ \mathbf{in}\ \left(f\ @^D\, 11^S, f\ @^D\, 6^S\right)^D\ : \left(\mathit{Int}^D, \mathit{Int}^D\right)^D$$

cannot be specialized, because f' cannot have both types $\hat{1}1 \to \mathit{Int}$ and $\hat{6} \to \mathit{Int}$. A useful partial evaluator must use polyvariant specialization, generating in this case at least two versions of f': one for each static argument.

Operator **poly** is introduced to produce polyvariant specializations. An expression enclosed by **poly** is specialized to a general residual type (also wrapped in a **poly** annotation) that can be instantiated in each use. Operator **spec** produces a suitable instantiation. Rules (POLY) and (SPEC) specify these operators, using predicate IsMG and proper evidence to capture the relationship between the general type and its instances.

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

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

Rule (POLY) describes how an expression e can be made polyvariant. If it has residual type σ', then for each specialization of e we are going to use, its residual type must be an adequate instance of σ'. In other words, σ' is an *upper bound* for e''s polyvariant type. However, the type is not necessarily **poly** σ', because it could be furthered constrained by the context (for example, e could be the argument of a function expecting a more restricted residual type than σ'). So for any type σ, as long as it is constrained by this upper bound, **poly** σ is a valid type. Predicate IsMG reflects this; observe how evidence v (a conversion, in this case) is used to obtain a suitable expression of type σ from e'.

Rule (SPEC) also uses predicate IsMG and its evidence, in this case to ensure a correct instantiation of the polyvariant type. Here, τ' must be a proper instance of σ, so it constitutes a *lower bound* for the type scheme. Also τ and τ' must be in the source-residual relation, for reasons analogous to those of rule (DLAM).

Example 2.13 Function f in the expression above can now be made polyvariant, so it can be applied to more than one static argument. As it no longer has a function type, it cannot be applied directly — instead, it must be **spec**'ed first.

$$\begin{array}{ll} \vdash_{\mathrm{P}} & \mathbf{let}^D\ f = \mathbf{poly}\ \lambda^D x.\mathbf{lift}\ x \\ & \quad \mathbf{in}\ \left(\mathbf{spec}\ f\ @^D\ 11^S, \mathbf{spec}\ f\ @^D\ 6^S\right)^D : \left(Int^D, Int^D\right)^D \ \hookrightarrow \\ & \mathbf{let}\ f' = \Lambda h.\lambda x'.h \\ & \quad \mathbf{in}\ (f'(\!(11)\!)@\bullet, f'(\!(6)\!)@\bullet) : (Int, Int) \end{array}$$

In this specialization, the type of f' is $\mathbf{poly}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to Int)$, and h is the evidence for predicate IsInt. In the first component of the tuple, $\mathbf{spec}\ f$ must have residual type $\hat{1}1 \to Int$. Rule (SPEC) makes sure this is an instance of $\forall t.\mathrm{IsInt}\ t \Rightarrow t \to Int$, and takes the proper evidence (here, conversion $[](\!(11)\!)$) to form the residual expression. The derivation of the second component is analogous. ◇

Example 2.14 To obtain the principal specialization when static information is missing, type scheme variables and evidence abstractions can be used.

1. $\vdash_{\mathrm{P}}\ \mathbf{poly}\ \left(\lambda^D x.\mathbf{lift}\ (x +^S 1^S)\right) : \mathbf{poly}\ \left(Int^S \to^D Int^D\right) \ \hookrightarrow$
   $\Lambda h^u_s.h^u_s[\Lambda h'_t, h_t.\lambda x'.h_t] :$
   $\forall s.\mathrm{IsMG}\ \left(\forall t, t'.\mathrm{IsInt}\ t', t := t' + \hat{1} \Rightarrow t \to Int\right)\ s \Rightarrow \mathbf{poly}\ s$

2. $\vdash_{\mathrm{P}}\ \lambda^D f.\mathbf{spec}\ f\ @^D\ 11^S : \mathbf{poly}\ \left(Int^S \to^D Int^D\right) \to^D Int^D \ \hookrightarrow$
   $\Lambda h^u.\Lambda h^l.\lambda f'.h^l[f']@\bullet : \forall s.\ \mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to Int)\ s,$
   $\mathrm{IsMG}\ s\ \left(\hat{1}1 \to Int\right)$
   $\Rightarrow \mathbf{poly}\ s \to Int$

In the first case, evidence h^u_s represents a conversion proving that $\forall t.\mathrm{IsInt}\ t \Rightarrow t \to Int$ is an upper bound for the polyvariant type. In the second case, the upper bound constrains the type of f' to respect the source type (which is part of the input). The lower bound establishes that f' must be applied to a value of type $\hat{1}1$, and h^l represents a conversion adjusting f' for this application. ◇

2.3.2 Source-Residual relation: system SR

As we have seen in rules (DLAM) and (SPEC), system P is based on an auxiliary system that describes a *source-residual* relation: system SR. Judgments are of the form

$$\Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \sigma'$$

meaning residual type σ' can be obtained from a source type τ under predicate context Δ.

Rules to derive this judgment are presented in figure 2.3. There is one rule for each source type, plus the four structural rules to introduce and eliminate predicates and universal quantification. They are all quite straightforward, and naturally similar to the corresponding rules in system P.

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

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

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

$$\text{(SR-TUPLE)}\quad \frac{(\Delta \vdash_{\text{SR}} \tau_i \hookrightarrow \tau_i')_{i=1,..,n}}{\Delta \vdash_{\text{SR}} (\tau_1, \ldots, \tau_n)^D \hookrightarrow (\tau_1', \ldots, \tau_n')}$$

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

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

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

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

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

Figure 2.3: Rules defining the source-residual relation

2.3.3 Typing residual terms: system RT

A system for type-checking residual terms is given in section A.1. The rules are completely analogous to those in the specialization system. Martínez López proves specialization is well-behaved with respect to system RT [Martínez López, 2005], thus proving the residual terms are indeed typed.

2.4 Existence of a Principal Type Specialization

Martínez López extends the notion of *principal type scheme*, originally introduced in the study of combinatory logic [Curry and Feys, 1958; Hindley, 1969] and further studied by Damas and Milner [1982], and Mark Jones in the theory of qualified types, to a similar result for type specialization.

Definition 2.15 A *principal type specialization* of a source term e of type τ under the specialization assignment Γ is a residual term e'_p of type σ_p such that $\Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e'_p : \sigma_p$ and it is the case that for every $\Delta' \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$ there exist a conversion C and a substitution S satisfying $C : S\,\sigma_p \geq (\Delta' \mid \sigma)$ and $C[e'_p] = e'$.

One of system P's main properties is that of principality. That is, for each specialization assignment Γ and source term $e : \tau$, if there is a specialization of e under Γ, then there is a principal specialization of e under Γ.

The proof of this result follows the lines of the proof of principality for the theory of qualified types [Jones, 1994], which is constructive — it consists in showing an algorithm that computes principal specializations and fails when none exists. The proof proceeds in two steps. First, system P is transformed into an intermediate syntax-directed system S, which is

proved to be equivalent to system P in an appropriate way[1]. Then, the algorithm —described by system W— is introduced, proving that it is equivalent to system S. The complete system is presented in section A.2.1.

The *generalization* of a qualified type ρ with respect to predicate and specialization assignments Δ and Γ, noted as $\mathrm{Gen}_{\Delta,\Gamma}(\rho)$, is defined as $\forall\bar{\alpha}_i.\rho$ where $\bar{\alpha}_i$ is the set of type and type scheme variables $FV(\rho) \setminus (FV(\Delta) \cup FV(\Gamma))$. When Δ is the empty set we also note $\mathrm{Gen}_{\Gamma}(\rho)$.

Generalizations have some useful properties that we will use in some of our proofs.

Proposition 2.16 *The relation $\geq$ satisfies that, for all Γ and τ',*

1. *If* $\bar{h}' : \Delta' \Vdash \bar{v} : \Delta$ *and* $C = [](\bar{v})$
   then $C : \mathrm{Gen}_{\Gamma,\Delta''}(\Delta \Rightarrow \tau') \geq (\bar{h}' : \Delta' \mid \tau')$

2. *If* $\bar{h}' : \Delta' \Vdash \bar{v} : \Delta$ *and* $C = \Lambda\bar{h}'.[](\bar{v})$
   then $C : \mathrm{Gen}_{\Gamma,\Delta''}(\Delta \Rightarrow \tau') \geq \mathrm{Gen}_{\Gamma,\Delta''}(\Delta' \Rightarrow \tau')$

3. *for all substitutions R and all contexts Δ,*
   $[] : R\,\mathrm{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau') \geq \mathrm{Gen}_{R\,\Gamma,R\,\Delta'}(R\,\Delta \Rightarrow R\,\tau')$
   Furthermore, there is a substitution T such that $T\,\Gamma = R\,\Gamma$, $T\,\Delta' = R\,\Delta'$ and
   $$R\,\mathrm{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau') = \mathrm{Gen}_{T\,\Gamma,T\,\Delta'}(T\,\Delta \Rightarrow T\,\tau')$$

The rules describing the principal type specialization algorithm depend on a number of auxiliary subsystems which can be summarized as follows:

Unification Unification is based on Robinson's algorithm, with modifications to deal with substitution under quantification (which can happen inside a polyvariant residual type). The algorithm is specified by a system of rules to derive judgments of the form $\sigma_1 \sim^U \sigma_2$ — see details in section A.2.2. They can be interpreted as a partial function taking two residual types and returning a most general unifier, if it exists.

Entailment The entailment algorithm ($\Vdash_{\mathrm{W}}$) takes a target predicate δ and a current predicate assignment Δ, and calculates a set of predicates that should be added to Δ in order to entail δ. The result can be easily achieved by adding δ to Δ with a new evidence variable h. So the only rule necessary for specifying this algorithm is

$$h : \delta \mid \Delta \Vdash_{\mathrm{W}} h : \delta \quad (h \textit{ fresh})$$

More refined algorithms can be designed — for example, to handle ground predicates (such as IsInt $\hat{n}$) or predicates already appearing in Δ, but all these cases can be handled by the phase of simplification and constraint solving [Martínez López and Badenes, 2003].

Source-Residual relation Algorithm W-SR computes the source-residual relation between types. Given a source type τ, it returns a set of predicates Δ and a residual type τ' such that $\mathrm{Gen}_{\emptyset,\emptyset}(\Delta \Rightarrow \tau')$ is the most general type scheme SR-related to τ. The rules defining the system are presented in section A.2.3.

[1] Actually, system S is not strictly syntax-directed, since some of the rules are based on the SR system, which remains unchanged (and is not syntax-directed). However, it is still useful, since it is only introduced as an intermediate step toward the algorithm, making the proofs simpler.

2.5 Extension: Tagged Sum Types

The language considered so far is a small subset of a real programming language. A number of extensions have been added to the system [Martínez López, 2005], including booleans, static functions and let expressions, static recursion and program failure. Most relevant to our contribution is the addition of tagged sum types. Martínez López incorporates only static treatment of data types, and later Alejandro Russo [Russo, 2004] adds their dynamic version. In this section, we describe both extensions.

2.5.1 Source language

The source language now includes data declarations resembling those of Haskell, datatype constructors and **case** expressions. Datatypes are named and have no parameters. Constructors are distinguished lexically and take only one argument.

Definition 2.17 Let D denote a sum type name and K a constructor name. A *source term*, denoted by e, is an element of the language defined by the following grammar:

$$
\begin{array}{lll}
e & ::= & [\ ddcl\ ]^*\ e_p \\
ddcl & ::= & \mathbf{data}\ D^{n\,S}\ =\ cs_s \mid \mathbf{data}\ D^{D}\ =\ cs_d \\
cs_s & ::= & K_1^{\ S}\ \tau \mid\mid \ldots \mid\mid K_n^{\ S}\ \tau \\
cs_d & ::= & K_1^{\ D}\ \tau \mid\mid \ldots \mid\mid K_n^{\ D}\ \tau \\
e_p & ::= & \ldots \\
 & & \mid\ K^{\ S}\ e_p \mid K^{\ D}\ e_p \\
 & & \mid\ \mathbf{case}^S\, e_p\ \mathbf{of}\ [\ br_s\ ]^+ \\
 & & \mid\ \mathbf{case}^D\, e_p\ \mathbf{of}\ [\ br_d\ ]^+ \\
br_s & ::= & K^S\ x\ \rightarrow\ e_p \\
br_d & ::= & K^D\ x\ \rightarrow\ e_p
\end{array}
$$

where e_p extends the grammar describing source terms in definition 2.1 with two extra constructs in their static and dynamic versions.

Source types are extended with a new family of types — namely, the sum types defined in the data declarations, both static and dynamic.

Definition 2.18 A *source type*, denoted by τ, is an element of the language defined by the following grammar:

$$\tau ::= Int^D \mid Int^S \mid (\tau, \ldots, \tau)^{\,D} \mid \tau \rightarrow^D \tau \mid \mathbf{poly}\ \tau \mid D^S \mid D^D$$

where D is any element of a set of datatype names beginning with a capital letter and different from any already existing name, like *Int*, etc.

Expressions involving static and dynamic sum types differ in the amount of information that is obtained during specialization and moved into the residual type.

Constructors Specializing a constructor $K^{\,S}$ with argument $e : \tau$ yields e' of type $K\ \sigma'$, where e' and σ' are the residual term and type corresponding to e, and the constructor has been moved into the residual type. If $K^{\,D}$ is dynamic, it is meant to be kept in the residual code, so specialization yields $K\ e'$.

Case expressions Static case expressions are computed during specialization, that is, the correct branch is selected and replaces the case expression, which is completely removed from the residual code. Dynamic case expressions are preserved, but the control expression and branches are of course specialized individually. Moreover, only the *useful* branches are preserved. For example, if constructor K does not appear at all on the residual code, then the branch matching the control expression with $K\ x$ can be safely removed.

Data declarations The name of a static sum type does not appear at all in the residual code, so there is no data declaration for it. Dynamic sum types do have a declaration, where the type argument of each constructor has been specialized and some of the constructors removed (in the same way as case branches). Additional flexibility is introduced by allowing a single source-type declaration to yield more than one residual datatype — if the same constructor appears in completely unrelated expressions, each can be specialized to different constructors belonging to different residual types.

2.5.2 Residual language

In order to achieve the behavior described above, the residual language must be extended with new constructs for terms, evidence expressions, types and new predicates.

Definition 2.19 Let D denote a sum type name and K a constructor name. A *residual term*, denoted by e', is an element of the language defined by the following grammar:

$$
\begin{array}{lll}
e' & ::= & [\ ddcl'\ ]^*\ e'_p \\
ddcl' & ::= & \textbf{data}\ D^n\ =\ cs' \\
cs' & ::= & K_1^{v'_p}\ \tau\ ||\ \ldots\ ||\ K_n^{v'_p}\ \tau \\
e'_p & ::= & \ldots \\
 & & |\ K\ e'_p \\
 & & |\ \textbf{case}\ e'_p\ \textbf{of}\ [\ br'_d\ ]^+ \\
 & & |\ \textbf{case}_v\ v'_p\ \textbf{of}\ [\ br'_s\ ]^+ \\
 & & |\ \textbf{protocase}_v\ e'_p\ \textbf{with}\ v'_p\ \textbf{of}\ [\ br'_d\ ]^+ \\
br'_s & ::= & K\ \rightarrow\ e'_p \\
br'_d & ::= & K\ x\ \rightarrow\ e'_p \\
v'_p & ::= & \ldots\ |\ K\ |\ \bullet\ |\ \{K_k\ \}_{k\in I} \\
 & & |\ \textbf{if}_v\ K \in v'_p\ \textbf{then}\ v'_p\ \textbf{else}\ v'_p
\end{array}
$$

Dynamic source code yields equivalent constructs in the residual code: data declarations are added to the residual language, as well as tagged values and case expressions. The superscripts in the data declarations allow distinguishing among all the possible data types generated by a single source declaration.

Example 2.20 The same source datatype can be specialized to many different residual data types if constructors appear independently. Under the following declaration

$$data\ D^D\ =\ Only^D\ Int^S$$

the source expression

$$\left(Only^D\ 11^S\,,\, Only^D\ 4^S\right)^{\,D}\ :\ \left(D^D\,,\, D^D\right)^{\,D}$$

has two appearances of constructor *Only*, but they need not belong to the same residual datatype. So the specialized code has two data declarations

$$\begin{array}{l} data\ D^1 = Only^1\ \hat{1}1 \\ data\ D^2 = Only^2\ \hat{4} \end{array}$$

and expression

$$(Only^1\ \bullet, Only^2\ \bullet) : (D^1, D^2)$$

◇

A v subscript in expressions indicates they may be reduced during specialization. Static case expressions must be removed from the code, choosing the appropriate branch based on the type of the control expression. However, in principal type specialization, we must be able to specialize it independently, even if there is some information missing. The $\mathbf{case}_v$ construct can be reduced as soon as this information is available.

Example 2.21 Assuming the declaration

$$data\ Either^S = Left^S\ Int^D \mid Right^S\ Int^D$$

the source expression

$$\begin{array}{lll} \mathbf{case}^S\ Left^S\ 11^D\ \mathbf{of} & Left^S\ x & \rightarrow\ x \\ & Right^S\ y & \rightarrow\ y +^D 10^D \end{array}$$

specializes to

$$\begin{array}{lll} \mathbf{case}_v\ Left\ \mathbf{of} & Left & \rightarrow\ 11 \\ & Right & \rightarrow\ 11 + 10 \end{array}$$

Since the information for choosing the correct branch is available, the expression above can be reduced to $11 : Int$. ◇

Following the same line is the $\mathbf{protocase}_v$ construct. It is used for dynamic case expressions, where the residual code only keeps the branches that can actually occur. Since this information might not be available, the result of specializing a dynamic case expression is a $\mathbf{protocase}_v$ construct, which may be reduced to a regular **case** expression. Similarly, $\mathbf{if}_v$ constructs are reduced only when the first argument is a set of constructors, and it can be decided whether it includes one in particular or not. Reduction rules are presented in figure 2.4.

The new constructs introduce new evidence expressions. A constructor name K appears in $\mathbf{case}_v$ expressions, both as the control expression and in the branches. Void evidence ($\bullet$) is meant to appear only inside the body of branches that are not actually selected. A set of constructors $\{K_k\}_{k \in I}$ is used in the $\mathbf{protocase}_v$ construct to indicate which constructors of a given datatype can occur. See examples of the use of evidence in section 2.5.3.

New residual types include constructor types (coming from static tagged expressions), sum types (from dynamic tagged expressions) and new predicates. The definition of residual types is extended as follows.

$$\mathbf{case}_v\ K_k\ \mathbf{of}\ \left(K_j\ \rightarrow\ e'_j\right)_{j\in B}\ \triangleright\ e'_k \quad (k\in B)$$

$$\mathbf{protocase}_v\ e'\ \mathbf{with}\ \{K_k\}_{k\in I}\ \mathbf{of}\ \left(K_j\ x'_j\ \rightarrow\ e'_j\right)_{j\in B}\ \triangleright\ \mathbf{case}\ e'\ \mathbf{of}\ \left(K_j\ x'_j\ \rightarrow\ e'_j\right)_{j\in(I\cap B)}$$

$$\mathbf{if}_v\ K_j \in \{K_k\}_{k\in I}\ \mathbf{then}\ v_1\ \mathbf{else}\ v_2\ \triangleright\ v_1 \quad (K_j\in\{K_k\}_{k\in I})$$

$$\mathbf{if}_v\ K_j \in \{K_k\}_{k\in I}\ \mathbf{then}\ v_1\ \mathbf{else}\ v_2\ \triangleright\ v_2 \quad (K_j\notin\{K_k\}_{k\in I})$$

Figure 2.4: Reduction for residual terms involving sum types

Definition 2.22 Let t denote a *type variable* from a countably infinite set of variables and s a *type scheme variable* from another countably infinite set of variables, all of them disjoint with any other set of variables already used. Let D and K be datatype and constructor names respectively. A *residual type*, denoted by τ', is an element of the language given by the grammar

$$\begin{array}{rcl}
\tau' & ::= & t \mid \mathit{Int} \mid \hat{n} \mid \tau' \rightarrow \tau' \mid (\tau', \ldots, \tau') \mid \mathbf{poly}\ \sigma \mid K\ \tau' \mid D^n \\
\rho & ::= & \delta \Rightarrow \rho \mid \tau' \\
\sigma & ::= & s \mid \forall s.\sigma \mid \forall t.\sigma \mid \rho \\
\delta & ::= & \mathrm{IsInt}\ \tau' \mid \tau' := \tau' + \tau' \mid \mathrm{IsMG}\ \sigma\ \sigma \mid \tau \sim \tau' \mid \delta_s \mid \delta_d \\
\delta_s & ::= & \mathrm{IsConstrOf}\ D\ \tau' \\
 & \mid & \tau' := \mathbf{case}\ \tau''\ \mathbf{of}\ (K_j\ \rightarrow\ \tau'_j)_{j=1\ldots n} \\
 & \mid & (v\ \text{is a}\ K)\,?\,\delta \\
\delta_d & ::= & \mathrm{IsSum}\ \tau' \\
 & \mid & \mathrm{HasC}\ \tau'\ K\ \tau' \\
 & \mid & K \in \tau'\,?\,\delta
\end{array}$$

The new predicates are indicated by δ_s for predicates involving static sum types and δ_d for dynamic ones, except for the unification predicate $\tau \sim \tau'$ that is used for both. They express relationships involving residual types generated by a sum type or by one of its constructors. Intuitively, they have the following meaning:

- $\tau \sim \tau'$ is true whenever τ and τ' unify.
- IsConstrOf $D\ \tau'$ is true when D is defined as a static sum type, τ' is a constructor-type $K\ \tau''$ and K is in D's definition.
- $\tau' := \mathbf{case}\ \tau''\ \mathbf{of}\ (K_j\ \rightarrow\ \tau'_j)_{j=1\ldots n}$ is true when τ'' is a constructor-type with tag K_j for an integer j between 1 and n, and τ' is τ'_j.
- IsSum τ' is true when τ' is a residual sum type.
- HasC $\tau'\ K\ \tau''$ is true when τ' is a residual sum type that has constructor $K\ \tau''$ in its definition.

Guarded predicates are of the form *condition*$\,?\,\delta$, where δ is a predicate that could also appear by itself. Essentially, a guarded predicate is meant to be proved only if the condition is satisfied, or in other words, it can be proved trivially if it is not.

The meaning of the predicates is formalized in the entailment relation, which is presented in figure 2.5. Observe how the meaning of each condition in the guarded predicates is defined implicitly, by stating when the target predicate can be proved trivially and when it cannot.

New notation is used in this figure. For a residual sum type declaration

$$data\ D^n = K_1^n\ \tau_1' \mid \ldots \mid K_j^n\ \tau_j'$$

we define $D^n(K_i)$ to be τ_i', that is, the argument of the constructor K_i^n . Similarly, we define $D^S(K)$ or $D^D(K)$ as the argument of constructor K in the source declaration of D.

The relation also specifies the evidence proving the predicates, where it is relevant. In particular:

- IsConstrOf $D^S\ \ \tau'$ can be proved if τ' is a constructor-type belonging to D^S's definition, and the evidence is the name of the constructor.
- IsSum τ' can be proved when τ' is a residual sum type, and the evidence is the set of constructors in its declaration.
- HasC $\tau'\ \ K_j\ \ \tau''$ can be proved only if τ' is a residual sum type D^n and $K_j^n\ \tau''$ is in its declaration. The evidence is n, the number of the specialized version of D^D where the constructor appears with the corresponding argument.
- Rule (IsSum-Guard) shows the evidence for a conditional predicate when the information regarding the condition is missing. The decision is deferred by an $\mathbf{if}_v$ construct that may be reduced as soon as the decision can be made.

2.5.3 Specialization rules

Residual typing

System RT is extended to type the new residual terms — see section A.3.

Source-residual relation

In his extension, Martínez López [2005] does not include a source-residual rule for static sum types. It can be formulated straightforwardly but, since it is not of much relevance to our work, we do not include it here.

Dynamic sum types do have a new rule for system SR, presented in figure 2.6. Here, $D^D(K_j)$ is the source argument of the constructor in the data definition. Rule (SR-DDATA) states what residual types can be obtained from a dynamic sum type. It specifies that a type τ' can be generated from a source type D^D if:

1. τ' can be proved to be a sum type;
2. For every K_j in τ's definition, *if it is present in the residual version*, then it has a residual argument τ_j' that is SR-related to the source argument.

Note the use of conditional predicates so that the second condition is applied only to the constructors actually present in τ''s declaration. For those that are not, all the premises can be proved trivially — see entailment rules in figure 2.5.

$$\text{(IsConstr)}\quad \frac{\Delta \vdash_{\text{SR}} D^S\,(K)\ \hookrightarrow\ \tau'_k}{\Delta \Vdash K : \text{IsConstrOf}\ D^S\ \ (K\ \tau'_k)}$$

$$\text{(IsConstr-True)}\quad \frac{\Delta \Vdash v : \delta}{\Delta \Vdash v : (K_k \text{ is a } K_k)\,?\,\delta}$$

$$\text{(IsConstr-False)}\quad \Delta \Vdash \bullet : (K_k \text{ is a } K_j)\,?\,\delta \quad (k{\neq}j)$$

$$\text{(Case)}\quad \Delta \Vdash \bullet : \tau'_k := \textbf{case}\ (K_k\ \tau'')\ \textbf{of}\ \left(K_j\ \rightarrow\ \tau'_j\right)_{j\in I}$$

$$\text{(IsSum)}\quad \frac{D^n \text{ is defined as } \{K_k^n\ \tau'_k\}_{k\in I}}{\Delta \Vdash \{K_k\ \}_{k\in I} : \text{IsSum}\ D^n}$$

$$\text{(HasC)}\quad \frac{\Delta \Vdash D^n(K_j) \sim \tau'}{\Delta \Vdash n : \text{HasC}\ D^n\ \ K_j\ \ \tau'}$$

$$\text{(HasC-True)}\quad \frac{K_j \in D^n \quad \Delta \Vdash v : \Delta'}{\Delta \Vdash v : K_j \in D^n\,?\,\Delta'}$$

$$\text{(HasC-False)}\quad \frac{K_j \notin D^n}{\Delta \Vdash \bullet : K_j \in D^n\,?\,\Delta'}$$

$$\text{(Unify-HasC)}\quad \frac{\Delta, h : \text{HasC}\ \tau'\ \ K_j\ \ \tau'_1, \Delta' \Vdash \tau'_1 \sim \tau'_2}{\Delta, h : \text{HasC}\ \tau'\ \ K_j\ \ \tau'_1, \Delta' \Vdash h : \text{HasC}\ \tau'\ \ K_j\ \ \tau'_2}$$

$$\text{(HasC-Guard)}\quad \frac{\Delta, \text{HasC}\ \tau'\ \ K_j\ \ \tau'', \Delta' \Vdash v : \Delta''}{\Delta, \text{HasC}\ \tau'\ \ K_j\ \ \tau'', \Delta' \Vdash v : K_j \in \tau'\,?\,\Delta''}$$

$$\text{(IsSum-Guard)}\quad \frac{\bar{h} : \Delta \Vdash \bar{v}' : \Delta' \quad \Delta'' \Vdash v^d : \text{IsSum}\ \tau'}{\Delta'', \bar{h} : K_j \in \tau'\,?\,\Delta \Vdash \textbf{if}_v\ K_j \in v^d\ \textbf{then}\ \bar{v}'\ \textbf{else}\ \bullet : K_j \in \tau'\,?\,\Delta'}$$

Figure 2.5: Entailment rules for predicates involving sum types

$$\text{(SR-DDATA)}\quad \frac{\Delta \Vdash \text{IsSum } \tau' \quad \left(\begin{array}{l} \Delta_j \vdash_{\text{SR}} D^D\,(K_j) \;\hookrightarrow\; \tau'_j \\ \Delta \Vdash K_j \in \tau' \,?\, \Delta_j \\ \Delta \Vdash K_j \in \tau' \,?\, \text{HasC } \tau' \;\; K_j \;\; \tau'_j \end{array}\right)_{K_j \in D}}{\Delta \vdash_{\text{SR}} D^D \;\hookrightarrow\; \tau'}$$

Figure 2.6: Source-residual relation for dynamic sum types

$$\text{(SCONSTR)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} x : \tau \;\hookrightarrow\; x' : \tau'}{\Delta \mid \Gamma \vdash_{\text{P}} K_j{}^S\; x : D^S \;\hookrightarrow\; x' : K_j\; \tau'}$$

$$\text{(SCASE)}\quad \frac{\begin{array}{l} \Delta \mid \Gamma \vdash_{\text{P}} e : D^S \;\hookrightarrow\; e' : \tau'_e \\ \Delta \Vdash v : \text{IsConstrOf } D^S \;\; \tau'_e \\ \left(\begin{array}{l} \bar{h}_j : \Delta_j \mid \Gamma \vdash_{\text{P}} \lambda^S x_j.e_j : D^S\,(K_j) \to^S \tau \;\hookrightarrow\; e'_j : \tau'_j \\ \bar{h}_j : \Delta_j \Vdash w_j : \text{IsFunS } \tau'_j\; \mathbf{clos}(\tau'_{e'j} : \tau'_{j2} \to \tau'_{j1}) \\ \bar{h}_j : \Delta_j \vdash_{\text{SR}} \tau \;\hookrightarrow\; \tau'_{j1} \\ \Delta \Vdash \bar{v}_j : v \text{ is a } K_j \,?\, \Delta_j \end{array}\right)_{j \in B} \\ \Delta \Vdash \tau'_r := \mathbf{case}\; \tau'_e\; \mathbf{of}\; (K_j \;\to\; \tau'_{j1})_{j \in B} \end{array}}{\begin{array}{ll} \Delta \mid \Gamma \vdash_{\text{P}} & \mathbf{case}^S\, e\; \mathbf{of} \\ & \left(K_j{}^S\; x_j \;\to\; e_j\right)_{j \in B} : \tau \end{array} \;\hookrightarrow\; \begin{array}{l} \mathbf{case}_v\; v\; \mathbf{of} \\ \left(K_j \;\to\; (w_j @_v e'_j @_v e')[\bar{v}_j/\bar{h}_j]\right)_{j \in B} : \tau'_r \end{array}}$$

Figure 2.7: Specialization rules involving static sum types

System P

The specialization rules involving sum types are presented in figures 2.8 and 2.7. We begin by explaining specialization of static sum types and then do the same thing for dynamic ones.

Specializing static sum types Specialization of static sum types is based on specialization of static functions, a feature added as another extension in Martínez López's presentation [2005, chapter 9] that we do not present here. Instead, we explain the meaning of the predicates and constructs in the context of the specialization rules of our interest.

In static tagged expressions, the constructors are η-expanded and considered as static functions — $K^S\; e$ is a shortcut for $(\lambda^S x.K^S\; x)\; @^S\, e$. Static λ-expressions do not generate a function in the residual code; instead they are unfolded and applied during specialization. With this in mind, it is only necessary to specify how tagged *variables* are specialized. Rule (SCONSTR) states a tagged source variable specializes to a residual variable, where the tag is moved from the code into the type.

Specializing a case expression involves specializing the control expression, knowing which summand it lies in, and statically choosing the corresponding branch. Rule (SCASE) specifies this modularly and preserving principality.

The first two premises involve the specialization of the control expression $e : D^S$ — it must specialize to an expression with a residual type τ'_e that can be proved to be a constructor-type from D's definition.

The following premises involve specializing a branch; let us suppose for the moment it is branch j. Assuming a set of predicates Δ_j, the pattern matching variable x_j is bound to the body of the branch and specialized as a static function. Predicate IsFunS ensures type τ'_j is the result of specializing a static function from residual type τ'_{j2} to τ'_{j1}, so after (static) application, we know the final residual expression has type τ'_{j1}. The third premise states types τ and τ'_{j1} must be SR-related; this has the same purpose as rule (DLAM) in the original PTS presentation (see section 2.3) — namely to rule out undesirable specializations.

Now in order to choose branch j correctly, we would need to know which summand the control expression lies in. This cannot be assumed, since some of the information from the context could be missing! This is where the new predicates — including conditional predicates — play an important role. Indeed, specialization is not done for a particular branch j but for all of them: for each one, a (potentially different) set of predicates Δ_j is assumed, and all the predicates in it are guarded so that only the ones for the correct j must hold. The rest of them can be proved trivially by rule (IsConstr-False). Notice the use of evidence v to express the correct constructor.

A similar issue appears when determining the final residual type. As we have mentioned, it must be τ'_{j1} for a particular j, which is determined from the context when enough information is available. This is expressed by predicate **case-of** in the last premise.

Finally, the residual expression is constructed using $\mathbf{case}_v$ and $@_v$ expressions, that may be reduced at specialization time. The first one uses evidence v to choose the appropriate branch — see the first reduction rule in figure 2.4. The second one expresses the computation of static applications, applying the function e'_j to the specialized control expression e'. Evidence substitution is necessary from a technical point of view, to relate Δ_j and Δ.

Example 2.23 Static constructors generate residual constructor types.

$$data\ SD^S\ = Sta^S\ Int^S\ \mid Dyn^S\ Int^D$$

$$\vdash_{\mathrm{P}} \left(Sta^S\ 11^S, Dyn^S\ 4^D\right)^D : \left(SD^S, SD^S\right)^D \hookrightarrow (\bullet, 4) : (Sta\ \hat{1}1, Dyn\ Int)$$

$$\vdash_{\mathrm{P}} \left(Sta^S\ 11^S, Sta^S\ 4^S\right)^D : \left(SD^S, SD^S\right)^D \hookrightarrow (\bullet, \bullet) : (Sta\ \hat{1}1, Sta\ \hat{4})$$

◇

Example 2.24 Case expressions can be specialized even when we do not know which summand the control expression lies in. All branches are specialized assuming the argument matches, using conditional predicates to avoid incorrect assumptions, and a $\mathbf{case}_v$ construct to defer choosing the branch.

The expression

$$data\ SD^S\ = Sta^S\ Int^S\ \mid Dyn^S\ Int^D$$

$$\begin{array}{l}\lambda^D e.\mathbf{case}^S\ e\ \mathbf{of}\\ \quad Sta^S\ x\ \rightarrow\ \mathbf{lift}\ x\\ \quad Dyn^S\ y\ \rightarrow\ y\\ : SD^S \rightarrow^D Int^D\end{array}$$

specializes to

$$\begin{array}{l}\Lambda h_e, h_x, h_r.\lambda e'.\mathbf{case}_v\ h_e\ \mathbf{of}\\ \quad Sta\ \rightarrow\ h_x\\ \quad Dyn\ \rightarrow\ e' : \forall t_e, t_x.\ \text{IsConstrOf}\ SD\ t_e,\\ \qquad\qquad\qquad\qquad (t_e \text{ is a } Sta)\,?\,\text{IsInt}\ t_x,\\ \qquad\qquad\qquad\qquad Int := \mathbf{case}\ t_e\ \mathbf{of}\quad Sta\ \rightarrow\ Int\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Dyn\ \rightarrow\ Int \Rightarrow t_e \rightarrow Int\end{array}$$

Here, h_e is the evidence for t_e being a constructor type, so it represents the name of a constructor: either *Sta* or *Dyn*. Variable h_x is evidence for t_x — the type of *Sta*'s argument — being an integer one-point type, if t_e is actually the constructor-type *Sta*. Predicate **case-of** defers the choice of the result type.

If the function above is applied to $(Dyn^S\ 4^D)$, then e' has residual type $Dyn\ Int$, so evidence can be resolved assigning value *Dyn* to h_e and $\bullet$ to h_x, getting:

$$\begin{array}{l}(\lambda e'.\mathbf{case}_v\ Dyn\ \mathbf{of}\\ \quad Sta\ \rightarrow\ \bullet\\ \quad Dyn\ \rightarrow\ e')@4 : Int\end{array}$$

and after reduction

$$(\lambda e'.e')@4 : Int$$

where the *Sta* branch, whose body was not even a valid expression of type *Int*, has been erased.

If alternatively, the function is applied to $(Sta^S\ 11^S)$, then e' specializes to $\bullet : Sta\ \hat{1}1$. Evidence can be resolved assigning value *Sta* to h_e and 11 to h_x, getting:

$$\begin{array}{l}(\lambda e'.\mathbf{case}_v\ Sta\ \mathbf{of}\\ \quad Sta\ \rightarrow\ 11\\ \quad Dyn\ \rightarrow\ \bullet)@\bullet : Int\end{array}$$

and after reduction

$$(\lambda e'.11)@\bullet : Int$$

so now it is the *Right* branch the one with an invalid body that has been erased. ◇

Example 2.25 In an ordinary case expression, all the branches must have the same type. Indeed, a $\mathbf{case}^S$ expression is well-typed only if all the branches' (source) types are the same. However, the residual types need not match — the **case** construct is meant to disappear from the code anyway, and only one of the branches will remain!

The following expression

$$\begin{array}{l}data\ ABC^S\ = A^S\ Int^S\ \mid B^S\ Int^S\ \mid C^S\ Int^S\\ \lambda^D e.\mathbf{case}^S\ e\ \mathbf{of}\quad A^S\ x\ \rightarrow\ 4^S\\ \qquad\qquad\qquad\qquad\quad B^S\ y\ \rightarrow\ y\\ \qquad\qquad\qquad\qquad\quad C^S\ z\ \rightarrow\ 9^S\\ : ABC^S \rightarrow^D Int^S\end{array}$$

specializes to

$$
\begin{array}{l}
\Lambda h_e, h_x, h_y, h_z, h_{r1}, h_{r2}. \\
\lambda e'.\mathbf{case}_v\ h_e\ \mathbf{of}\ \ A\ \to\ \bullet \\
\qquad\qquad\qquad\qquad\quad\ B\ \to\ e' \\
\qquad\qquad\qquad\qquad\quad\ C\ \to\ \bullet \\
: \forall t_e, t_x, t_y, t_z, t_r.\ \mathrm{IsConstrOf}\ ABC\ t_e, \\
\qquad\qquad\qquad\qquad\quad (t_e \text{ is a } A)\,?\,\mathrm{IsInt}\ t_x, \\
\qquad\qquad\qquad\qquad\quad (t_e \text{ is a } B)\,?\,\mathrm{IsInt}\ t_y, \\
\qquad\qquad\qquad\qquad\quad (t_e \text{ is a } C)\,?\,\mathrm{IsInt}\ t_z, \\
\qquad\qquad\qquad\qquad\quad \mathrm{IsInt}\ t_r, \\
\qquad\qquad\qquad\qquad\quad t_r := \mathbf{case}\ t_e\ \mathbf{of}\ \ A\ \to\ \hat{4} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad B\ \to\ t_y \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad C\ \to\ \hat{9} \Rightarrow t_e \to t_r
\end{array}
$$

where each $\bullet$ expression has a different residual type, but only one of them is relevant to the type of the expression. If the function above is applied for example to $B^S\ 11^S$, we get:

$$(\lambda e'.e')@\bullet : \hat{11}$$

$\diamond$

Example 2.26 Non-matching branches could even have a specialization that is impossible to solve.

$$
\begin{array}{l}
\vdash_{\mathrm{P}}\ \lambda^D e.\mathbf{case}^S\ e\ \mathbf{of}\ \ A^S\ x\ \to\ \mathbf{let}^D\ id = \lambda^D w.w\ \mathbf{in}\ \left(id\ @^D\ 3^S, id\ @^D\ 4^S\right)^D \\
\qquad\qquad\qquad\qquad\quad\ B^S\ y\ \to\ (y,y)^D \\
\qquad\qquad\qquad\qquad\quad\ C^S\ z\ \to\ \left(3^S, 4^S\right)^D\ : \\
\quad ABC^S \to^D \left(Int^S, Int^S\right)^D\ \hookrightarrow \\
\quad \Lambda h_e, h_x, h_{id}, h_3, h_4, h_y, h_z, h_{r1}, h_{r2}, h_r. \\
\quad \lambda e'.\mathbf{case}_v\ h_e\ \mathbf{of} \\
\qquad\qquad A\ \to\ \mathbf{let}\ id' = \lambda w'.w'\ \mathbf{in}\ (id'@\bullet, id'@\bullet) \\
\qquad\qquad B\ \to\ (e', e') \\
\qquad\qquad C\ \to\ (\bullet, \bullet) \\
\qquad\qquad : \forall t_e, t_x, t_y, t_z, t_{r_1}, t_{r2}.\ \mathrm{IsConstrOf}\ ABC\ t_e, \\
\qquad\qquad\qquad\qquad\qquad (t_e \text{ is a } A)\,?\,\mathrm{IsInt}\ t_x, \\
\qquad\qquad\qquad\qquad\qquad (t_e \text{ is a } A)\,?\,\mathrm{IsInt}\ t_{id}, \\
\qquad\qquad\qquad\qquad\qquad (t_e \text{ is a } A)\,?\,(t_{id} \to t_{id}) \sim (\hat{3} \to t_3), \\
\qquad\qquad\qquad\qquad\qquad (t_e \text{ is a } A)\,?\,(t_{id} \to t_{id}) \sim (\hat{4} \to t_4), \\
\qquad\qquad\qquad\qquad\qquad (t_e \text{ is a } B)\,?\,\mathrm{IsInt}\ t_y, \\
\qquad\qquad\qquad\qquad\qquad (t_e \text{ is a } C)\,?\,\mathrm{IsInt}\ t_z, \\
\qquad\qquad\qquad\qquad\qquad \mathrm{IsInt}\ t_{r1}, \\
\qquad\qquad\qquad\qquad\qquad \mathrm{IsInt}\ t_{r2}, \\
\qquad\qquad\qquad\qquad\qquad (t_{r1}, t_{r2}) := \mathbf{case}\ t_e\ \mathbf{of}\ \ A\ \to\ (\hat{3}, \hat{3}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad B\ \to\ (t_y, t_y) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad C\ \to\ (\hat{3}, \hat{4}) \Rightarrow t_e \to t_r
\end{array}
$$

Here, the branch corresponding to constructor A applies a monovariant function to two different static integers. The only way it could be specialized is if types $\hat{3}$ and $\hat{4}$ were the

$$\text{(DCONSTR)}\quad \frac{\begin{array}{c}\Delta \vdash_{\text{SR}} D^D \;\hookrightarrow\; \tau'_e \\ \Delta \mid \Gamma \vdash_{\text{P}} e : D^D\,(K_j) \;\hookrightarrow\; e' : \tau'_j \qquad \Delta \Vdash v_j : \text{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j\end{array}}{\Delta \mid \Gamma \vdash_{\text{P}} {K_j}^{D}\ e : D^D \;\hookrightarrow\; {K_j}^{v_j}\ e' : \tau'_e}$$

$$\text{(DCASE)}\quad \frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\text{P}} e : D^D \;\hookrightarrow\; e' : \tau'_e \\ \Delta \Vdash v^d : \text{IsSum}\ \tau'_e \\ \Delta \vdash_{\text{SR}} \tau \;\hookrightarrow\; \tau' \\ \left(\begin{array}{l}\bar{h}_j : \Delta_j \mid \Gamma \vdash_{\text{P}} \ \lambda^D x_j.e_j : D^D\,(K_j) \to^D \tau \;\hookrightarrow \\ \qquad\qquad\qquad\qquad \lambda x'_j.e'_j : \tau'_j \to \tau' \\ \Delta \Vdash \bar{v}_j : K_j \in \tau'_e\,?\,\Delta_j \\ \Delta \Vdash w_j : K_j \in \tau'_e\,?\,\text{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j\end{array}\right)_{j\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\text{P}} \ \textbf{case}^D\, e\ \textbf{of}\ ({K_j}^{D}\ x_j \ \to\ e_j)_{j\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \textbf{protocase}_v\ e'\ \textbf{with}\ v^d\ \textbf{of} \\ \qquad\qquad\quad ({K_j}^{w_j}\ x'_j \ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : \tau'\end{array}}$$

Figure 2.8: Specialization rules involving dynamic sum types

same! This is ensured by two unification predicates, but being guarded, they only need to hold if A's branch is taken. If the function is applied, for example, to $B^S\ 11^S$, we get:

$$(\lambda e'.(e', e'))@\bullet : (\hat{1}\hat{1}, \hat{1}\hat{1})$$

If alternatively it is applied to $A^S\ 4^S$, the residual expression cannot be solved. ◇

Specializing dynamic sum types Specialization of dynamic sum types is meant to preserve the constructors and case structures in the residual code. The data declarations are preserved as well, but they can generate more than one version, and each version can be modified to leave out unreachable constructors. The case branches corresponding to these constructors are erased as well.

Rule (DCONSTR) specifies how a dynamic tagged expression is specialized. Firstly, the residual type has to be SR-related to the source type; this rules out incorrect residual types, such as those whose constructors cannot all be obtained from D's source definition. Also, the expression e must specialize to an expression e' of residual type τ'_j such that the residual sum type includes a summand $K_j\ \tau'_j$ in its definition. Predicate HasC expresses this last condition; notice the use of v_j to get the correct copy of the constructor.

Rule (DCASE) specifies the specialization of case expressions. The first two premises involve the specialization of the control expression e: it must specialize a residual type τ'_e that can be proved to be a sum type. The following premise states that the residual type τ' must be SR-related to the source type. The next three conditions must hold for every branch in the case expression. Assuming a set of predicates Δ_j and that the constructor's argument has residual type τ'_j, the right hand side of each branch is specialized to an expression of type τ' — specialization of dynamic functions is used to express this. Now since not necessarily all the constructors in the branches will appear in τ'_e's declaration, the predicates assumed

for proving this must only hold in the cases in which they do, which explains the use of the conditional predicate to prove Δ_j. Similarly, only if K_j appears in τ'_e's declaration must τ'_j be its argument. The evidence for predicate IsSum is used in the **protocase**$_v$ expression as the actual set of constructors appearing in τ'_e's declaration.

Example 2.27 Evidence for predicate HasC is used to generate the correct constructor in the residual expression.

$$data\ D^D\ = Only^D\ Int^S$$

$$\begin{array}{l} \vdash_{\mathrm{P}}\ \ Only^D\ 11^S\ :\ D^D \\ \qquad \hookrightarrow\ \Lambda h_1, h_2.Only^{h_2}\ \bullet : \forall t.\ \mathrm{IsSum}\ t, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{HasC}\ t\ \ Only\ \ \hat{1\mathrm{1}} \Rightarrow t \end{array}$$

Here, h_1 is evidence for t being a sum type and h_2 is the number of D's declaration where *Only* appears with argument $\hat{1\mathrm{1}}$.

Similarly,

$$\begin{array}{l} \vdash_{\mathrm{P}}\ \ \left(Only^D\ 11^S\,, Only^D\ 4^S\right)^D\ :\ \left(D^D\,, D^D\right)^D \\ \qquad \hookrightarrow\ \Lambda h_1, h_2, h_3, h_4.(Only^{h_2}\ \bullet, Only^{h_4}\ \bullet) : \forall t, t'.\ \mathrm{IsSum}\ t, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{HasC}\ t\ \ Only\ \ \hat{1\mathrm{1}}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{IsSum}\ t', \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{HasC}\ t'\ \ Only\ \ \hat{4} \Rightarrow (t, t') \end{array}$$

If constraint solving detects t and t' are not related, two different types can be generated, so h_2 and h_4 will be assigned values 1 and 2 respectively, yielding the specialization shown in example 2.20. If, on the contrary, it detects t and t' must be the same, then the specialization cannot be solved, since $\hat{1\mathrm{1}}$ is not the same as $\hat{4}$. ◇

Example 2.28 Not all constructors need appear in the source code. For those that do not, only conditional information appears in the residual type.

$$data\ DS^D\ = Dyn^D\ Int^D\ \mid Sta^D\ Int^S$$

$$\begin{array}{l} \vdash_{\mathrm{P}}\ \ Dyn^D\ 11^D\ :\ DS^D \\ \qquad \hookrightarrow\ \Lambda h_1, h_2, h_3, h_4.Dyn^{h_2}\ 11 : \forall t, t'.\ h_1 : \mathrm{IsSum}\ t, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad h_2 : \mathrm{HasC}\ t\ \ Dyn\ \ Int, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad h_3 : Sta \in t\,?\,\mathrm{HasC}\ t\ \ Sta\ \ t', \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad h_4 : Sta \in t\,?\,\mathrm{IsInt}\ t' \Rightarrow t \end{array}$$

If constructor *Sta* is ever applied to an argument, the type must unify with t' by entailment rules (HasC-Guard) and (Unify-HasC). If it is never used, constraint solving detects it and removes it from the residual data declaration.

$$data\ DS^1\ = Dyn^1\ Int$$

$$Dyn^1\ 11 : DS^1$$

◇

Example 2.29 The specialization rules for dynamic sum types always expect a constructor applied to an argument. Constructors with multiple arguments can be simulated by tuples, whereas constructors with no arguments are η-expanded before specialization and η-reduced after all post-processing phases.

$$\begin{array}{l} \mathit{data}\ D^D\ =\ \mathit{Only}^D\ \mathit{Int}^S \\ \\ \vdash_{\mathrm{P}}\ \ \mathit{Only}^D : \mathit{Int}^S \to^D D^D \\ \qquad \hookrightarrow\ \Lambda h_1, h_2, h_3.\mathit{Only}^{h_3} : \forall t, t'.\ h_1 : \mathrm{IsInt}\ t', \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_2 : \mathrm{IsSum}\ t, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_3 : \mathrm{HasC}\ t\ \ \mathit{Only}\ \ t' \Rightarrow t' \to t \end{array}$$

◇

Example 2.30 If two expressions tagged with the same constructor belong to the same residual type, information flows from one to the other by means of predicate (Unify-HasC).

$$\begin{array}{l} \mathit{data}\ D^D\ =\ \mathit{Only}^D\ \mathit{Int}^S \\ \\ \vdash_{\mathrm{P}}\ \ \mathbf{let}^D\ \ id = \lambda^D z.z \\ \qquad \mathbf{in}\ \lambda^D x.\big(id\ @^D\ (\mathit{Only}^D\ x), id\ @^D\ (\mathit{Only}^D\ 4^S), \mathbf{lift}\ x\big)^{\ D} : \mathit{Int}^S \to^D \big(D^D, D^D, \mathit{Int}^D\big)^{\ D} \\ \quad \hookrightarrow \\ \quad \Lambda h_1, h_2, h_3, h_4. \\ \quad \mathbf{let}\ id' = \lambda z'.z' \\ \quad\ \ \mathbf{in}\ \lambda x'.(id'@(\mathit{Only}^{h_3}\ \bullet), id'@(\mathit{Only}^{h_4}\ \bullet), h_1) \\ \quad\ \ \ : \forall t, t'.\ h_1 : \mathrm{IsInt}\ t', \\ \qquad\qquad\ h_2 : \mathrm{IsSum}\ t, \\ \qquad\qquad\ h_3 : \mathrm{HasC}\ t\ \ \mathit{Only}\ \ t', \\ \qquad\qquad\ h_4 : \mathrm{HasC}\ t\ \ \mathit{Only}\ \ \hat{4} \Rightarrow \hat{4} \to (t, t, \mathit{Int}) \end{array}$$

Here, the identity function id is monovariant, so $\mathit{Only}^D\ x$ and $\mathit{Only}^D\ 4^S$ must have the same residual type t. Constraint solving detects this, and by predicate (Unify-HasC), t' and $\hat{4}$ must unify so h_3 and h_4 are assigned the same value.

$$\begin{array}{l} \mathit{data}\ D^1\ =\ \mathit{Only}^1\ \hat{4} \\ \\ \mathbf{let}\ id' = \lambda z'.z' \\ \quad \mathbf{in}\ \lambda x'.(id'@(\mathit{Only}^1\ \bullet), id'@(\mathit{Only}^1\ \bullet), 4) : \hat{4} \to (D^1, D^1, \mathit{Int}) \end{array}$$

◇

Example 2.31 Specialization and entailment rules involving dynamic sum types were designed so that constraint solving can detect when a **case** branch can never be taken, erasing it as dead code.

$$data\ SD^D\ = Sta^D\ Int^S\ \mid Dyn^D\ Int^D$$

$$\begin{array}{ll}
\vdash_{\mathrm{P}} & \mathbf{case}^D\,(Sta^D\ 11^S)\ \mathbf{of} \\
& \quad Sta^D\ x\ \rightarrow\ \mathbf{lift}\ x \\
& \quad Dyn^D\ x\ \rightarrow\ x \quad : Int^D \\
& \hookrightarrow\ \Lambda h_1, h_2, h_3. \\
& \quad \mathbf{protocase}_v\ (Sta^{h_2}\ \bullet)\ \mathbf{with}\ h_1\ \mathbf{of} \\
& \quad Sta^{h_2}\ x'\ \rightarrow\ 11 \\
& \quad Dyn^{h_3}\ x'\ \rightarrow\ x' \quad : \forall t.\ h_1 : \mathrm{IsSum}\ t, \\
& \qquad\qquad\qquad\qquad\quad h_2 : \mathrm{HasC}\ t\ \ Sta\ \ \hat{1}1, \\
& \qquad\qquad\qquad\qquad\quad h_3 : Dyn \in t\,?\,\mathrm{HasC}\ t\ \ Dyn\ \ Int \Rightarrow Int
\end{array}$$

Constraint solving assigns the set $\{Sta\}$ to h_1 and produces

$$data\ SD^1\ = Sta^1\ \hat{1}1$$

$$\mathbf{case}\ (Sta^1\ \bullet)\ \mathbf{of}\ Sta^1\ x\ \rightarrow\ 11 : Int$$

◇

Example 2.32 When there is no information regarding the use of constructors in the residual code, all predicates that affect them are guarded, and the $\mathbf{protocase}_v$ expression cannot be reduced.

$$data\ SD^D\ = Sta^D\ Int^S\ \mid Dyn^D\ Int^D$$

$$\begin{array}{ll}
\vdash_{\mathrm{P}} & \lambda^D e.\mathbf{case}^D\ e\ \mathbf{of} \\
& \qquad Sta^D\ n\ \rightarrow\ \mathbf{lift}\ n \\
& : SD^D \rightarrow^D Int^D\ \ \hookrightarrow \\
& \Lambda h_1, h_2, h_3, h_4.\lambda e'.\mathbf{protocase}_v\ e'\ \mathbf{with}\ h_1\ \mathbf{of} \\
& \qquad Sta^{h_2}\ n'\ \rightarrow\ h_3 \\
& : \forall t, t'.\ h_1 : \mathrm{IsSum}\ t, \\
& \qquad\quad h_2 : Sta \in t\,?\,\mathrm{HasC}\ t\ \ Sta\ \ t', \\
& \qquad\quad h_3 : Sta \in t\,?\,\mathrm{IsInt}\ t', \\
& \qquad\quad h_4 : Dyn \in t\,?\,\mathrm{HasC}\ t\ \ Dyn\ \ Int \Rightarrow t \rightarrow Int
\end{array}$$

◇

Chapter 3

Static branch erasure in dynamic sum types

Let us recall the meaning of static and dynamic expressions in type specialization. In general, a static source expression is intended to disappear from the code during specialization, passing its information to the residual type. A dynamic expression, on the contrary, must in general remain in the program, adding no new information to the type.

One of the distinctive features of principal type specialization is that it makes use of static information in an extensive and flexible way, even when it is only partially static information.

In this section, we propose a slight modification to one of the rules introduced by Russo's extension to the system (presented in section 2.5), namely rule (DCASE), to take the flexibility and use of static information a step further. Our formulation resolves a family of conflicts that originally prevented specialization of some case expressions, thus allowing more valid specializations without losing consistency.

3.1 Static information in expressions involving dynamic sums

Expressions involving dynamic sum types are of two different kinds — tagged expressions, beginning with a constructor name, and case expressions. Being dynamic, when specializing the first kind we expect the constructor name to remain in the residual program, and for the second kind, the case structure must be kept. This way, no information regarding the source expressions passes directly to the residual type.

However, in Russo's formulation for dynamic sum types, some information *is* obtained statically and kept in the residual type, namely information regarding the sum's definition — the data declaration! A residual sum type τ' is constrained by predicates that specify which summands its declaration must necessarily include. During constraint solving, only these are kept in the code. In other words, the constructors that need not be in τ' are *statically* removed from the residual program. As a result, the residual data declaration can have fewer constructors than the corresponding source one, and the residual case expressions can have fewer branches (because branches matching the constructors that no longer exist must also be removed). Branch erasure during specialization of dynamic case expressions is one of its most interesting features.

Russo's extension introduces new guarded predicates for some of the constraints. These make them relative to the inclusion of a certain constructor in the type's declaration —

informally, if a predicate is guarded, it must hold only when a certain constructor belongs to a certain sum type, otherwise it can be proved trivially.

Guarded predicates are essential for specifying branch erasure when specializing the right hand side of the branches in a case expression. Recall the following premises in rule (DCASE).

$$\left(\begin{array}{l} \bar{h}_j : \Delta_j \mid \Gamma \vdash_{\mathrm{P}} \quad \lambda^D x_j.e_j : D^D\,(K_j) \to^D \tau \;\hookrightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad \lambda x'_j.e'_j : \tau'_j \to \tau' \\ \Delta \Vdash \bar{v}_j : K_j \in \tau'_e \,?\, \Delta_j \\ \Delta \Vdash w_j : K_j \in \tau'_e \,?\, \mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \end{array}\right)_{j \in B}$$

These must hold for each branch of the case expression to specialize. The set of predicates Δ_j is assumed for specializing each branch j, but only those corresponding to a constructor that belongs to the sum type must actually hold. During constraint solving, evidence for Δ_j need not be constructed if the branch can be removed. This allows solving expressions that would otherwise be ambiguous.

Example 3.1 Given the following specialization

$$\begin{array}{l} \mathit{data}\ \mathit{Either}^D\ = \mathit{Left}^D\ \mathit{Int}^S\ \mid \mathit{Right}^D\ \mathit{Int}^S \\ \\ \vdash_{\mathrm{P}}\ \ \mathbf{case}^D\ \mathit{Left}^D\ 2^S\ \mathbf{of} \\ \qquad \mathit{Left}^D\ x\ \to\ \mathbf{lift}\ x \\ \qquad \mathit{Right}^D\ y\ \to\ \mathbf{lift}\ (y +^S 1^S) : \mathit{Int}^D \\ \quad \hookrightarrow \\ \quad \Lambda h_1, h_2, h_3, h_4, h_5. \\ \quad \mathbf{protocase}_v\ \mathit{Left}^{h_2}\ \bullet\ \mathbf{with}\ h_1\ \mathbf{of} \\ \qquad \mathit{Left}^{h_2}\ x'\ \to\ 2 \\ \qquad \mathit{Right}^{h_5}\ y\ \to\ h_4 \\ \quad : \forall t_1, t_2, t_3.\ h_1 : \mathrm{IsSum}\ t_1, \\ \qquad\qquad\qquad h_2 : \mathrm{HasC}\ t_1\ \ \mathit{Left}\ \ \hat{2}, \\ \qquad\qquad\qquad h_3 : \mathit{Right} \in t_1 \,?\, \mathrm{IsInt}\ t_2, \\ \qquad\qquad\qquad h_4 : \mathit{Right} \in t_1 \,?\, t_3 := t_2 + \hat{1}, \\ \qquad\qquad\qquad h_5 : \mathit{Right} \in t_1 \,?\, \mathrm{HasC}\ t_1\ \ \mathit{Right}\ \ t_3 \Rightarrow \mathit{Int} \end{array}$$

evidence h_2 can be solved to number 1, and h_1 to the set $\{\mathit{Left}^1\}$. The other three predicates, since constructor *Right* needs not belong to the residual sum type, can be proved trivially with evidence $\bullet$, yielding

$$\begin{array}{l} \mathit{data}\ \mathit{Either}^1\ = \mathit{Left}^1\ \hat{2} \\ \\ \mathbf{protocase}_v\ \mathit{Left}^1\ \bullet\ \mathbf{with}\ \{\mathit{Left}^1\}\ \mathbf{of} \\ \quad \mathit{Left}^1\ x\ \to\ 2 \\ \quad \mathit{Right}^\bullet\ y\ \to\ \bullet : \mathit{Int} \end{array}$$

which, by means of the reduction rules for $\mathbf{protocase}_v$ expressions, is finally reduced to

$$\begin{array}{l} \mathbf{case}\ \mathit{Left}^1\ \bullet\ \mathbf{of} \\ \quad \mathit{Left}^1\ x\ \to\ 2 : \mathit{Int} \end{array}$$

Predicates IsInt t_2 and $t_3 := t_2 + \hat{1}$ belong to the set Δ_2 in the premise of our rule, corresponding to constructor *Right*. Had they not been guarded, there would be no feasible evidence to replace h_3 and h_4, so the constraint solving step could not have been performed[1]. The **protocase**$_v$ construct would not be reduced, and we would only get the first specialization, which is not very useful! ◇

Examples like this one illustrate the main motivation for branch erasure, and more generally, the use of guarded predicates — they are not only convenient, but essential for achieving reasonable specializations in a number of cases. There is a big family of valid source terms that could not be solved into useful residual code if dead branches were not erased.

Another interesting consequence of using guarded predicates is that certain expressions can be specialized successfully or not, just depending on what constructors appear in them.

Example 3.2 Consider the specialization

$$\begin{array}{l}
\mathit{data}\ \mathit{Bool}^D\ = \mathit{True}^D\ ()^D\ \mid \mathit{False}^D\ ()^D \\
\\
\vdash_{\mathrm{P}}\ \mathbf{case}^D\ \mathit{False}^D\ ()^D\ \mathbf{of} \\
\qquad \mathit{True}^D\ x\ \rightarrow\ \mathbf{let}^D\ \mathit{id} = \lambda^D z.z\ \mathbf{in}\ \left(\mathit{id}\ @^D\ 4^S\,, \mathit{id}\ @^D\ 3^S\right)^D \\
\qquad \mathit{False}^D\ x\ \rightarrow\ \left(4^S\,, 3^S\right)^D \\
: \left(\mathit{Int}^S\,, \mathit{Int}^S\right)^D\ \hookrightarrow \\
\Lambda h_1, h_2, h_3, h_4, h_5, h_6.\mathbf{protocase}_v\ \mathit{False}^{h_6}\ ()\ \mathbf{with}\ h_1\ \mathbf{of} \\
\qquad \mathit{True}^{h_5}\ x'\ \rightarrow\ \mathbf{let}\ \mathit{id}' = \lambda z'.z'\ \mathbf{in}\ (\mathit{id}'@\bullet, \mathit{id}'@\bullet) \\
\qquad \mathit{False}^{h_6}\ x'\ \rightarrow\ (\bullet, \bullet) \\
: \forall t, t_{id}.\ h_1 : \mathrm{IsSum}\ t, \\
\qquad\qquad h_2 : \mathit{True} \in t\,?\,\mathrm{IsInt}\ t_{id}, \\
\qquad\qquad h_3 : \mathit{True} \in t\,?\,(t_{id} \rightarrow t_{id}) \sim (\hat{3} \rightarrow \hat{3}), \\
\qquad\qquad h_4 : \mathit{True} \in t\,?\,(t_{id} \rightarrow t_{id}) \sim (\hat{4} \rightarrow \hat{4}), \\
\qquad\qquad h_5 : \mathit{True} \in t\,?\,\mathrm{HasC}\ t\ \ \mathit{True}\ ()^D, \\
\qquad\qquad h_6 : \mathrm{HasC}\ t\ \ \mathit{False}\ ()^D \\
\qquad\qquad \Rightarrow (\hat{4}, \hat{3})
\end{array}$$

Here the identity function $\lambda^D z.z$ is monovariant, so it could not be applied both to 4^S and 3^S. The unification predicates could never be proved together, and would therefore yield an error in constraint solving. Indeed, the right hand side of the first branch could not be specialized in isolation! However, as the predicate is guarded with a false condition, constraint solving is possible, and after evidence elimination and reduction we obtain

$$\begin{array}{l}
\mathit{data}\ \mathit{Bool}^1\ = \mathit{False}^1\ () \\
\\
\mathbf{case}^D\ \mathit{False}^1\ ()\ \mathbf{of} \\
\qquad \mathit{False}^1\ x\ \rightarrow\ (\bullet, \bullet)\ \ : (\hat{4}, \hat{3})
\end{array}$$

which is a desirable result, since the problematic branch could never be taken in the residual code. ◇

[1] We say expressions like these are *ambiguous*, meaning they contain predicates that cannot be solved, not because it is impossible to build evidence for them, but because we do not know which evidence is required. Ambiguous type schemes have been defined by Mark Jones as a part of his theory of qualified types [Jones, 1994; Jones, 1993]

Example 3.3

$$
\begin{array}{l}
\mathit{data}\ \mathit{Either}^D\ = \mathit{Left}^D\ \mathit{Int}^S\ \mid \mathit{Right}^D\ \mathit{Int}^S \\
\\
\mathbf{let}^D\ f = \lambda^D x.\lambda^D y.x\ \mathbf{in} \\
\quad \mathbf{let}^D\ g = \lambda^D e.\ \mathbf{case}^D\ e\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{Left}^D\ x\ \rightarrow\ \mathbf{lift}\ x \\
\qquad\qquad\qquad \mathit{Right}^D\ y\ \rightarrow\ \mathbf{let}^D\ \mathit{id} = \lambda^D z.z\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}^D\ \mathit{dummy} = \left(\mathit{id}\ @^D\ f, \mathit{id}\ @^D\ (\lambda^D x.\lambda^D y.y)\right)^D\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \mathbf{lift}\ y \\
\ \mathbf{in}\ \left(f\ @^D\ 4^S\ @^D\ 3^S, g\ @^D\ (\mathit{Left}^D\ 2^S)\right)^D \\
: \left(\mathit{Int}^S, \mathit{Int}^D\right)^D
\end{array}
$$

This example is more complex. In the second branch, function f, which returns the first of two (static) arguments, must have the same type as a function that returns the second argument, by virtue of the monovariant identity function. As a result, both function arguments would have to belong to the same residual type, and the expression $f\ @^D\ 4^S\ @^D\ 3^S$ could not be specialized. However, since constructor *Right* does not appear in the type, there is no error.

$$
\begin{array}{l}
\mathit{data}\ \mathit{Either}^1\ = \mathit{Left}^1\ \hat{2} \\
\\
\mathbf{let}\ f' = \lambda x'.\lambda y'.x'\ \mathbf{in} \\
\quad \mathbf{let}\ g' = \lambda e'.\ \mathbf{case}\ e'\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{Left}^1\ x\ \rightarrow\ 2 \\
\ \mathbf{in}\ \left(f'@\bullet@\bullet, g'@(\mathit{Left}^1\ \bullet)\right)^D \\
: (\hat{4}, \mathit{Int})
\end{array}
$$

Should we change the source expression to

$$
\begin{array}{l}
\mathbf{let}^D\ f = \ldots \\
\ \mathbf{in}\ \left(f\ @^D\ 4^S\ @^D\ 3^S, g\ @^D\ (\mathit{Right}^D\ 2^S)\right)^D
\end{array}
$$

we would not be able to solve it, since types $\hat{4}$ and $\hat{3}$ are not the same.

Here, the branches can be correctly specialized by themselves, and it is their combination with the rest of the expression what causes the problems. The principle is the same — knowing that the problematic part could never be evaluated in the residual code, it is good to erase it and specialize successfully. ◇

It is in examples like the ones above when the partially static quality of a dynamic sum definition becomes most noticeable. A specialization can switch from valid to impossible with no modification whatsoever of the problematic part, only by changing which constructors must appear in the type! We have seen this behavior in static case expressions — see example 2.26.

The following examples are similar to the examples above, only making the sum types and case expressions static instead of dynamic. As a result, the constructors and the case structure are removed from the residual code. The appropriate branch is taken and the rest of them are naturally discarded, even if they cannot be specialized correctly in isolation, or if they cause problems when combined with the rest of the expression.

Example 3.4

$$
\begin{array}{l}
\mathit{data}\ \mathit{Bool}^S\ =\ \mathit{True}^S\ ()^D\ \mid\ \mathit{False}^S\ ()^D \\
\\
\vdash_{\mathrm{P}}\ \mathbf{case}^S\ \mathit{False}^S\ ()^D\ \mathbf{of} \\
\qquad\quad \mathit{True}^S\ x\ \rightarrow\ \mathbf{let}^D\ \mathit{id} = \lambda^D z.z\ \mathbf{in}\ \left(\mathit{id}\ @^D\ 4^S\,, \mathit{id}\ @^D\ 3^S\right)^D \\
\qquad\quad \mathit{False}^S\ x\ \rightarrow\ \left(4^S\,, 3^S\right)^D \\
\qquad : \left(\mathit{Int}^S\,, \mathit{Int}^S\right)^D \\
\qquad \hookrightarrow \\
\qquad \mathbf{case}_v\ \mathit{False}\ \mathbf{of} \\
\qquad\quad \mathit{True}\ \rightarrow\ \mathbf{let}\ \mathit{id}' = \lambda^D z'.z'\ \mathbf{in}\ (\mathit{id}'@\bullet, \mathit{id}'@\bullet) \\
\qquad\quad \mathit{False}\ \rightarrow\ (\bullet, \bullet) \\
\qquad : (\hat{4}, \hat{3})
\end{array}
$$

where the residual expression reduces to

$$(\bullet, \bullet) : (\hat{4}, \hat{3})$$

◇

Example 3.5

$$
\begin{array}{l}
\mathit{data}\ \mathit{Either}^S\ =\mathit{Left}^S\ \mathit{Int}^S\ \mid\ \mathit{Right}^S\ \mathit{Int}^S \\
\\
\mathbf{let}^D\ f = \lambda^D x.\lambda^D y.x\ \mathbf{in} \\
\quad \mathbf{let}^D\ g = \lambda^D e.\ \mathbf{case}^S\ e\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{Left}^S\ x\ \rightarrow\ \mathbf{lift}\ x \\
\qquad\qquad\qquad \mathit{Right}^S\ y\ \rightarrow\ \mathbf{let}^D\ \mathit{id} = \lambda^D z.z\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}^D\ \mathit{dummy} = \left(\mathit{id}\ @^D\ f, \mathit{id}\ @^D\ (\lambda^D x.\lambda^D y.y)\right)^D\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{lift}\ y \\
\quad \mathbf{in}\ \left(f\ @^D\ 4^S\ @^D\ 3^S\,, g\ @^D\ (\mathit{Left}^S\ 2^S)\right)^D \\
: \left(\mathit{Int}^S\,, \mathit{Int}^D\right)^D
\end{array}
$$

Here, function g is applied to an argument that specializes to residual type $\mathit{Left}\ \hat{2}$, so e must have the same type. The $\mathbf{case}_v$ construct generated by the static **case** can then be reduced choosing the *Left* branch, getting

$$
\begin{array}{l}
\mathbf{let}\ f' = \lambda x'.\lambda y'.x'\ \mathbf{in} \\
\quad \mathbf{let}\ g' = \lambda e'.2 \\
\ \mathbf{in}\ (f'@\bullet\ @\bullet, g'@\bullet)^D \\
: (\hat{4}, \mathit{Int})
\end{array}
$$

◇

These examples illustrate the similarities between static and dynamic sum type specialization. Indeed, in both levels there is static information allowing to discard the parts that would lead to a specialization failure. There is certainly more static information in a static case expression than in a dynamic one — with the former we can tell during specialization precisely which branch *will* be taken, whereas with the latter, we can only know certain branches *will not* be taken!

Example 3.6 It is not always possible to discard the presence of a dynamic constructor in the residual type. In that case, no branches can be erased.

$$
\begin{array}{l}
\mathit{data}\ \mathit{Bool}^D\ =\ \mathit{True}^D\ ()^D\ \mid \mathit{False}^D\ ()^D \\
\\
\mathbf{let}^D\ \mathit{id} = \lambda^D z.z\ \mathbf{in} \\
\mathbf{let}^D\ \mathit{true} = \mathit{id}\ @^D\ (\mathit{True}^D\ ()^D)\ \mathbf{in} \\
\mathbf{let}^D\ \mathit{false} = \mathit{id}\ @^D\ (\mathit{False}^D\ ()^D)\ \mathbf{in} \\
\ \mathbf{case}^D\ \left(\mathbf{fst}^D\ (\mathit{false},\ \mathit{true})^D\right)\ \mathbf{of} \\
\quad \mathit{True}^D\ x\ \rightarrow\ \mathbf{let}^D\ \mathit{id} = \lambda^D z.z\ \mathbf{in}\ \left(\mathit{id}\ @^D\ 4^S, \mathit{id}\ @^D\ 3^S\right)^D \\
\quad \mathit{False}^D\ x\ \rightarrow\ \left(4^S, 3^S\right)^D \\
:\left(\mathit{Int}^S, \mathit{Int}^S\right)^D
\end{array}
$$

Compare this expression to example 3.2. Here, the **fst** projection is meant to remain in the residual code, so both constructors must belong to the residual type. No branches can be erased; in particular, the first one needs to be specialized, which cannot be done successfully since $\hat{4}$ and $\hat{3}$ are different residual types. ◇

3.2 Enhancing branch erasure

Russo's formulation uses static information as we have shown, mainly to allow solving otherwise ambiguous expressions, which occur rather frequently — at least every time a constructor with a static argument is not used in the source program. The effect of guarded predicates in rule (DCASE) is to somehow ignore the specializations of branches that are removed from the code. However, they are not entirely ignored! The first premise involving specialization of the branches —see premises transcribed above— states that they should all specialize to the same residual type τ', *including those that will be removed from the residual code.* This is independent of making the predicate contexts Δ_j conditional.

Example 3.7

$$
\begin{array}{l}
\mathit{data}\ \mathit{Bool}^D\ =\ \mathit{True}^D\ ()^D\ \mid \mathit{False}^D\ ()^D \\
\\
\mathbf{case}^D\ \mathit{False}^D\ ()^D\ \mathbf{of} \\
\quad \mathit{True}^D\ x\ \rightarrow\ \left(11^S, 3^S\right)^D \\
\quad \mathit{False}^D\ x\ \rightarrow\ \left(4^S, 3^S\right)^D : \left(\mathit{Int}^S, \mathit{Int}^S\right)^D
\end{array}
$$

Applying rule (DCASE) to this expression, the set of predicates Δ_j is empty in both branches, because no assumption is needed for specializing tuples of static numbers. However, specialization is not possible, because the body of the first branch has residual type $(\hat{11}, \hat{3})$ whereas the second branch has residual type $(\hat{4}, \hat{3})$. Making Δ_j conditional does not help: the residual types are not the same, so the rule cannot be applied. ◇

Compare this example to example 3.2, where the expression has a similar structure. In that example, specialization was possible despite the problems in the first branch. Here, specialization cannot be achieved because the first and second branches have different residual types, even though the first branch can never be taken. Surely this is an undesirable limitation — if certain branches can be safely erased, their residual types need not be taken into account.

In our approach, we relax some of the rules so that only the branches that will remain in the code must have the same residual type. This has the effect of ignoring unnecessary

$$\text{(DCASE-2)}\quad \frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} e : D^{D} \;\hookrightarrow\; e' : \tau'_e \\ \Delta \Vdash v^{d} : \mathrm{IsSum}\ \tau'_e \\ \Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \tau' \\ \left(\begin{array}{l} \bar{h}_j : \Delta_j \mid \Gamma \vdash_{\mathrm{P}} \lambda^{D} x_j.e_j : D^{D}\,(K_j) \to^{D} \tau \;\hookrightarrow\; \lambda x'_j.e'_j : \tau'_j \to \tau''_j \\ \Delta \Vdash \bar{v}_j : K_j \in \tau'_e \,?\, \Delta_j \\ \Delta \Vdash w_j : K_j \in \tau'_e \,?\, \mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \\ \Delta \Vdash K_j \in \tau'_e \,?\, \tau''_j \sim \tau' \end{array}\right)_{j \in B} \end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^{D}\, e\ \mathbf{of}\ (K_j^{\,D}\ x_j\ \to\ e_j)_{j \in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{protocase}_{v}\ e'\ \mathbf{with}\ v^{d}\ \mathbf{of} \\ \qquad\qquad\quad (K_j^{\,w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j \in B} : \tau' \end{array}}$$

$$\text{(RT-DCASE-2)}\quad \frac{\begin{array}{l}\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_e \\ \Delta \Vdash v^{d} : \mathrm{IsSum}\ \tau'_e \\ \left(\begin{array}{l} \bar{h}_j : \Delta_j \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \lambda x'_j.e'_j : \tau'_j \to \tau''_j \\ \Delta \Vdash \bar{v}_j : K_j \in \tau'_e \,?\, \Delta_j \\ \Delta \Vdash w_j : K_j \in \tau'_e \,?\, \mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \\ \Delta \Vdash K_j \in \tau'_e \,?\, \tau''_j \sim \tau' \end{array}\right)_{j \in B} \end{array}}{\begin{array}{l}\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}}\ \mathbf{protocase}_{v}\ e'\ \mathbf{with}\ v^{d}\ \mathbf{of} \\ \qquad\qquad (K_j^{\,w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j \in B} : \tau' \end{array}}$$

Figure 3.1: New version of rules for branch elimination in dynamic case expressions

branches completely, as opposed to just ignoring the constraints used for their specialization. We argue it is an improvement over the original formulation, in the sense that it yields more valid specializations without losing consistency.

Our proposal involves changing two of the rules defined by Russo [2004] (see section 2.5) namely rules (DCASE) and (RT-DCASE). The new versions are presented in figure 3.1.

Rule (DCASE-2) differs from rule (DCASE) in how the specialization of branches is specified. Instead of having all the same residual type τ', now the body of each branch has a potentially different residual type τ''_j. The last premise states that they must all unify with the final residual type τ', by means of a predicate that is made conditional, as the rest of them. As a result, only the branches appearing in the specialized case expression must have the same residual type; otherwise the unification predicate can be proved trivially.

Rule (RT-DCASE-2) is a natural adaptation of rule (RT-DCASE) to fit the new corresponding specialization rule.

Example 3.8 Having modified the rules, the following expression

$$\begin{array}{l} \mathit{data}\ \mathit{Bool}^{D} = \mathit{True}^{D}\ ()^{\,D} \mid \mathit{False}^{D}\ ()^{\,D} \\ \\ \mathbf{case}^{D}\ \mathit{False}^{D}\ ()^{\,D}\ \mathbf{of} \\ \quad \mathit{True}^{D}\ x\ \to\ \left(11^{S}, 3^{S}\right)^{D} \\ \quad \mathit{False}^{D}\ x\ \to\ \left(4^{S}, 3^{S}\right)^{D} : \left(\mathit{Int}^{S}, \mathit{Int}^{S}\right)^{D} \end{array}$$

can be specialized to

$$
\begin{array}{l}
\Lambda h_1, h_2, h_3, h_4. \\
\textbf{protocase}_v\ \mathit{False}^{h_3}\ ()\ \textbf{with}\ h_1\ \textbf{of} \\
\quad \mathit{True}^{h_2}\ x\ \rightarrow\ (\bullet, \bullet) \\
\quad \mathit{False}^{h_3}\ x\ \rightarrow\ (\bullet, \bullet) : \forall t, t_1, t_2.\ h_1 : \mathrm{IsSum}\ t, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_2 : \mathit{True} \in t\,?\,\mathrm{HasC}\ t\ \ \mathit{True}\ (), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_3 : \mathrm{HasC}\ t\ \ \mathit{False}\ (), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_6 : \mathit{True} \in t\,?\,(\hat{1\!1}, \hat{3}) \sim (\hat{4}, \hat{3}) \Rightarrow (\hat{4}, \hat{3})
\end{array}
$$

Constraint solving can detect constructor *True* needs not appear in the code, so h_3 can be assigned value 1, h_1 is $\{\mathit{False}^1\}$, and h_2 and h_4 are assigned evidence $\bullet$. Finally we get

$$
\begin{array}{l}
\mathit{data}\ \mathit{Bool}^1\ = \mathit{False}^1\ () \\
\\
\textbf{protocase}_v\ \mathit{False}^1\ ()\ \textbf{with}\ \{\mathit{False}^1\}\ \textbf{of} \\
\quad \mathit{True}^{\bullet}\ x\ \rightarrow\ (\bullet, \bullet) \\
\quad \mathit{False}^1\ x\ \rightarrow\ (\bullet, \bullet) : (\hat{4}, \hat{3})
\end{array}
$$

and after reduction,

$$
\begin{array}{l}
\textbf{case}\ \mathit{False}^1\ ()\ \textbf{of} \\
\quad \mathit{False}^1\ x\ \rightarrow\ (\bullet, \bullet) : (\hat{4}, \hat{3})
\end{array}
$$

$\diamond$

The new rules preserve all the properties proved for the original extension. The proofs are presented together with those of the next chapter, for the propositions involving the rules that have changed.

In summary, we have explored the role of partially static information in the specialization of dynamic case expressions. We have shown it is not only convenient but essential for achieving useful specializations in a big family of terms. Guided by this notion, we have introduced a slight modification to the rule for specializing case expressions which improves the use of static information. Our proposal takes better advantage of branch erasure and allows more valid specializations than the original formulation, keeping consistency and preserving all the good properties of the system.

Chapter 4

Type Specialization of Polyvariant Sums

The specialization of dynamic sum types we have presented so far can generate multiple copies of a data type, but for each copy, there can be at most one residual summand for each constructor in the corresponding source definition. For example, given the data declaration

$$data\ D^D\ = Only^D\ Int^S$$

the source expression

$$\left(Only^D\ 11^S\,, Only^D\ 4^S\right)^D : \left(D^D\,, D^D\right)^D$$

can be specialized and solved to

$$\begin{array}{l} data\ D^1\ = Only^1\ \hat{11} \\ data\ D^2\ = Only^2\ \hat{4} \end{array}$$

$$(Only^1\ \bullet, Only^2\ \bullet) : (D^1, D^2)$$

but the similar expression

$$\begin{array}{l} \mathbf{let}^D\ \ id = \lambda^D x.x\ \mathbf{in} \\ \quad \left(id\ @^D\ (Only^D\ 11^S), id\ @^D\ (Only^D\ 4^S)\right)^D : \left(D^D\,, D^D\right)^D \end{array}$$

cannot be specialized at all. Function id being monovariant, both tagged expressions must specialize to the same residual sum type D', but D' cannot be defined to have constructors $Only\ \hat{11}$ and $Only\ \hat{4}$ at the same time!

Certainly, function id could be made polyvariant so each tagged expression could belong to a different residual type — we would then have constructors $Only^1\ \hat{11}$ and $Only^2\ \hat{4}$ belonging to residual types D^1 and D^2 respectively, as before. But we could also build a new residual type D' having both summands, and let id specialize to type $D' \to D'$. We would then have the following residual code:

$$data\ D' = Only_1\ \hat{11} \mid Only_2\ \hat{4}$$

$$\begin{array}{l} \mathbf{let}\ id' = \lambda x'.x'\ \mathbf{in} \\ \quad (id'@(Only_1\ \bullet), id'@(Only_2\ \bullet)) : (D', D') \end{array}$$

$$\mathbf{let}^D\ f = \lambda^D x.\mathbf{lift}\ x\ \mathbf{in}\ \left(f\ @^D\ 11^S, f\ @^D\ 6^S\right)^D : \left(Int^D, Int^D\right)^D$$

Expression cannot be specialized

$$\begin{array}{l}\mathbf{let}^D\ f = \mathbf{poly}\ \lambda^D x.\mathbf{lift}\ x\\ \quad \mathbf{in}\ \left(\mathbf{spec}\ f\ @^D\ 11^S, \mathbf{spec}\ f\ @^D\ 6^S\right)^D\\ : \left(Int^D, Int^D\right)^D\end{array}$$

$$\begin{array}{l}\mathbf{let}\ f' = \Lambda h.\lambda x'.h\\ \quad \mathbf{in}\ (f'(\!(11)\!)@\bullet, f'(\!(6)\!)@\bullet)\\ : (Int, Int)\end{array}$$

Specialization via polyvariant functions

$$\begin{array}{l}\mathbf{let}^D\ f = \lambda^D px.\mathbf{case}^D\ px\ \mathbf{of}\\ \qquad\qquad Poly^D\ x\ \rightarrow\ \mathbf{lift}\ x\\ \quad \mathbf{in}\ \left(f\ @^D\ (Poly^D\ 11^S), f\ @^D\ (Poly^D\ 6^S)\right)^D\\ : \left(Int^D, Int^D\right)^D\end{array}$$

$$\begin{array}{l}\mathbf{let}\ f' = \lambda px'.\mathbf{case}\ px'\ \mathbf{of}\\ \qquad\qquad Poly_1\ x\ \rightarrow\ 11\\ \qquad\qquad Poly_2\ x\ \rightarrow\ 6\\ \quad \mathbf{in}\ (f@(Poly_1\ \bullet), f@(Poly_2\ \bullet))\\ : (Int, Int)\end{array}$$

Specialization via polyvariant sums

Figure 4.1: Alternatives for achieving polyvariance

Here function *id* needs not be polyvariant, because both arguments belong to the same residual type. Indeed, specializing dynamic tagged terms this way is an alternative approach for achieving polyvariance! In the expression above, constructor *Only* is applied to expressions with different residual types — 11^S and 4^S — to produce expressions with the same one — D'. Thus, they provide another way to pass different static arguments to the same function, *id* in this case. For this reason, sum types that can be specialized this way are called *polyvariant sum types*. Figure 4.1 compares the two approaches for polyvariance in a simple example.

In this chapter we extend the principal type specialization system to introduce polyvariant sum types. These are not presented as a modification to dynamic sum types but as an addition: both kinds of sum types are kept, identified by a keyword used in their declaration and specialized accordingly.

The extension involves adding new constructs to the term and type languages, in both the source and residual versions, to express polyvariant sum types. New predicates are introduced, as well as new rules for entailment, residual term reduction, specialization and residual typing. Our extension is mainly based on dynamic sum types (see section 2.5 and modifications in chapter 3) and takes elements from specialization of polyvariant functions as well (section 2.3.1).

4.1 Source language

We extend the source language to allow polyvariant sum declarations together with regular data declarations. As before, constructors are distinguished lexically and take only one argument. As we use letter D for regular dynamic sum types and K for their constructors, we will use letters Y and L for their polyvariant counterparts.

Definition 4.1 Let D and Y denote sum type names and K and L constructor names. A *source term*, denoted by e, is an element of the language defined by the following grammar:

$$
\begin{array}{lll}
e & ::= & [\ ddcl\ ]^*\ e_p \\
ddcl & ::= & \ldots \mid \textbf{polydata}\ Y^D\ =\ es \\
es & ::= & L_1^{\ D}\ \tau \mid\mid \ldots \mid\mid L_n^{\ D}\ \tau \\
e_p & ::= & \ldots \mid L^{\ D}\, e_p \\
 & \mid & \ldots \mid \textbf{case}^D\, e_p\ \textbf{of}\ [\ br_p\ ]^+ \\
br_p & ::= & L^D\ x\ \rightarrow\ e_p
\end{array}
$$

The structure of the grammar is the same as the one presented in section 2.5, with the only addition of the **polydata** keyword to declare polyvariant sum types. Polyvariant constructors can appear wherever regular dynamic ones can, and the **case** construct is the same for expressions of any of these types.

Source types are only extended with polyvariant sum names.

Definition 4.2 A *source type*, denoted by τ, is an element of the language defined by the following grammar:

$$\tau ::= Int^D \mid Int^S \mid (\tau, \ldots, \tau)^{\ D} \mid \tau \rightarrow^D \tau \mid \textbf{poly}\ \tau \mid D^D \mid Y^D$$

where the type $(\tau_1, \ldots, \tau_n)^{\ D}$ is a finite tuple for every possible arity n. The names D and Y cannot be names that already exist, like *Int*, etc.

4.2 Residual language

4.2.1 Residual terms

Residual terms also have a similar structure to those defined previously for dynamic data types.

Definition 4.3 Let D and Y denote sum type names and K and L constructor names. A *residual term*, denoted by e', is an element of the language defined by the following grammar:

$$
\begin{array}{lll}
e' & ::= & [\ ddcl'\ ]^*\ e'_p \\
ddcl' & ::= & \ldots \mid \textbf{data}\ Y^n\ =\ es' \\
es' & ::= & L_1^{v'_p}\ \tau \mid\mid \ldots \mid\mid L_n^{v'_p}\ \tau \\
e'_p & ::= & \ldots \mid L\ e'_p \\
 & \mid & \ldots \mid \textbf{case}\ e'_p\ \textbf{of}\ [\ br'_p\ ]^+ \\
 & \mid & \ldots \mid \textbf{polycase}_v\ e'_p\ \textbf{with}\ v'_p\ \textbf{and}\ [\ v'_p\ ]^+\ \textbf{of}\ [\ br''_p\ ]^+ \\
br'_p & ::= & L\ x\ \rightarrow\ e'_p \\
br''_p & ::= & L\ \rightarrow\ e'_p \\
v'_p & ::= & \ldots \mid \{L_{k,i}\}_{k\in I, i\in I'_k} \mid \begin{smallmatrix} n \\ i \end{smallmatrix} \mid \left\langle n, \left(v'_p\right)_{i\in I'} \right\rangle \mid v'_p \diamond v'_p
\end{array}
$$

Residual expressions involving sum-types may now be generated by polyvariant or monovariant definitions — they are both declared with the **data** keyword. Constructors generated by polyvariant sums can appear in tagged values and case expressions just as regular dynamic ones. Superscripts in L tags are used as a pair $\begin{smallmatrix} n \\ i \end{smallmatrix}$, where n distinguishes among all the possible data types generated by a single source declaration, and i identifies the different copies generated by a single source constructor. When $v = \begin{smallmatrix} n \\ i \end{smallmatrix}$, we will note L_k^v as $L_{k,i}^n$.

Example 4.4 Polyvariant sum types can generate multiple residual data types just as monovariant sum types can. They can also generate multiple versions of their constructors. The following expression

$$\textbf{polydata}\ P^D\ =\ \mathit{Poly}^D\ \mathit{Int}^S$$

$$\textbf{let}^D\ \mathit{id} = \lambda^D z.z\ \textbf{in}\\ \left(\mathit{id}\ @^D\ (\mathit{Poly}^D\ 11^S), \mathit{id}\ @^D\ (\mathit{Poly}^D\ 4^S), \mathit{Poly}^D\ 9^S\right)^D : \left(P^D, P^D, P^D\right)^D$$

has three appearances of constructor *Poly*: the first two must belong to the same residual type since they are both arguments of *id*, but the third appears independently. So the specialized code has two data declarations, the first of which has two copies of the constructor.

$$\mathit{data}\ P^1 = \mathit{Poly}_1^1\ \hat{1}1 \mid \mathit{Poly}_2^1\ \hat{4}\\ \mathit{data}\ P^2 = \mathit{Poly}_1^2\ \hat{9}$$

$$\textbf{let}\ \mathit{id}' = \lambda z'.z'\ \textbf{in}\\ (\mathit{id}'@(\mathit{Poly}_1^1\ \bullet), \mathit{id}'@(\mathit{Poly}_2^1\ \bullet), \mathit{Poly}_1^2\ \bullet) : (P^1, P^1, P^2)$$

◇

Specialization of a case expression on polyvariant sum types requires information about the residual data definition: if a constructor does not appear in it, the branches must be erased, and if it does, a different branch must be built for each of its versions. Now all this information might not be available if the expression is specialized independently. The $\textbf{polycase}_v$ structure — just as the $\textbf{protocase}_v$ for regular dynamic sum types — expresses the result of specializing a case construct when some information is missing, and can be reduced to a residual **case** expression as soon as it is possible — see section 4.2.4.

New forms of evidence are necessary for gathering this information. Described by v_p' in the grammar above, they are explained in detail in section 4.2.3.

4.2.2 Residual types

Residual types are extended with polyvariant sum-type names and with new predicates.

Definition 4.5 Let t denote a *type variable* from a countably infinite set of variables and s a *type scheme variable* from another countably infinite set of variables, all of them disjoint with any other set of variables already used. A *residual type*, denoted by τ', is an element of the language given by the grammar

$$\begin{array}{rcl}
\tau' & ::= & t \mid \mathit{Int} \mid \hat{n} \mid \tau' \to \tau' \mid (\tau', \ldots, \tau') \mid \textbf{poly}\ \sigma \mid D^n \mid Y^n \\
\rho & ::= & \delta \Rightarrow \rho \mid \tau' \\
\sigma & ::= & s \mid \forall s.\sigma \mid \forall t.\sigma \mid \rho \\
\delta & ::= & \text{IsInt}\ \tau' \mid \tau' := \tau' + \tau' \mid \text{IsMG}\ \sigma\ \sigma \mid \delta_d \mid \delta_e \\
\delta_d & ::= & \text{IsSum}\ \tau' \mid \text{HasC}\ \tau'\ \ K_k\ \ \tau' \mid K \in \tau' \,?\, \delta \\
\delta_e & ::= & \text{IsPolySum}\ \tau' \\
 & \mid & \text{HasMGC}\ \tau'\ \ L_k\ \ \sigma \\
 & \mid & \text{HasMGBr}\ \tau'\ \ L_k\ \ \sigma\ \ \tau' \\
 & \mid & \text{HasPolyC}\ \tau'\ \ L_k\ \ \tau' \\
 & \mid & L \in \tau' \,?\, \delta
\end{array}$$

The predicates described by δ_e express relationships between residual types generated from a polyvariant sum and their constructors. Intuitively, they have the following meaning:

- IsPolySum τ' is true whenever type τ' is a residual sum type generated from a polyvariant sum definition.
- HasMGC $\tau' \;\; L_k \;\; \sigma$ is true if τ' is a residual type including summands $L^n_{k,1}\ \tau'_1, \ldots,$ $L^n_{k,m}\ \tau'_m$ such that type scheme σ is more general than τ'_i for $i = 1, \ldots, m$.
- HasMGBr $\tau' \;\; L_k \;\; \sigma \;\; \tau''$ is true if τ' is a residual type including summands $L^n_{k,1}\ \tau'_1, \ldots,$ $L^n_{k,m}\ \tau'_m$ such that type scheme σ is more general than the function type $\tau'_i \to \tau''$ for $i = 1, \ldots, m$.
- HasPolyC $\tau'_1 \;\; L_k \;\; \tau'_2$ is true if type τ'_1 is a residual sum type that includes a summand $L^n_{k,i}\ \tau'_2$.

4.2.3 Entailment and evidence

The idea behind the predicates is formalized in the entailment relation, whose new rules are stated in figure 4.2.

New notation is introduced for some of these rules. For a residual sum type declaration

$$data\ Y^n = L^n_{1,1}\ \tau'_{1,1} \mid L^n_{1,2}\ \tau'_{1,2} \mid \ \ldots \ \mid L^n_{k,i}\ \tau'_{k,i} \mid \ \ldots \ \mid L^n_{m,j}\ \tau'_{m,j}$$

we define $Y^n(L_{k,i}\,)$ to be $\tau'_{k,i}$, that is, the argument of the constructor $L^n_{k,i}$, whereas $\alpha_k(Y^n)$ denotes the number of copies of constructor L_k in Y^n's declaration.

Entailment rules describe under what conditions a predicate can be proved, providing evidence for it when necessary. Of the eight rules presented in figure 4.2, the first four reflect the basic meaning of each predicate:

- IsPolySum τ' can only be proved if τ' is a sum type generated from a polyvariant source declaration, and the evidence for this is the set of constructor names in its definition.
- HasMGC $\tau' \;\; L_k \;\; \sigma$ can be proved if σ can be proved to be more general than any argument of a constructor L_k in τ'.
- HasMGBr $\tau' \;\; L_k \;\; \sigma \;\; \tau''$ can be proved if τ' is a sum type Y^n and σ can be proved to be more general than function $\tau'_{k,i} \to \tau''$ for any $L^n_{k,i}\ \tau'_{k,i}$ in τ''s declaration. The evidence has two components: the first one is n and the second is an enumeration of the evidence proving the more general relation between σ and each function.

  This predicate is used to express constraints on the possible branches of a case expression on τ' — see explanation of rule (POLYCASE) in section 4.4. Intuitively, it constrains the arguments of each constructor L_k so that all the residual branches corresponding to this tag can be correctly generated. Whereas predicate HasMGC constrains the residual arguments themselves, predicate HasMGBr places restrictions on the branches they can generate.
- HasPolyC $\tau' \;\; L_k \;\; \tau''$ can be proved if τ' is a sum type Y^n, and there exists $L^n_{k,i}\ \tau_{k,i}$ in its declaration such that $\tau_{k,i}$ and τ'' unify. The evidence is then the pair ${}^{n}_{i}$.

$$\text{(IsPolySum)}\quad \frac{Y^n \text{ is defined as } \left\{L^n_{k,i}\ \tau'_{k,i}\right\}_{k\in I, i\in I'_k}}{\Delta \Vdash \{L_{k,i}\}_{k\in I, i\in I'_k} : \text{IsPolySum } Y^n}$$

$$\text{(HasMGC)}\quad \frac{(\Delta \Vdash \text{IsMG } \sigma'\ Y^n(L_{k,i}\ ))_{i=1,\ldots,\alpha_k(Y^n)}}{\Delta \Vdash \text{HasMGC } Y^n\ \ L_k\ \ \sigma'}$$

$$\text{(HasMGBr)}\quad \frac{\left(\Delta \Vdash v_{k,i} : \text{IsMG } \sigma'\ (Y^n(L_{k,i}\ ) \to \tau')\right)_{i=1,\ldots,\alpha_k(Y^n)}}{\Delta \Vdash \langle n, (v_{k,i})_{i=1,\ldots,\alpha_k(Y^n)}\rangle : \text{HasMGBr } Y^n\ \ L_k\ \ \sigma'\ \ \tau'}$$

$$\text{(HasPolyC)}\quad \frac{\Delta \Vdash Y^n(L_{k,i}\ ) \sim \tau'}{\Delta \Vdash \begin{smallmatrix} n \\ i \end{smallmatrix} : \text{HasPolyC } Y^n\ \ L_k\ \ \tau'}$$

$$\text{(Comp-MGC)}\quad \frac{\Delta \Vdash \text{HasMGC } \tau'\ \ L_k\ \ \sigma'_2 \quad \Delta \Vdash \text{IsMG } \sigma'_1\ \sigma'_2}{\Delta \Vdash \text{HasMGC } \tau'\ \ L_k\ \ \sigma'_1}$$

$$\text{(Comp-MGBr)}\quad \frac{h : \Delta \Vdash v : \text{HasMGBr } \tau'\ \ L_k\ \ \sigma'\ \ \tau'_r \quad h : \Delta \Vdash v' : \text{IsMG } \sigma\ \sigma'}{h : \Delta \Vdash v \diamond v' : \text{HasMGBr } \tau'\ \ L_k\ \ \sigma\ \ \tau'_r}$$

$$\text{(HasPolyC-Guard)}\quad \frac{\Delta, \text{HasPolyC } \tau'\ \ L_k\ \ \tau'', \Delta' \Vdash v : \Delta''}{\Delta, \text{HasPolyC } \tau'\ \ L_k\ \ \tau'', \Delta' \Vdash v : L_k \in \tau'\,?\,\Delta''}$$

$$\text{(IsPolySum-Guard)}\quad \frac{\bar{h} : \Delta \Vdash \bar{v}' : \Delta' \quad \Delta'' \Vdash v^y : \text{IsPolySum } \tau'}{\Delta'', \bar{h} : L_k \in \tau'\,?\,\Delta \Vdash \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ \bar{v}'\ \mathbf{else}\ \bullet : L_k \in \tau'\,?\,\Delta'}$$

Figure 4.2: Entailment rules for predicates involving polyvariant sums

$$\left\langle n, (v'_i)_{i\in I}\right\rangle \diamond v \ \triangleright\ \left\langle n, (v'_i \circ v)_{i\in I}\right\rangle$$

$$\begin{array}{l}\textbf{polycase}_v\ e'\ \textbf{with}\ \{L_{k,i}\}_{k\in I, i\in I'_k}\ \textbf{and}\ \left(\left\langle n_j, (v_{j,i})_{i\in I'_j}\right\rangle\right)_{j\in B} \\ \quad \textbf{of}\ \left(L_j\ \rightarrow\ e''_j\right)_{j\in B} \\ \triangleright \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \left(v_{j,i}[e''_j] \triangleright \lambda x'_{j,i}.e'_{j,i}\right) \\ \textbf{case}\ e'\ \textbf{of} \\ \quad \left(L^{n_j}_{j,i}\ x'_{j,i}\ \rightarrow\ e'_{j,i}\right)_{j\in(B\cap I), i\in I'_j}\end{array}$$

Figure 4.3: Reduction rules for parallel composition and the $\textbf{polycase}_v$ construct

The other four rules are just natural extensions to the system for dynamic sum types. Rule (Comp-MGC) gives an alternative way for proving HasMGC $\tau'\ L_k\ \sigma$ by proving it for a type scheme σ' and proving σ is more general than σ'. Rule (Comp-MGBr) is the equivalent for predicate HasMGBr — evidence $v_1 \diamond v_2$ represents composition of a single conversion with an enumeration of them, see reduction rules in figure 4.3. Rule (HasPolyC-Guard) expresses that when HasPolyC $\tau'\ L_k\ \tau''$ holds, we are sure that L_k belongs to τ'. Finally, rule (IsPolySum-Guard) provides evidence for a conditional predicate involving a polyvariant sum type when we have no hypothesis as to whether the condition holds.

The predicates and entailment rules respect the good properties of Mark Jones's abstract predicate system [1994] (see section 1.3), and in particular, rules (Trans), (Close), (Evars) and (Cut), which will be repeatedly used in our proofs. Substitution of type variables in predicates and of evidence variables in expressions is straightforward — see extensions in appendices A.4 and A.5.

The following lemma allows applying rule (IsPolySum-Guard) with a little more flexibility.

Lemma 4.6 *If* $\bar{h} : \Delta \Vdash v^y : \text{IsPolySum}\ \tau'$,
$\bar{h}_1 : \Delta'_1 \Vdash \bar{v}_2 : \Delta'_2$ *and*
$\bar{h} : \Delta \Vdash \bar{v}_1 : L_k \in \tau' \,?\, \Delta'_1$
then $\bar{h} : \Delta \Vdash \bar{v}' : L_k \in \tau' \,?\, \Delta'_2$
where $\bar{v}' = \textbf{if}_v\ L_k \in v^y\ \textbf{then}\ \bar{v}_2[\bar{v}_1/\bar{h}_1]\ \textbf{else}\ \bullet$

4.2.4 Reduction of residual terms

As we have mentioned, a $\textbf{polycase}_v$ expression can be reduced to a residual case expression when all the required information is available. Figure 4.3 shows the reduction rule, together with a reduction rule for one of the evidence expressions introduced.

The evidence construct $v_1 \diamond v_2$ represents parallel composition. It can be reduced when v_2 is of the form $\langle n, (v'_i)_{i\in I}\rangle$, and the result is composing v with each v'_i.

A $\textbf{polycase}_v$ construct that can be reduced has four elements:

- The control expression e'.
- A constructor set $\{L_{k,i}\}_{k\in I, i\in I'_k}$. It is meant to be the set of constructors that make up the type of e'.

- For each branch L_j, a *modified* branch $L_j \ \rightarrow \ e''_j$. Here, the variable that is normally bound by pattern matching is embedded in e''_j.
- For each branch L_j, a piece of evidence $\left\langle n_j, \left(v'_{j,i}\right)_{i \in I'_j} \right\rangle$ where n is an index and each $v'_{j,i}$ represents a conversion such that $v'_{j,i}[e''_j]$ reduces to a residual lambda expression $\lambda x'_{j,i}.e'_{j,i}$.

The reduction has two main aspects. Firstly, only the branches corresponding to the constructors that are actually part of the sum type remain — branches in the resulting **case** expression are those for $j \in (I \cap B)$, just as in the $\mathbf{protocase}_v$ reduction rule. Secondly, for the constructors that do remain, a separate branch is generated for each conversion available — there is intended to be one per constructor copy. After applying the conversions, variables $x'_{j,i}$ can be again bound by pattern matching.

Example 4.7 In the following expression

$$\begin{array}{l} \mathbf{polycase}_v \ e' \ \mathbf{with} \ \{Poly_1, Poly_2\} \ \mathbf{and} \ \langle 1, ([]((6)), []((11)))\rangle \\ \quad \mathbf{of} \ Poly \ \rightarrow \ \Lambda h.\lambda x.h \end{array}$$

there are two conversions for a single constructor. Both of them can be applied to the body of the branch, yielding expressions $\lambda x.6$ and $\lambda x.11$ respectively. Then the term reduces to

$$\begin{array}{lll} \mathbf{case} \ e' & & \\ \quad \mathbf{of} & Poly_1^1 \ x & \rightarrow \ 6 \\ & Poly_2^1 \ x & \rightarrow \ 11 \end{array}$$

$\diamond$

Additionally, the equivalence of residual terms is extended to handle $\mathbf{polycase}_v$ expressions with free evidence variables, that cannot be reduced. See appendix A.6 for details.

4.3 Residual typing

System RT specifies the typing of residual terms. We extend it with two rules for type checking tagged and case expressions on polyvariant sum types — the rules are presented in figure 4.4. They have a natural correspondence with specialization rules in section 4.4.2 so they will not be further explained here.

The following lemma does not appear in Martínez López's work but is necessary for some of our proofs.

Lemma 4.8 *If* $\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma$ *then* $EV(e') \subseteq \bar{h}$

The proofs of the following propositions are extended for the new rules. The first one states that an RT judgment can be weakened by strengthening the predicate context; the second shows that a conversion between two type schemes indeed relates them in their contexts.

Proposition 4.9 *If* $\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma$ *and* $\Delta' \Vdash \bar{v} : \Delta$ *then* $\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'[\bar{v}/\bar{h}] : \sigma$

$$\text{(RT-POLYCONSTR)}\quad \frac{\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : \tau'_j \qquad \Delta \Vdash \text{IsPolySum } \tau' \quad \Delta \Vdash v_j : \text{HasPolyC } \tau' \ L_j \ \tau'_j}{\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} L_j^{v_j} \ e' : \tau'}$$

$$\text{(RT-POLYCASE)}\quad \frac{\begin{array}{l}\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : \tau'_e \\ \Delta \Vdash v^y : \text{IsPolySum } \tau'_e \\ \left(\begin{array}{l} \bar{h}_k : \Delta_k \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e''_k : \sigma_k \\ \Delta \Vdash \bar{v}_k : L_k \in \tau'_e \,?\, \Delta_k \\ \Delta \Vdash w_k : L_k \in \tau'_e \,?\, \text{HasMGBr } \tau'_e \ L_k \ \sigma_k \ \tau' \end{array}\right)_{k \in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} \ \textbf{polycase}_v \ e' \ \textbf{with} \ v^y \ \textbf{and} \ (w_k)_{k \in B} \ \textbf{of} \\ \qquad\qquad \left(L_k \ \rightarrow \ e''_k[\bar{v}_k/\bar{h}_k]\right)_{k \in B} : \tau'\end{array}}$$

Figure 4.4: Residual typing rules involving polyvariant sum types

$$\text{(SR-POLYDATA)}\quad \frac{\Delta \Vdash \text{IsPolySum } \tau' \qquad \left(\begin{array}{l} \Delta_k \vdash_{\text{SR}} Y(L_k) \ \hookrightarrow \ \sigma_k \\ \Delta \Vdash L_k \in \tau' \,?\, \Delta_k \\ \Delta \Vdash L_k \in \tau' \,?\, \text{HasMGC } \tau' \ L_k \ \sigma_k \end{array}\right)_{L_k \in Y}}{\Delta \vdash_{\text{SR}} Y^D \ \hookrightarrow \ \tau'}$$

Figure 4.5: Source-residual relation for polyvariant sum types

Theorem 4.10 *If* $\bar{h} : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : \sigma$, *and* $C : (\bar{h} : \Delta \mid \sigma) \geq (\bar{h}' : \Delta' \mid \sigma')$ *then* $\bar{h}' : \Delta' \mid \Gamma_{\text{R}} \vdash_{\text{RT}} C[e'] : \sigma'$

4.4 Specialization rules

4.4.1 SR Relation

The SR relation associates source types with residual types that can be generated from them. It is one of the two relations that make up the specialization system, and is essential to the achievement of principality as shown by Martínez López [2002; 2005]. Our extension to the SR system consists of just one rule, specifying which residual types can be obtained from the only source type we have added — Y^D. The rule is presented in figure 4.5. We define $Y(L_k)$ as the argument of constructor L_k in the (source) definition of Y.

Rule (SR-POLYDATA) specifies that an expression of residual type τ' can be generated from one of source type Y^D if:

1. τ' has been generated from a polyvariant sum type;
2. For every L_k in the definition of τ', there is a type scheme σ_k SR-related to the source argument of L_k and more general than any of the residual arguments of L_k.

The conditional predicates in the second condition restrict it to apply only to constructors

$$\text{(POLYCONSTR)}\quad \frac{\Delta \vdash_{\text{SR}} Y^D \;\hookrightarrow\; \tau'_e \qquad \Delta \mid \Gamma \vdash_{\text{P}} e : Y(L_j) \;\hookrightarrow\; e' : \tau'_j \qquad \Delta \Vdash v_j : \text{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j}{\Delta \mid \Gamma \vdash_{\text{P}} L_j^{\,D}\ e : Y^D \;\hookrightarrow\; L_j^{\,v_j}\ e' : \tau'_e}$$

$$\text{(POLYCASE)}\quad \frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\text{P}} e : Y^D \;\hookrightarrow\; e' : \tau'_e \\ \Delta \Vdash v^y : \text{IsPolySum}\ \tau'_e \\ \Delta \vdash_{\text{SR}} \tau \;\hookrightarrow\; \tau' \\ \left(\begin{array}{l}\bar{h}_k : \Delta_k \mid \Gamma \vdash_{\text{P}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \;\hookrightarrow\; e'_k : \sigma_k \\ \Delta \Vdash \bar{v}_k : L_k \in \tau'_e\,?\,\Delta_k \\ \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\text{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'\end{array}\right)_{k\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\text{P}} \ \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^{\,D}\ \ x_k\ \ \to\ \ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ \ (w_k)_{k\in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ \ \left(L_k\ \ \to\ \ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B} : \tau'\end{array}}$$

Figure 4.6: Specialization rules involving polyvariant sums

actually present in the definition of τ'. For the ones that do not, premises are trivially proved with evidence $\bullet$ — see entailment rules in figure 4.2.

System SR has several useful properties, which are preserved by our addition.

Proposition 4.11 *If* $\Delta \vdash_{\text{SR}} \tau \;\hookrightarrow\; \sigma$ *then* $S\,\Delta \vdash_{\text{SR}} \tau \;\hookrightarrow\; S\,\sigma$

Proposition 4.12 *If* $\Delta \vdash_{\text{SR}} \tau \;\hookrightarrow\; \sigma$ *and* $\Delta' \Vdash \Delta$ *then* $\Delta' \vdash_{\text{SR}} \tau \;\hookrightarrow\; \sigma$

Theorem 4.13 *If* $\Delta \vdash_{\text{SR}} \tau \;\hookrightarrow\; \sigma$ *and* $C : (\bar{h} : \Delta \mid \sigma) \geq (\bar{h}' : \Delta' \mid \sigma')$ *then* $\Delta' \vdash_{\text{SR}} \tau \;\hookrightarrow\; \sigma'$

We also add the following lemma, necessary for some of our proofs.

Lemma 4.14 *If* $\Delta \vdash_{\text{SR}} Y^D \;\hookrightarrow\; \sigma$ *then* $\sigma = \forall\bar{\beta}.\Delta' \Rightarrow \tau$ *and* $\Delta, \Delta' \Vdash \text{IsPolySum}\ \tau$

4.4.2 P Relation

Finally, we extend specialization rules to specialize tagged and case expressions that involve polyvariant sums. These are presented in figure 4.6.

Rule (POLYCONSTR) specifies how a constructor expression from a polyvariant sum type is specialized. Firstly, the residual type has to be SR-related to the source type; this premise is analogous to the one in rule (DCONSTR) for specializing dynamic sum types (section 2.5.3) and has the same purpose — to rule out undesirable specializations. Additionally, the expression e must specialize to an expression e' of residual type τ'_j such that the residual

sum type must include a summand $L_j\ \tau'_j$ in its definition. Predicate HasPolyC expresses this last condition; evidence v_j is used to obtain the right copy of both the data definition and the constructor.

Rule (POLYCASE) specifies the specialization of case expressions. The first two premises involve the specialization of the control expression e — it must specialize to an expression with a residual type τ'_e that can be proved to be generated from a polyvariant sum. The next premise states that the residual type τ' must be SR-related to the source type. The next three conditions must hold for every branch in the case expression, that is for each k in the set B of branch numbers. The pattern variable x_k is bound to the body e_k and specialized as a dynamic function — notice the resulting type is not syntactically restricted to be a function type as in rule (DCASE) but can be any type scheme. The specialized function is used to build the branches of the resulting $\mathbf{polycase}_v$ expression. Now since each constructor can generate more than one copy, each branch could be replicated. The last premise uses predicate HasMGBr to constrain every possible branch: seen as functions, they must all be a suitable instance of σ'_k, and all bodies must have residual type τ', the type of the final specialized expression — see entailment rule (HasMGBr) in figure 4.2. Evidence for this predicate is also a part of the $\mathbf{polycase}_v$ construct and is used for generating the correct branches when reducing it — see figure 4.3.

As in rule (DCASE), guarded predicates are used to discard the branches corresponding to constructors that do not appear in the definition of τ'_e. Evidence substitution in the body of the specialized branches relates Δ to Δ_k.

Example 4.15 Specialization of polyvariant sum expressions can lead to multiple data declarations as in regular dynamic sums, and also to replication of the constructors in a single declaration.

$$\textbf{polydata}\ P^D\ =\ \mathit{Poly}^D\ \mathit{Int}^S$$

$$\begin{array}{ll}\vdash_{\mathrm{P}} & \left(\mathit{Poly}^D\ 11^S, \mathit{Poly}^D\ 4^S\right)^D : \left(P^D, P^D\right)^D \hookrightarrow \\ & \Lambda h_1, h_2, h_3, h_4, h_5, h_6. \\ & (\mathit{Poly}^{h_3}\ \bullet, \mathit{Poly}^{h_6}\ \bullet) : \forall t_1, t_2.\ h_1 : \mathrm{IsPolySum}\ t_1, \\ & \qquad\qquad h_2 : \mathrm{HasMGC}\ t_1\ \ \mathit{Poly}\ \ (\forall t.\mathrm{IsInt}\ t \Rightarrow t), \\ & \qquad\qquad h_3 : \mathrm{HasPolyC}\ t_1\ \ \mathit{Poly}\ \ \hat{1}\hat{1}, \\ & \qquad\qquad h_4 : \mathrm{IsPolySum}\ t_2, \\ & \qquad\qquad h_5 : \mathrm{HasMGC}\ t_2\ \ \mathit{Poly}\ \ (\forall t.\mathrm{IsInt}\ t \Rightarrow t), \\ & \qquad\qquad h_6 : \mathrm{HasPolyC}\ t_2\ \ \mathit{Poly}\ \ \hat{4} \qquad\qquad \Rightarrow (t_1, t_2)\end{array}$$

If constraint solving detects t_1 and t_2 are independent, two different types are generated: h_3 is assigned value $\begin{smallmatrix}1\\1\end{smallmatrix}$ and h_6, $\begin{smallmatrix}2\\1\end{smallmatrix}$. The rest of the predicates are successfully verified but the evidence is not used in the expression.

$$\begin{array}{l}\mathit{data}\ P^1 = \mathit{Poly}^1_1\ \ \hat{1}\hat{1} \\ \mathit{data}\ P^2 = \mathit{Poly}^2_1\ \ \hat{4}\end{array}$$

$$(\mathit{Poly}^1_1\ \ \bullet, \mathit{Poly}^2_1\ \ \bullet) : (P^1, P^2)$$

If, on the contrary, t_1 and t_2 must unify, a single data type must be built: h_3 and h_6 are assigned values $\begin{smallmatrix}1\\1\end{smallmatrix}$ and $\begin{smallmatrix}1\\2\end{smallmatrix}$ respectively.

$$data\ P^1 = Poly_1^1\ \hat{1}1 \mid Poly_2^1\ \hat{4}$$

$$(Poly_1^1\ \bullet, Poly_2^1\ \bullet) : (P^1, P^1)$$

$\diamond$

Example 4.16 Since polyvariant sum types can generate more summands than there are in the source definition, constructors do not cause information flow between their arguments. Compare this to example 2.30.

$$\textbf{polydata}\ P^D\ =\ Poly^D\ Int^S$$

$$\begin{array}{ll}
\vdash_{\mathrm{P}} & \textbf{let}^D\ id = \lambda^D z.z \\
& \quad \textbf{in}\ \lambda^D x.\big(id\ @^D\ (Poly^D\ x), id\ @^D\ (Poly^D\ 4^S), \textbf{lift}\ x\big)^{\,D} : Int^S \to^D \big(P^D, P^D, Int^D\big)^{\,D} \\
& \hookrightarrow \\
& \Lambda h_1, h_2, h_3, h_4, h_5. \\
& \textbf{let}\ id' = \lambda z'.z' \\
& \quad \textbf{in}\ \lambda x'.(id'@(Poly^{h_4}\ x'), id'@(Poly^{h_5}\ \bullet), h_1) \\
& \qquad : \forall t_1, t_2.\ h_1 : \mathrm{IsInt}\ t_2, \\
& \qquad\qquad h_2 : \mathrm{IsPolySum}\ t_1, \\
& \qquad\qquad h_3 : \mathrm{HasMGC}\ t_1\ \ Poly\ \ (\forall t.\mathrm{IsInt}\ t \Rightarrow t), \\
& \qquad\qquad h_4 : \mathrm{HasPolyC}\ t_1\ \ Poly\ \ t_2, \\
& \qquad\qquad h_5 : \mathrm{HasPolyC}\ t_1\ \ Poly\ \ \hat{4} \Rightarrow t_2 \to (t_1, t_1, Int)
\end{array}$$

Here, constructor *Poly* on t_1 must have at least arguments t_2 and $\hat{4}$, but being polyvariant, they do not have to be the same. The function could be, for instance, applied to 3^S:

$$\begin{array}{ll}
\vdash_{\mathrm{P}} & \textbf{let}^D\ id = \lambda^D z.z \\
& \quad \textbf{in}\ \big(\lambda^D x.\big(id\ @^D\ (Poly^D\ x), id\ @^D\ (Poly^D\ 4^S), \textbf{lift}\ x\big)^{\,D}\big)\ @^D\ 3^S \\
& : \big(P^D, P^D, Int^D\big)^{\,D} \\
& \hookrightarrow \\
& \Lambda h_2, h_3, h_4, h_5. \\
& \textbf{let}\ id' = \lambda z'.z' \\
& \quad \textbf{in}\ \big(\lambda x'.(id'@(Poly^{h_4}\ x'), id'@(Poly^{h_5}\ \bullet), 3)\big)\ @\bullet \\
& \qquad : \forall t_1.\ h_2 : \mathrm{IsPolySum}\ t_1, \\
& \qquad\qquad h_3 : \mathrm{HasMGC}\ t_1\ \ Poly\ \ (\forall t.\mathrm{IsInt}\ t \Rightarrow t), \\
& \qquad\qquad h_4 : \mathrm{HasPolyC}\ t_1\ \ Poly\ \ \hat{3}, \\
& \qquad\qquad h_5 : \mathrm{HasPolyC}\ t_1\ \ Poly\ \ \hat{4} \Rightarrow (t_1, t_1, Int)
\end{array}$$

which can be solved and reduced with no problems to

$$data\ P^1 = Poly_1^1\ \hat{3} \mid Poly_2^1\ \hat{4}$$

$$(\textbf{let}\ id' = \lambda z'.z' \quad \textbf{in}\ \lambda x'.(id'@(Poly_1^1\ x'), id'@(Poly_2^1\ \bullet), 3))@\bullet : (P^1, P^1, Int)$$

$\diamond$

Example 4.17 Polyvariant sums are an alternative way to achieve polyvariance. Constructors are applied to static arguments of different types to obtain the same one, and the correct argument is chosen among the branches that have been replicated. The source expression

$$\begin{array}{l}\textbf{polydata}\ P^D\ =\ Poly^D\ Int^S \\ \\ \textbf{let}^D\ f = \lambda^D px.\textbf{case}^D\, px\ \textbf{of} \\ \qquad\qquad\qquad Poly^D\ x\ \rightarrow\ \textbf{lift}\ x \\ \quad \textbf{in}\ \left(f\ @^D\,(Poly^D\ 11^S\,), f\ @^D\,(Poly^D\ 6^S\,)\right)^D : \left(Int^D\,, Int^D\right)^D\end{array}$$

specializes to

$$\begin{array}{l}\Lambda h_1, h_2, h_3, h_4, h_5. \\ \textbf{let}\ f' = \lambda px'.\textbf{polycase}_v\ px'\ \textbf{with}\ h_1\ \textbf{and}\ h_2\ \textbf{of} \\ \qquad\qquad\qquad Poly\ \rightarrow\ \Lambda h.\lambda x.h \\ \quad \textbf{in}\ (f@(Poly^{h_4}\bullet), f@(Poly^{h_5}\bullet)) \\ : \forall t.\ h_1 : \text{IsPolySum}\ t, \\ \qquad h_2 : \text{HasMGBr}\ t\ \ Poly\ \ (\forall t'.\text{IsInt}\ t' \Rightarrow t' \rightarrow Int)\ \ Int, \\ \qquad h_3 : \text{HasMGC}\ t\ \ Poly\ \ (\forall t'.\text{IsInt}\ t' \Rightarrow t'), \\ \qquad h_4 : \text{HasPolyC}\ t\ \ Poly\ \ \hat{1}\hat{1}, \\ \qquad h_5 : \text{HasPolyC}\ t\ \ Poly\ \ \hat{6} \Rightarrow (Int, Int)\end{array}$$

Constraint solving can detect there is only one type with two constructors and assign values $\begin{smallmatrix}1\\1\end{smallmatrix}$ and $\begin{smallmatrix}1\\2\end{smallmatrix}$ to h_4 and h_5 respectively. Then h_1 can be associated with the set $\left\{Poly_1^1, Poly_2^1\right\}$. Predicate HasMGC is verified, since both $\hat{1}\hat{1}$ and $\hat{6}$ are instances of $(\forall t'.\text{IsInt}\ t' \Rightarrow t')$. Finally, predicate HasMGBr must be proved by finding a conversion from the type scheme to the function $\tau \rightarrow Int$, for each argument τ of $Poly$ — namely for $\hat{1}\hat{1}$ and $\hat{6}$. It can proved with

$$\begin{array}{l}[]((11)) : (\forall t'.\text{IsInt}\ t' \Rightarrow t' \rightarrow Int) \geq (\hat{1}\hat{1} \rightarrow Int) \\ []((6)) : (\forall t'.\text{IsInt}\ t' \Rightarrow t' \rightarrow Int) \geq (\hat{6} \rightarrow Int)\end{array}$$

So h_2 is assigned value $\langle 1, ([]((11)), []((6)))\rangle$. The residual expression and type can then be solved to

$$\begin{array}{l}data\ P^1 = Poly_1^1\ \ \hat{1}\hat{1} \mid Poly_2^1\ \ \hat{6} \\ \\ \textbf{let}\ f' = \lambda px'.\textbf{polycase}_v\ px'\ \textbf{with}\ \left\{Poly_1^1, Poly_2^1\right\}\ \textbf{and}\ \langle 1, ([]((11)), []((6)))\rangle\ \textbf{of} \\ \qquad\qquad\qquad Poly\ \rightarrow\ \Lambda h.\lambda x.h \\ \quad \textbf{in}\ (f@(Poly_1^1\ \bullet), f@(Poly_2^1\ \bullet)) : (Int, Int)\end{array}$$

and reduced to

$$\begin{array}{l}data\ P^1 = Poly_1^1\ \ \hat{1}\hat{1} \mid Poly_2^1\ \ \hat{6} \\ \\ \textbf{let}\ f' = \lambda px'.\textbf{case}\ px'\ \textbf{of} \\ \qquad\qquad\qquad\qquad Poly_1^1\ \ x\ \rightarrow\ 11 \\ \qquad\qquad\qquad\qquad Poly_2^1\ \ x\ \rightarrow\ 6 \\ \quad \textbf{in}\ (f@(Poly_1^1\ \bullet), f@(Poly_2^1\ \bullet)) : (Int, Int)\end{array}$$

◇

Example 4.18 In the following source expressions, we will repeatedly use the following data definition and function:

$$\textbf{polydata}\ \mathit{Either}^D\ =\ \mathit{Left}^D\ \mathit{Int}^S\ \mid \mathit{Right}^D\ \mathit{Int}^S$$

$$\begin{aligned} f = \lambda^D ex.\textbf{case}^D\ ex\ \textbf{of}\\ &\mathit{Left}^D\ x\ \rightarrow\ x\\ &\mathit{Right}^D\ x\ \rightarrow\ x +^S 1^S \end{aligned}$$

f is a function of source type $\mathit{Either}^D \rightarrow^D \mathit{Int}^S$.

Rules for specializing a case expression are designed so that all the branches in the residual version have the same type. This is ensured by predicate HasMGBr.

$$\begin{aligned} \vdash_{\mathrm{P}}\ & \textbf{let}^D\ f = \ldots\ \textbf{in}\ \left(f\ @^D\ (\mathit{Left}^D\ 4^S), f\ @^D\ (\mathit{Right}^D\ 3^S)\right)^D : \left(\mathit{Int}^S, \mathit{Int}^S\right)^D\\ & \hookrightarrow\ \Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8.\\ & \textbf{let}\ f' = \lambda ex.\textbf{polycase}_v\ ex\ \textbf{with}\ h_2\ \textbf{and}\ (h_3, h_4)\ \textbf{of}\\ & \qquad \mathit{Left}\ \rightarrow\ \Lambda h.\lambda x'.x'\\ & \qquad \mathit{Right}\ \rightarrow\ \Lambda h, h'.\lambda x'.\bullet\\ & \textbf{in}\ \left(f'@(\mathit{Left}^{h_7}\ \bullet), f'@(\mathit{Right}^{h_8}\ \bullet)\right)^D\\ & : \forall t_e, t_r.\ h_1 : \mathrm{IsInt}\ t_r,\\ & \qquad h_2 : \mathrm{IsPolySum}\ t_e,\\ & \qquad h_3 : \mathrm{HasMGBr}\ t_e\ \mathit{Left}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \rightarrow t)\ t_r,\\ & \qquad h_4 : \mathrm{HasMGBr}\ t_e\ \mathit{Right}\ (\forall t, t'.\mathrm{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \rightarrow t')\ t_r,\\ & \qquad h_5 : \mathrm{HasMGC}\ t_e\ \mathit{Left}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t),\\ & \qquad h_6 : \mathrm{HasMGC}\ t_e\ \mathit{Right}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t),\\ & \qquad h_7 : \mathrm{HasPolyC}\ t_e\ \mathit{Left}\ \hat{4},\\ & \qquad h_8 : \mathrm{HasPolyC}\ t_e\ \mathit{Right}\ \hat{3} \Rightarrow (t_r, t_r) \end{aligned}$$

The last two predicates indicate that t_e must be a sum type — let us call it Either^1 — with summands $\mathit{Left}_1^1\ \hat{4}$ and $\mathit{Right}_1^1\ \hat{3}$. In order to prove the third predicate, function $\hat{4} \rightarrow t_r$ must be an instance of $(\forall t.\mathrm{IsInt}\ t \Rightarrow t \rightarrow t)$, so t_r must unify with $\hat{4}$. Evidence h_3 is then assigned value $\langle 1, [](\!(4)\!)\rangle$. Now for the fourth predicate, function $\hat{3} \rightarrow \hat{4}$ must be an instance of $(\forall t, t'.\mathrm{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \rightarrow t')$, which actually holds. Evidence h_4 is then assigned the value $\langle 1, [](\!(3)\!)(\!(4)\!)\rangle$. After solving and reducing we get

$$\begin{aligned} & \mathit{data}\ \mathit{Either}^1 = \mathit{Left}_1^1\ \hat{4} \mid \mathit{Right}_1^1\ \hat{3}\\ & \textbf{let}\ f' = \lambda ex'.\textbf{case}^D\ ex'\ \textbf{of}\\ & \qquad\qquad \mathit{Left}_1^1\ x'\ \rightarrow\ x'\\ & \qquad\qquad \mathit{Right}_1^1\ x'\ \rightarrow\ \bullet\\ & \textbf{in}\ (f'@(\mathit{Left}_1^1\ \bullet), f'@(\mathit{Right}_1^1\ \bullet)) : (\hat{4}, \hat{4}) \end{aligned}$$

If constructor *Right* is applied to any other value, we get a similar specialization

$$
\begin{array}{ll}
\vdash_{\mathrm{P}} & \mathbf{let}^{D}\ f = \ldots \\
& \ \mathbf{in}\ \left(f\ @^{D}\ (\mathit{Left}^{D}\ 4^{S}), f\ @^{D}\ (\mathit{Right}^{D}\ 3^{S}), f\ @^{D}\ (\mathit{Right}^{D}\ 9^{S})\right)^{D} : \left(\mathit{Int}^{S}, \mathit{Int}^{S}, \mathit{Int}^{S}\right)^{D} \\
& \ \hookrightarrow\ \Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7. \\
& \ \mathbf{let}\ f' = \ldots \\
& \ \mathbf{in}\ \left(f'@(\mathit{Left}^{h_5}\ \bullet), f'@(\mathit{Right}^{h_6}\ \bullet), f'@(\mathit{Right}^{h_7}\ \bullet)\right)^{D} \\
& : \forall t_e, t_r.\ h_1 : \mathrm{IsInt}\ t_r, \\
& \qquad\qquad \vdots \\
& \qquad\qquad h_5 : \mathrm{HasPolyC}\ t_e\ \ \mathit{Left}\ \ \hat{4}, \\
& \qquad\qquad h_6 : \mathrm{HasPolyC}\ t_e\ \ \mathit{Right}\ \ \hat{3}, \\
& \qquad\qquad h_7 : \mathrm{HasPolyC}\ t_e\ \ \mathit{Right}\ \ \hat{9} \Rightarrow (t_r, t_r, t_r)
\end{array}
$$

Constraint solving proceeds the same way, but now to prove the HasMGBr predicate on *Right*, it must verify that both functions $\hat{3} \to \hat{4}$ and $\hat{9} \to \hat{4}$ are instances of $(\forall t, t'.\mathrm{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \to t')$, which is not true! So the expression cannot be solved at all.

The fact that all residual branches must have the same type can also lead to information flow between the arguments of different summands.

$$
\begin{array}{ll}
\vdash_{\mathrm{P}} & \lambda y.\mathbf{let}^{D}\ f = \ldots \\
& \ \mathbf{in}\ \left(f\ @^{D}\ (\mathit{Left}^{D}\ y), f\ @^{D}\ (\mathit{Right}^{D}\ 3^{S}), \mathbf{lift}\ y\right)^{D} : \mathit{Int}^{S} \to^{D} \left(\mathit{Int}^{S}, \mathit{Int}^{S}, \mathit{Int}^{D}\right)^{D} \\
& \ \hookrightarrow\ \Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9. \\
& \ \lambda y'.\mathbf{let}\ f' = \lambda ex'.\mathbf{polycase}_{v}\ ex'\ \mathbf{with}\ h_3\ \mathbf{and}\ (h_4, h_5)\ \mathbf{of} \\
& \qquad\qquad\qquad\qquad \mathit{Left}\ \to\ \Lambda h.\lambda x'.x' \\
& \qquad\qquad\qquad\qquad \mathit{Right}\ \to\ \Lambda h, h'.\lambda x'.\bullet \\
& \ \mathbf{in}\ \left(f'@(\mathit{Left}^{h_8}\ y'), f'@(\mathit{Right}^{h_9}\ \bullet), h_2\right)^{D} \\
& : \forall t_e, t_y, t_r.\ h_1 : \mathrm{IsInt}\ t_r, \\
& \qquad\qquad h_2 : \mathrm{IsInt}\ t_y, \\
& \qquad\qquad h_3 : \mathrm{IsPolySum}\ t_e, \\
& \qquad\qquad h_4 : \mathrm{HasMGBr}\ t_e\ \ \mathit{Left}\ \ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to t)\ \ t_r, \\
& \qquad\qquad h_5 : \mathrm{HasMGBr}\ t_e\ \ \mathit{Right}\ \ (\forall t, t'.\mathrm{IsInt}\ t, t' := t + \hat{1} \Rightarrow t \to t')\ \ t_r, \\
& \qquad\qquad h_6 : \mathrm{HasMGC}\ t_e\ \ \mathit{Left}\ \ (\forall t.\mathrm{IsInt}\ t \Rightarrow t), \\
& \qquad\qquad h_7 : \mathrm{HasMGC}\ t_e\ \ \mathit{Right}\ \ (\forall t.\mathrm{IsInt}\ t \Rightarrow t), \\
& \qquad\qquad h_8 : \mathrm{HasPolyC}\ t_e\ \ \mathit{Left}\ \ t_y, \\
& \qquad\qquad h_9 : \mathrm{HasPolyC}\ t_e\ \ \mathit{Right}\ \ \hat{3} \Rightarrow t_y \to (t_r, t_r, \mathit{Int})
\end{array}
$$

Constraint solving can detect that t_r must unify with $\hat{4}$ to prove the HasMGBr predicate on *Right*. Now to prove it for *Left*, the function type $t_y \to t_r$ must be an instance of $\forall t.\mathrm{IsInt}\ t \Rightarrow t \to t$, which leads to unifying t_y with $\hat{4}$ to get the function $\hat{4} \to \hat{4}$. The second predicate can now be proved and h_2 can be assigned value 4. After solving and reducing, we get

$$
\begin{array}{l}
\mathit{data}\ \mathit{Either}^{1} = \mathit{Left}_1^1\ \ \hat{4} \mid \mathit{Right}_1^1\ \ \hat{3} \\
\\
\lambda y'.\mathbf{let}\ f' = \lambda ex'.\mathbf{case}^{D}\ ex'\ \mathbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Left}_1^1\ \ x'\ \to\ x' \\
\qquad\qquad\qquad\qquad \mathit{Right}_1^1\ \ x'\ \to\ \bullet \\
\ \mathbf{in}\ (f'@(\mathit{Left}_1^1\ \ y'), f'@(\mathit{Right}_1^1\ \ \bullet), 4) : \hat{3} \to (\hat{4}, \hat{4}, \mathit{Int})
\end{array}
$$

$\diamond$

Example 4.19 As in regular dynamic sum types, conditional predicates appear when there is not enough information on the summands. They also have the effect of ignoring branches that cannot be reached.

The expression

$$
\begin{array}{l}
\textbf{polydata}\ \mathit{EitherDS}^D \ = \ \mathit{Left}^D\ \mathit{Int}^D \mid \mathit{Right}^D\ \mathit{Int}^S \\
\\
\vdash_{\mathrm{P}}\ \textbf{let}^D\ g = \lambda^D e.\textbf{case}^D\ e\ \textbf{of} \\
\qquad\qquad \mathit{Left}^D\ x\ \rightarrow\ \textbf{let}^D\ \mathit{id} = \lambda^D z.z\ \textbf{in} \\
\qquad\qquad\qquad\qquad \textbf{fst}^D\ \left(\textbf{lift}\ (\mathit{id}\ @^D\ 11^S), \textbf{lift}\ (\mathit{id}\ @^D\ 15^S)\right)^D \\
\qquad\qquad \mathit{Right}^D\ x\ \rightarrow\ \textbf{lift}\ x +^D 1^D \\
\quad \textbf{in}\ \left(g\ @^D\ (\mathit{Right}^D\ 4^S), g\ @^D\ (\mathit{Right}^D\ 2^S)\right)^D : \left(\mathit{Int}^D, \mathit{Int}^D\right)^D \\
\quad \hookrightarrow \\
\quad \Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7. \\
\quad \textbf{let}\ g' = \lambda e'.\textbf{polycase}_v\ e'\ \textbf{with}\ h_1\ \textbf{and}\ (h_2, h_3)\ \textbf{of} \\
\qquad \mathit{Left}\ \rightarrow\ \Lambda h'_1, h'_2, h'_3.\lambda x'.\textbf{let}\ \mathit{id}' = \lambda z'.z'\ \textbf{in}\ \textbf{fst}\ (h'_1, h'_1) \\
\qquad \mathit{Right}\ \rightarrow\ \Lambda h.\lambda x'.h + 1 \\
\quad \textbf{in}\ (g'@(\mathit{Right}^{h_6}\ \bullet), g'@(\mathit{Right}^{h_7}\ \bullet)) \\
\qquad : \forall t_e, t'.\ h_1 : \mathrm{IsPolySum}\ t_e, \\
\qquad\qquad h_2 : \mathit{Left} \in t_e\,?\,\mathrm{HasMGBr}\ t_e\ \ \mathit{Left}\ \ (\forall t.\ \mathrm{IsInt}\ t, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad t \sim \hat{1}1, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad t \sim \hat{1}5 \Rightarrow \mathit{Int} \rightarrow \mathit{Int})\ \ \mathit{Int}, \\
\qquad\qquad h_3 : \mathrm{HasMGBr}\ t_e\ \ \mathit{Right}\ \ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \rightarrow \mathit{Int})\ \ \mathit{Int}, \\
\qquad\qquad h_4 : \mathit{Left} \in t_e\,?\,\mathrm{HasMGC}\ t_e\ \ \mathit{Left}\ \ \mathit{Int}, \\
\qquad\qquad h_5 : \mathrm{HasMGC}\ t_e\ \ \mathit{Right}\ \ (\forall t.\mathrm{IsInt}\ t \Rightarrow t), \\
\qquad\qquad h_6 : \mathrm{HasPolyC}\ t_e\ \ \mathit{Right}\ \ \hat{4}, \\
\qquad\qquad h_7 : \mathrm{HasPolyC}\ t_e\ \ \mathit{Right}\ \ \hat{2} \Rightarrow (\mathit{Int}, \mathit{Int})
\end{array}
$$

Function id' has residual type $t \rightarrow t$, and evidence h'_1 proves t is a one-point integer type. Then the specialization of $\textbf{lift}\ (\mathit{id}\ @^D\ n)$ must be h'_1 for any n. But since id is applied to arguments of types $\hat{1}1$, and $\hat{1}5$, they should both unify with t'. This is expressed in the type scheme constraining the *Left* branch (the one with evidence h_2).

Notice that predicate HasMGBr could never be proved if *Left* had any arguments — the type scheme itself has a set of predicates that does not hold. However, since it cannot be derived that *Left* is actually part of the residual sum type, all predicates regarding this constructor appear guarded. Constraint solving, detecting it does not appear in the code, can leave it out of the data definition, and prove these predicates with evidence $\bullet$.

Now constructor *Right* does not have any guarded predicates, because the predicates HasPolyC make the condition true (see rule (HasPolyC-Guard)). Constraint solving forms a sum type with summands $\left\{\mathit{Right}^1_1, \mathit{Right}^1_2\right\}$, and proves the HasMGBr predicate with conversions $[](\!(4)\!)$ and $[](\!(2)\!)$.

$$
\begin{array}{l}
\textbf{let}\ f' = \lambda e'.\textbf{polycase}_v\ e'\ \textbf{with}\ \left\{\mathit{Right}^1_1, \mathit{Right}^1_2\right\}\ \textbf{and}\ (\bullet, \langle 1, ([](\!(4)\!), [](\!(2)\!))\rangle)\ \textbf{of} \\
\quad \mathit{Left}\ \rightarrow\ \Lambda h'_1, h'_2, h'_3.\lambda x.\textbf{let}\ \mathit{id}' = \lambda z'.z'\ \textbf{in}\ \textbf{fst}\ (h'_1, h'_1) \\
\quad \mathit{Right}\ \rightarrow\ \Lambda h.\lambda x.h + 1 \\
\textbf{in}\ (f'@(\mathit{Right}^1_1\ \bullet), f'@(\mathit{Right}^1_2\ \bullet)) : (\mathit{Int}, \mathit{Int})
\end{array}
$$

After reduction, we obtain

$$\begin{array}{l} \mathit{data}\ \mathit{EitherDS}^1 = \mathit{Right}_1^1\ \hat{4} \mid \mathit{Right}_2^1\ \hat{2} \\ \\ \textbf{let}\ id' = \lambda^D z'.z'\ \textbf{in} \\ \textbf{let}\ f' = \lambda e'.\textbf{case}\ e'\ \textbf{of} \\ \quad \mathit{Right}_1^1\ x\ \rightarrow\ 4+1 \\ \quad \mathit{Right}_2^1\ x\ \rightarrow\ 2+1 \\ \textbf{in}\ (f'@(\mathit{Right}_1^1\ \bullet), f'@(\mathit{Right}_2^1\ \bullet)) : (\mathit{Int}, \mathit{Int}) \end{array}$$

where the *Left* branch has been erased. ◇

The following properties show that P is well behaved with respect to systems RT and SR.

Theorem 4.20 *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$
then $\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} e' : \sigma$
where $\Gamma_{(\mathrm{RT})} = \{x_i' : \tau_i \mid i = 1, \ldots, n\}$
if $\Gamma = \{x_i : \tau_i \hookrightarrow x_i' : \tau_i \mid i = 1, \ldots, n\}$

Theorem 4.21 *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$
then $\Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow \sigma$

The following properties also hold.

Proposition 4.22 ⋆ *If* $\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma'$ *and* $\Delta' \Vdash \bar{v} : \Delta$
then $\Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'[\bar{v}/\bar{h}] : \sigma'$

Proposition 4.23 ⋆ *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$
then $S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\,\sigma$

Lemma 4.24 *If* $\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$
then $EV(e') \subseteq \bar{h}$

Lemma 4.25 *If* $\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$
then there exist $\bar{\beta}$, Δ_σ *and* τ'' *such that* $\sigma = \forall\bar{\beta}.\Delta_\sigma \Rightarrow \tau''$

4.4.3 A note on HasC, HasPolyC and upper bounds

The new rules on system P look quite similar to the ones for expressions involving regular dynamic sum types (section 2.5.3 and changes in section 3.2). In fact, in the rules for specializing tagged expressions — rules (DCONSTR) and (POLYCONSTR) — there is only one change in the residual sum type's description: where there used to be a predicate HasC, now there is a predicate HasPolyC. These two predicates in turn have very similar meanings, the only difference being that the first one states that a type includes a constructor with an *only* argument, whereas the second one states that it is one of *potentially many* arguments.

Predicate HasC has two additional counterparts for polyvariant sum types, used for specifying the source-residual relation and for specializing case expressions respectively: predicates HasMGC and HasMGBr.

Rules defining the SR relation for both kinds of sum types differ in just one predicate, as well — compare rules (SR-DATA) and (SR-POLYDATA). In both of them, for a source sum type τ to be SR-related to a residual type τ', it is necessary that the argument of each source constructor be related to the argument(s) of the corresponding constructor(s) in τ', if they exist. Suppose constructor j has a source argument τ_j. In the case of regular sum types, it suffices to say that if the constructor exists in τ''s definition, then the only argument it has is a type τ'_j such that $\vdash_{\text{SR}} \tau_j \hookrightarrow \tau'_j$. Predicate HasC $\tau'\ K_j\ \tau'_j$ is enough for expressing this. In the case of polyvariant sums, there is potentially more than one of these τ'_js, and all of them need to be in SR-relation with τ_j. Since we do not know a priori how many of them there are, we cannot use HasPolyC for stating explicitly what residual arguments the constructor has. However, we do know that for any such set of residual types, there should be a residual type scheme σ'_j more general than all of them and such that $\vdash_{\text{SR}} \tau_j \hookrightarrow \sigma'_j$. This is why predicate HasMGC is useful. In this sense, it expresses an *upper bound* to every possible argument of a residual constructor.

Rules specifying specialization of case expressions also have the same structure for both kinds of sum types. Rule (POLYCASE) uses predicate HasMGBr where rule (DCASE-2) uses predicate HasC in combination with the unification predicate. The parallel here is a little more subtle, and is related to branches being specialized as functions.

For regular dynamic sum types, each branch can be specialized at most once. For a given constructor K_j, the pattern matching variable is assumed to have a type τ'_j that must be the type of the corresponding argument in the sum type definition, if it exists. Predicate HasC is enough to express this. Separately, the body of the branch has a residual type τ''_j that must be the same as the type of the expression only if the K_j is a part of the sum definition. The guarded unification predicate ensures this last condition.

Now for polyvariant sum types, branches can be specialized more than once, but the exact number and specific versions cannot be decided a priori. What can be determined is that every possible branch should respect the form of a function from a summand argument to the residual type of the expression — a general type scheme σ_k such that every branch on L_k (seen as such a function) is an instance of σ_k. Predicate HasMGBr expresses this condition, acting as an *upper bound* to every possible branch on a certain constructor and with a certain residual argument. In rule (POLYCASE), it appears guarded, so no restriction is placed on the general branch if there cannot be any instances.

In conclusion, HasC has a triple purpose:

1. To specify the summands of a residual sum type;
2. To state that the arguments are *the same as* a certain type SR-related to the source argument (modulo unification).
3. To make sure (in combination with the unification predicate) the *only* branch a constructor can have in a case expression is generated by specialization of the source branch.

With polyvariant sums, these three purposes are fulfilled by different predicates:

1. HasPolyC specifies the summands of a residual sum type;
2. HasMGC states that the arguments are *instances* of a certain type scheme SR-related to the source argument.

3. HasMGBr makes sure that *all the branches* a constructor can have in a case expression are instances of a type scheme generated by specialization of the source branch.

Upper bounds are useful when multiple versions of a single source expression can be generated. They are a key concept for expressing polyvariance in Principal Type Specialization (see section 2.3.1), and a natural feature of polyvariant sums.

Chapter 5

Extending The Algorithm and The Proof

In chapter 4, we extended the principal type specialization system to handle polyvariant sum types, and proved that certain properties are kept after our addition. However, the preservation of the main property — that of principality — was not established there.

In this chapter, we extend the algorithm for computing principal type specializations to consider our new rules and constructs. We follow the lines of the constructive proof in Martínez López's formulation [2005, chapter 7]: the system has principal specializations because there is an *equivalent* algorithm that computes them, for a suitable definition of equivalence.

The algorithm we extend in this chapter is implemented in a simple prototype written in Haskell. In section 5.3, we discuss other aspects of it that are not directly related with the proof of principality.

5.1 A syntax-directed system, S

The rules in system P clearly specify how each construct in both term and type languages is specialized. However, they are not suitable for describing and algorithm, because they do not completely follow the structure of a term, that is, they are not *syntax-directed*. As an intermediate step to a specialization algorithm, a syntax-directed system — system S — is presented, with judgments of the form

$$\Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow \ e' : \tau'$$

The rules are adjusted so that for every term, there is at most one rule applicable. Also, only type expressions are involved in the process, handling qualified types and type schemes with a generalization operator — see section 2.4.

We extend system S with the rules corresponding to (POLYCONSTR) and (POLYCASE). They are presented in figure 5.1.

Rule (S-POLYCONSTR) is exactly the same as the original, since there are no type schemes involved. Rule (S-POLYCASE), however, is different, since the specialization of the branches must be dealt with. Each of them is specialized under system S, and the corresponding HasMGBr predicate is built with the generalization of the obtained results.

$$\text{(S-POLYCONSTR)}\quad \frac{\Delta \vdash_{\text{SR}} Y^D \;\hookrightarrow\; \tau'_e \qquad \Delta \mid \Gamma \vdash_{\text{S}} e : Y(L_j) \;\hookrightarrow\; e' : \tau'_j \qquad \Delta \Vdash v_j : \text{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j}{\Delta \mid \Gamma \vdash_{\text{S}} L_j^D\ e : Y^D \;\hookrightarrow\; L_j^{v_j}\ e' : \tau'_e}$$

$$\text{(S-POLYCASE)}\quad \frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\text{S}} e : Y^D \;\hookrightarrow\; e' : \tau'_e \\ \Delta \Vdash v^y : \text{IsPolySum}\ \tau'_e \\ \Delta \vdash_{\text{SR}} \tau \;\hookrightarrow\; \tau' \\ \left(\begin{array}{l} \bar{h}_k : \Delta_k \mid \Gamma \vdash_{\text{S}} \lambda^D x_k.e_k : Y(L_k) \rightarrow^D \tau \;\hookrightarrow\; e''_k : \tau'_k \\ \sigma_k = \text{Gen}_\Gamma(\Delta_k \Rightarrow \tau'_k), \qquad e'_k = \Lambda \bar{h}_k.e''_k \\ \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\text{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau' \end{array}\right)_{k\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\text{S}} \ \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^D\ \ x_k\ \ \rightarrow\ \ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ \ (w_k)_{k\in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ \ (L_k\ \ \rightarrow\ \ e'_k)_{k\in B} : \tau'\end{array}}$$

Figure 5.1: Syntax-directed specialization system for polyvariant sum types

The following properties are preserved by our addition. The first two show that the system is well behaved with respect to entailment and substitutions, whereas the last establish an equivalence between systems S and P.

Proposition 5.1 *If* $h : \Delta \mid \Gamma \vdash_{\text{S}} e : \tau \;\hookrightarrow\; e' : \tau'$ *then* $h : S\,\Delta \mid S\,\Gamma \vdash_{\text{S}} e : \tau \;\hookrightarrow\; e' : S\,\tau'$

Proposition 5.2 *If* $h : \Delta \mid \Gamma \vdash_{\text{S}} e : \tau \;\hookrightarrow\; e' : \tau'$ *and* $\Delta' \Vdash v : \Delta$ *then* $\Delta' \mid \Gamma \vdash_{\text{S}} e : \tau \;\hookrightarrow\; e'[h/v] : \tau'$

Theorem 5.3 ⋆ *If* $\Delta \mid \Gamma \vdash_{\text{S}} e : \tau \;\hookrightarrow\; e' : \tau'$ *then* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \;\hookrightarrow\; e' : \tau'$

Theorem 5.4 ⋆ *If* $\bar{h} : \Delta \mid \Gamma \vdash_{\text{P}} e : \tau \;\hookrightarrow\; e' : \sigma$ *then there exist* $\bar{h}'_s$, Δ'_s, e'_s, τ'_s *and* C'_s *such that* $\bar{h}'_s : \Delta'_s \mid \Gamma \vdash_{\text{S}} e : \tau \;\hookrightarrow\; e'_s : \tau'_s$, $C'_s : \text{Gen}_\Gamma(\Delta'_s \Rightarrow \tau'_s) \geq (\bar{h} : \Delta \mid \sigma)$, $C'_s[\Lambda \bar{h}'_s.e'_s] = e'$

Theorem 5.3 establishes the soundness of the syntax-directed system with respect to the original specialization rules. Theorem 5.4 establishes a form of completeness property showing that every specialization can be described by a syntax-directed derivation. This cannot be done the obvious way — for instance, if $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \;\hookrightarrow\; e' : \sigma$, then it will not in general be possible to derive the same typing in system S, since σ is a type scheme, not necessarily a simple type. However, for any such derivation, theorem 5.4 guarantees the existence of an S-derivation yielding $e'_s : \tau'_s$ under predicate context $\bar{h}'_s : \Delta'_s$, such that the inferred type scheme $\text{Gen}_\Gamma(\Delta'_s \Rightarrow \tau'_s)$ is more general than the constrained type scheme $(\Delta \mid \sigma)$ determined by the original derivation, and that $\Lambda \bar{h}'_s.e'_s$ can be converted to e'.

$$\frac{\tau \sim^{U} \tau'}{\text{IsPolySum } \tau \sim^{U} \text{IsPolySum } \tau'}$$

$$\frac{\tau \sim^{U_1} \tau' \quad U_1\,\sigma \sim^{U_2} U_1\,\sigma'}{\text{HasMGC } \tau\ \ L_k\ \ \sigma \sim^{U_2 U_1} \text{HasMGC } \tau'\ \ L_k\ \ \sigma'}$$

$$\frac{\tau \sim^{U_1} \tau' \quad U_2\,\sigma \sim^{U_1} U_2\,\sigma' \quad U_2\,U_1\,\tau_r \sim^{U_3} U_2\,U_1\,\tau_r'}{\text{HasMGBr } \tau\ \ L_k\ \ \sigma\ \ \tau_r \sim^{U_3 U_2 U_1} \text{HasMGBr } \tau'\ \ L_k\ \ \sigma'\ \ \tau_r'}$$

$$\frac{\tau \sim^{U_1} \tau' \quad U_1\,\tau_r \sim^{U_2} U_1\,\tau_r'}{\text{HasPolyC } \tau\ \ L_k\ \ \tau_r \sim^{U_2 U_1} \text{HasPolyC } \tau'\ \ L_k\ \ \tau_r'}$$

Figure 5.2: Rules for unification of predicates involving polyvariant sums

5.2 The Principal Type Specialization Algorithm

We are now able to extend the algorithm presented by Martínez López [2005] and extended by Russo [2004], that computes the principal type specialization for a given typed source term, and express a notion of equivalence to system P via system S.

We use a number of subsystems corresponding to the algorithmic versions of the different systems used in $\vdash$ and $\vdash_{\mathrm{S}}$. They are presented next.

5.2.1 A unification algorithm

The unification algorithm is specified with judgments of the form $\sigma_1 \sim^{U} \sigma_2$, where σ_1 and σ_2 are input and U is output, a most general unifier for them.

We only need to extend it with unification rules for the new predicates. They are presented in figure 5.2.

The following properties establish that the output, if it exists, is indeed a most general unifier.

Proposition 5.5 *If $\sigma \sim^{U} \sigma'$ then $U\,\sigma = U\,\sigma'$*

Proposition 5.6 *If $S\,\sigma = S\,\sigma'$ then $\sigma \sim^{U} \sigma'$ and there exists a substitution T such that $S = TU$*

5.2.2 An entailment algorithm

The entailment algorithm expects a predicate δ as input and an assumed predicate context Δ. It computes the set of predicates Δ' that should be added to Δ to entail δ, and the evidence for this entailment. Judgments are of the form

$$\Delta' \mid \Delta \Vdash_{\mathrm{W}} v : \delta$$

The only rule for this algorithm is the following:

$$h : \delta \mid \Delta \Vdash_{\mathrm{W}} h : \delta$$

$$\text{(WSR-POLYDATA)}\quad \frac{\left(\begin{array}{l}\Delta_k \vdash_{\text{W-SR}} Y(L_k) \hookrightarrow \tau'_k \\ \sigma_k = \text{Gen}_{\emptyset,\emptyset}(\Delta_k \Rightarrow \tau'_k)\end{array}\right)_{L_k \in Y}}{\begin{array}{l}\text{IsPolySum } t, \\ (L_k \in t \,?\, \text{HasMGC } t\ \ L_k\ \ \sigma_k)_{L_k \in Y}\end{array} \quad \vdash_{\text{W-SR}} Y^D \hookrightarrow t}\ (t \text{ fresh})$$

Figure 5.3: Algorithm for the source-residual relation for polyvariant sum types

That is, Δ' is a singleton set assuming evidence for δ.

This formulation allows for more refined rules proving each particular predicate according to its meaning and entailment rules. This is currently handled by simplification; see section 5.3.

The following proposition can be easily proved.

Proposition 5.7 *If* $\Delta' \mid \Delta \Vdash_{\text{W}} \delta$ *then* $\Delta', \Delta \Vdash \delta$

5.2.3 An algorithm for the SR relation

The source-residual relation algorithm, W-SR, expects a source type as input. It calculates the most general residual type in the sense given in proposition 5.9, and a predicate assignment expressing the constraints on the type variables. Judgments in system W-SR are of the form $\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'$, where τ is input and both Δ and τ' are output.

We extend this system to calculate the SR relation for polyvariant sum types. Only one rule is needed; it is presented in figure 5.3.

We extend the propositions relating system W-SR with SR. The first one proves that the former is sound with respect to the latter. The second one shows that W-SR yields the most general type and predicate assignment in SR-relation with the input.

Proposition 5.8 *If* $\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'$ *then* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'$

Proposition 5.9 *If* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$
then there exist Δ'_w, τ'_w *and* C'_w *such that*
$\Delta'_w \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'_w$ *with all the residual variables fresh and* $C'_w : \text{Gen}_{\emptyset,\emptyset}(\Delta'_w \Rightarrow \tau'_w) \geq (\Delta \mid \sigma)$

5.2.4 An algorithm for principal type specialization, W

The principal type specialization algorithm takes an assignment Γ and a typed source term $e : \tau$. It returns a specialized expression $e' : \tau'$, a predicate assignment Δ and a substitution S that must be applied to Γ to adjust the types appearing in it. Judgments are of the form

$$\Delta \mid S\,\Gamma \vdash_{\text{W}} e : \tau \hookrightarrow e' : \tau'$$

with the same meaning as in systems P and S. For each syntax-directed rule in system S, there is a counterpart in system W.

Figure 5.4 shows our extension to the algorithm to consider rules (POLYCONSTR) and (POLYCASE). They are essentially the same as rules in system S, only taking substitutions into account.

Our extension preserves the following property:

$$\text{(W-POLYCONSTR)}\quad \frac{\begin{array}{l}\Delta \vdash_{\text{W-SR}} Y^D \;\hookrightarrow\; \tau'_e \\ \Delta' \mid S\,\Gamma \vdash_{\text{W}} e : Y(L_j) \;\hookrightarrow\; e' : \tau'_j \\ \Delta'' \mid \Delta, \Delta' \Vdash_{\text{W}} v_j : \text{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j\end{array}}{\Delta, \Delta', \Delta'' \mid S\,\Gamma \vdash_{\text{W}} L_j^D\ e : Y^D \;\hookrightarrow\; L_j^{v_j}\ e' : \tau'_e}$$

$$\text{(W-POLYCASE)}\quad \frac{\begin{array}{l}\Delta_e \mid S_e\,\Gamma \vdash_{\text{W}} e : Y^D \;\hookrightarrow\; e' : \tau'_e \\ \Delta_p \mid \Delta_e \Vdash_{\text{W}} v^y : \text{IsPolySum}\ \tau'_e \\ \Delta_{SR} \vdash_{\text{W-SR}} \tau \;\hookrightarrow\; \tau' \\ \big(h_k : \Delta_k \mid S_k\,S^*_{k-1}\,\Gamma \vdash_{\text{W}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \;\hookrightarrow\; e''_k : \tau'_k\big)_{k\in B} \\ \big(\sigma_k = \text{Gen}_{S^*_n\,\Gamma}(S^n_{k+1}\,(\Delta_k \Rightarrow \tau'_k)), \qquad e'_k = \Lambda h_k.e''_k\big)_{k\in B} \\ \left(\begin{array}{l}\Delta'_k \mid \Delta'_{k-1}, \ldots, \Delta'_1, \Delta^*_e, \Delta^*_p, \Delta_{SR} \Vdash_{\text{W}} \\ \qquad\qquad w_k : L_k \in (S^n_1\,\tau'_e)\,?\,\text{HasMGBr}\ (S^n_1\,\tau'_e)\ \ L_k\ \ \sigma_k\ \ \tau'\end{array}\right)_{k\in B}\end{array}}{\begin{array}{l}\Delta'_n, \ldots, \Delta'_1, \Delta^*_e, \Delta^*_p, \Delta_{SR} \mid S^*_n\,\Gamma \vdash_{\text{W}} \mathbf{case}^D\ e\ \mathbf{of}\ (L_k^D\ \ x_k\ \ \to\ \ e_k)_{k\in B} : \tau \\ \qquad\qquad\qquad\qquad \hookrightarrow \\ \qquad\qquad\qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad\qquad\qquad\qquad \mathbf{of}\ (L_k\ \to\ e'_k)_{k\in B} : \tau'\end{array}}$$

where

$$\begin{array}{lcl} S^j_i & = & (S_j \circ (\ldots \circ S_i)) \\ S^*_j & = & (S^j_1 \circ S_e) \\ \Delta^*_e & = & S^n_1\,\Delta_e \\ \Delta^*_p & = & S^n_1\,\Delta_p \\ n & = & |B| \end{array}$$

Figure 5.4: Principal type specialization algorithm for polyvariant sum types

Lemma 5.10 *If* $\bar{h} : \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \hookrightarrow e' : \tau'$ *then* $EV(e') \subseteq \bar{h}$

Many of the rules in the system introduce fresh variables (see appendix A.2.1 and theorem 5.9), that is, variables which do not appear in the hypotheses of the rule nor in any other distinct branches of the complete derivation. Note that it is always possible to choose type variables in this way because the set of type variables is assumed to be countably infinite. In the presence of fresh variables, it is convenient to work with a weaker form of equality on substitutions, defined in the same way as in the theory of qualified types [Jones, 1994]. Two substitutions R and S are *similar* (written $R \approx S$), if they only differ in a finite number of fresh variables. In most cases, we can treat $R \approx S$ as $R = S$, since the only differences between the substitutions occur at variables which are not used elsewhere in the algorithm.

The results obtained by system W are equivalent to those obtained by system S, in the sense established in the following theorems. The first one establishes soundness of system W with respect to system S.

Theorem 5.11 ⋆ *If* $\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \hookrightarrow e' : \tau'$
then $\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau'$

The completeness result expresses that every residual term and type obtained by the syntax-directed system can be expressed as a particular case (with respect to substitutions) of the residual term and type produced by the algorithm.

Theorem 5.12 ⋆ *If* $\bar{h} : \Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau'$
then $\bar{h}'_w : \Delta'_w \mid T'_w\,\Gamma \vdash_{\mathrm{W}} e : \tau \hookrightarrow e'_w : \tau'_w$
and there exist a substitution R *and evidence* $\bar{v}'_w$ *such that*
$S \approx R T'_w$
$\tau' = R\,\tau'_w$
$\bar{h} : \Delta \Vdash \bar{v}'_w : R\,\Delta'_w$
$e' = e'_w[\bar{v}'_w/\bar{h}'_w]$

5.3 Constraint Solving

The algorithm presented in the previous section computes the principal type specialization of a given term. Because of its local quality, it produces potentially more predicates than needed, some of them being redundant or expressible in simpler forms. Moreover, when type or scheme variables have more than one possible value to assign, the decision is not made — only predicates constraining the variables are gathered to describe the possibilities. This is not enough for obtaining useful residual expressions!

Pablo E. Martínez López and Hernán Badenes address this issue [Martínez López and Badenes, 2003; Martínez López, 2005] with two related mechanisms:

Simplification is a variation of the process of *simplification and improvement* introduced by Mark Jones [1994; 1995]. The goal is to eliminate the redundant predicates and to decide the value of type and scheme variables with a unique solution.

Constraint solving is the process of assigning a value — even among several possibilities — to a variable whose context information has been gathered. By the introduction of this mechanism, a complete specialization consists of two clearly separated parts:

- the *specification* part, where a description of the problem is built in the form of constrained type schemes and residual terms with evidence expressions
- an *implementation* part, where a solution for the problem is found, and evidence expressions are replaced with the associated values

Following Aiken [1999], this approach treats program specialization as a static program analysis, where after analyzing each part of the program locally, different resolution techniques can be applied for solving the generated constraints.

Martínez López and Badenes present a thorough formalization of the simplification and constraint solving processes, they establish their structural rules and specify them for the predicates presented in section 2.2.3, namely IsInt, IsSum and IsMG. An extension is needed for handling the predicates involving polyvariant sum types, which is informally described next. A more precise specification is left for future work.

5.3.1 Our extension to constraint solving

When specializing expressions involving polyvariant sum types, the algorithm defined by system W yields a residual type involving at least a residual sum type t, where t is a type variable constrained with predicates such as IsPolySum, HasMGC, HasMGBr or HasPolyC. The residual expression may include variables representing the evidence for these predicates — we have shown several examples of this in chapter 4.

To obtain a useful result, constraint solving must take place. More specifically, the type variable t must be assigned a residual datatype Y^n with a fully defined set of constructors and residual arguments. We have extended the constraint solving mechanism with an heuristic to obtain such definition. Starting with a predicate h : IsPolySum t, all predicates involving t (without syntactical duplicates) are gathered and solved as follows:

Step 1. Get an index n for Y that has not been used so far.

Step 2. For each distinct predicate HasPolyC $t\ \ L_k\ \ \tau$, assign an index i that has not yet been used for L_k and include the summand $L^n_{k,i}\ \tau$ in the definition of Y^n. Prove the predicate with evidence $\overset{n}{i}$.

Call I the set of indexes k such that there is at least one predicate HasPolyC $t\ \ L_k\ \ \tau$. For each k, define $I'_k = \{1, \ldots, n_k\}$ where n_k is the number of predicates HasPolyC $t\ \ L_k\ \ \tau$. Then we have defined

$$data\ Y^n = \left\{L_{k,i}\ \tau'_{k,i}\right\}_{k\in I, i\in I'_k}$$

Assign value $\{L_{k,i}\}_{k\in I, i\in I'_k}$ to h, the evidence for IsPolySum t.

Step 3. Prove each guarded predicate $L_k \in t\,?\,\delta$ by simplifying it to δ if $k \in I$, and with evidence $\bullet$ if $k \notin I$.

Step 4. Verify each predicate HasMGC $t\ \ L_k\ \ \sigma$ — by the previous step we know $k \in I$ — by proving IsMG $\sigma\ \tau'_{k,i}$ for each $i \in I'_k$. Fail if any of them cannot be proved.

Step 5. Prove each predicate HasMGBr $t\ \ L_k\ \ \sigma\ \ \tau$ — as before we know $k \in I$ — with evidence $\left\langle n, (v_i)_{i\in I'_k} \right\rangle$, where v_i : IsMG $\sigma\ (\tau'_{k,i} \to \tau)$. Fail if any of them cannot be proved.

Example 5.13 Taking the following expression as input

$$\begin{array}{l}\textbf{polydata}\ \mathit{PolySDF}^D\ =\ \mathit{Sta}^D\ \mathit{Int}^S\ \mid\ \mathit{Dyn}^D\ \mathit{Int}^D\ \mid\ \mathit{Fun}^D\ (\mathit{Int}^S \to^D \mathit{Int}^D)\\[2ex]
\textbf{let}^D\ \mathit{id} = \lambda^D z.z\\
\quad \textbf{in case}^D\ \pi^D_{1,3}\left(\mathit{id}\ @^D\ (\mathit{Sta}^D\ 11^S), \mathit{id}\ @^D\ (\mathit{Dyn}^D\ 5^D), \mathit{id}\ @^D\ (\mathit{Sta}^D\ 24^S)\right)^D\ \textbf{of}\\
\qquad \mathit{Sta}^D\ x\ \to\ \textbf{lift}\ x\\
\qquad \mathit{Dyn}^D\ y\ \to\ y\\
\qquad \mathit{Fun}^D\ f\ \to\ f\ @^D\ 3^S\\
: \mathit{Int}^D\end{array}$$

the W algorithm, after simplification, yields the following specialization, where the duplicate predicates have been removed.

$$\begin{array}{l}\Lambda h_1, \ldots, h_{10}.\\
\textbf{let}\ \mathit{id}' = \lambda z'.z'\ \textbf{in}\\
\quad \textbf{polycase}_v\ \pi_{1,3}\,(\mathit{id}'@(\mathit{Sta}^{h_4}\bullet), \mathit{id}'@(\mathit{Dyn}^{h_5}\,5), \mathit{id}'@(\mathit{Sta}^{h_8}\bullet))\ \textbf{with}\ h_1\ \textbf{and}\\
\qquad (h_3, h_7, h_{10})\ \textbf{of}\\
\qquad\quad \mathit{Sta}\ \to\ \Lambda h_x.\lambda x'.h_x\\
\qquad\quad \mathit{Dyn}\ \to\ \lambda y'.y'\\
\qquad\quad \mathit{Fun}\ \to\ \lambda f'.f'@\bullet\\
: \forall t.\ h_1 : \text{IsPolySum}\ t,\\
\qquad h_2 : \text{HasMGC}\ t\ \ \mathit{Sta}\ \ (\forall t_x.\text{IsInt}\ t_x \Rightarrow t_x),\\
\qquad h_3 : \text{HasMGBr}\ t\ \ \mathit{Sta}\ \ (\forall t_x.h_x : \text{IsInt}\ t_x \Rightarrow t_x \to \mathit{Int})\ \ \mathit{Int},\\
\qquad h_4 : \text{HasPolyC}\ t\ \ \mathit{Sta}\ \ \hat{1}1,\\
\qquad h_5 : \text{HasPolyC}\ t\ \ \mathit{Sta}\ \ \hat{2}4,\\
\qquad h_6 : \text{HasMGC}\ t\ \ \mathit{Dyn}\ \ \mathit{Int},\\
\qquad h_7 : \text{HasMGBr}\ t\ \ \mathit{Dyn}\ \ (\mathit{Int} \to \mathit{Int})\ \ \mathit{Int},\\
\qquad h_8 : \text{HasPolyC}\ t\ \ \mathit{Dyn}\ \ \mathit{Int},\\
\qquad h_9 : \mathit{Fun} \in t\,?\,\text{HasMGC}\ t\ \ \mathit{Fun}\ \ (\forall t_f.\text{IsInt}\ t_f \Rightarrow t_f \to \mathit{Int}),\\
\qquad h_{10} : \mathit{Fun} \in t\,?\,\text{HasMGBr}\ t\ \ \mathit{Fun}\ \ ((\hat{3} \to \mathit{Int}) \to \mathit{Int})\ \ \mathit{Int} \Rightarrow \mathit{Int}\end{array}$$

Now since t does not appear in the residual type, it can be solved. The constraint solving algorithm gathers all predicates involving this variable (all of them in this example) and proceeds as follows

Step 1. Get index 1 for *PolySDF*.

Step 2. a) For constructor Sta^1, include summands $\mathit{Sta}^1_1\ \ \hat{1}1$ and $\mathit{Sta}^1_2\ \ \hat{2}4$. Assign evidence $\begin{smallmatrix}1\\1\end{smallmatrix}$ to h_4 and $\begin{smallmatrix}1\\2\end{smallmatrix}$ to h_5.

b) For constructor Dyn^1, include the summand $\mathit{Dyn}^1_1\ \ \mathit{Int}$. Assign evidence $\begin{smallmatrix}1\\1\end{smallmatrix}$ to h_8.

c) Since there is no HasPolyC t *Fun* τ among the predicates, there are no summands for this constructor.

Assign the set $\{\mathit{Sta}^1_1, \mathit{Sta}^1_2, \mathit{Dyn}^1_1\}$ to h_1.

Step 3. Only two guarded predicates remain, both on constructor *Fun*, that does not appear in our constructor set. Assign $\bullet$ to h_9 and h_{10}.

Step 4. a) Calling $\sigma_{Sta} = \forall t_x.\text{IsInt } t_x \Rightarrow t_x$, predicate HasMGC t Sta σ_{Sta} is verified since both IsMG σ_{Sta} $\hat{1}1$ and IsMG σ_{Sta} $\hat{2}4$ can be proved.

b) Predicate HasMGC t Dyn Int is verified by proving IsMG Int Int.

Step 5. a) Calling $\sigma'_{Sta} = \forall t_x.\text{IsInt } t_x \Rightarrow t_x \to Int$, predicate HasMGBr t Sta σ'_{Sta} Int is verified with $[]((11))$: IsMG σ'_{Sta} $(\hat{1}1 \to Int)$ and $[]((24))$: IsMG σ'_{Sta} $(\hat{2}4 \to Int)$. Evidence h_3 is assigned value $\langle 1, ([]((11)), []((24)))\rangle$.

b) Predicate HasMGBr t Dyn $(Int \to Int)$ Int is verified with $[]$: IsMG $(Int \to Int)$ $(Int \to Int)$. Evidence h_7 is assigned value $\langle 1, []\rangle$.

After solving variable t successfully, the residual expression is

$$
\begin{array}{l}
data\ PolySDF^1 = Sta_1^1\ \ \hat{1}1 \mid Sta_2^1\ \ \hat{2}4 \mid Dyn_1^1\ \ Int \\
\\
\textbf{let } id' = \lambda z'.z' \textbf{ in} \\
\quad \textbf{polycase}_v\ \pi_{1,3}\,(id'@(Sta_1^1\ \ \bullet), id'@(Dyn_1^1\ \ 5), id'@(Sta_2^1\ \ \bullet))\ \textbf{with} \\
\quad \left\{Sta_1^1, Sta_2^1, Dyn_1^1\right\}\ \textbf{and} \\
\quad\quad (\langle 1, ([]((11)), []((24)))\rangle, \langle 1, []\rangle, \bullet)\ \textbf{of} \\
\quad\quad\quad Sta\ \to\ \Lambda h_x.\lambda x'.h_x \\
\quad\quad\quad Dyn\ \to\ \lambda y'.y' \\
\quad\quad\quad Fun\ \to\ \lambda f'.f'@\bullet \\
: Int
\end{array}
$$

which can be reduced to

$$
\begin{array}{l}
\textbf{let } id' = \lambda z'.z' \\
\quad \textbf{in case } \pi_{1,3}\,(id'@(Sta_1^1\ \ \bullet), id'@(Dyn_1^1\ \ 5), id'@(Sta_2^1\ \ \bullet))\ \textbf{of} \\
\quad\quad Sta_1^1\ \ x'\ \to\ 11 \\
\quad\quad Sta_2^1\ \ x'\ \to\ 24 \\
\quad\quad Dyn_1^1\ \ y'\ \to\ y' \\
: Int
\end{array}
$$

Notice how every aspect of polyvariant sum type specialization has been achieved by constraint solving: generation of multiple data declarations, replication of constructors with their corresponding branches, and elimination of dead ones. ◇

5.3.2 Discussion

There is a fundamental difference between the processes of simplification and constraint solving. Although they are both related to instantiating variables and proving predicates, they differ in the kind of variables about which they can make decisions. Whereas simplification assigns *unique* values deduced by entailment, constraint solving chooses values that are not a direct consequence of the context.

When solving a type variable t representing a polyvariant residual sum type, it is reasonable to wonder if it is in fact a solving, that is, if the value is decided among several possibilities. Indeed, the predicates constraining t might include:

- *Upper bounds* on the types of the summands, by predicates HasMGC and HasMGBr.

- *Requirements* for the inclusion of a particular summand, by predicate HasPolyC.

Clearly these do not uniquely determine a residual sum type! The algorithm we have described builds the data definition that includes all the summands required and *only* them, provided they respect the upper bounds. Other solutions are possible — they would include more summands, with different type arguments or even different constructors. Our algorithm chooses the minimum definition (with respect to inclusion) that satisfies all the predicates.

Another issue to consider is the potential generation of duplicate summands. Constraint solving can proceed on a variable as soon as it can be established that no more context information can affect it — namely, when it does not appear in the residual type itself but only in the predicates. This means that a sum type variable can be solved even when one of its arguments — appearing in a HasPolyC predicate — is also a type variable and has not yet been solved[1].

Example 5.14 The following specialization is the result of the W algorithm and the simplification process.

$$\textbf{polydata } Poly^D \;=\; Poly^D\ Int^S$$

$$\begin{array}{ll}
\vdash_{\mathrm{P}} & \textbf{let}^D\ id = \lambda^D z.z \\
& \quad \textbf{in } \lambda^D x.\textbf{case}^D\ \textbf{fst}^D\ \left(id\ @^D\ (Poly^D\ 11^S), id\ @^D\ (Poly^D\ x)\right)^D\ \textbf{of} \\
& \qquad\qquad Poly^D\ y\ \rightarrow\ \textbf{lift}\ y : Int^S \rightarrow^D Int^D \\
& \hookrightarrow \\
& \Lambda h_1, \ldots, h_6. \\
& \textbf{let } id' = \lambda z'.z' \\
& \quad \textbf{in } \lambda x'.\textbf{polycase}_v\ \textbf{fst}\,(id'@(Poly^{h_4}\ \bullet), id'@(Poly^{h_5}\ x'))\ \textbf{with}\ h_1\ \textbf{and}\ (h_3)\ \textbf{of} \\
& \qquad\qquad Poly\ \rightarrow\ \Lambda h_y.\lambda y'.h_y \\
& : \forall t_x, t_e.\ h_1 : \text{IsInt}\ t_x h_2 : \text{IsPolySum}\ t_e, \\
& \qquad\quad h_3 : \text{HasMGC}\ t_e\ \ Poly\ \ (\forall t.\text{IsInt}\ t \Rightarrow t), \\
& \qquad\quad h_4 : \text{HasMGBr}\ t_e\ \ Poly\ \ (\forall t_y.h_y : \text{IsInt}\ t_y \Rightarrow t_y \rightarrow Int)\ \ Int, \\
& \qquad\quad h_5 : \text{HasPolyC}\ t_e\ \ Poly\ \ \hat{11}, \\
& \qquad\quad h_6 : \text{HasPolyC}\ t_e\ \ Poly\ \ t_x \Rightarrow t_x \rightarrow Int
\end{array}$$

Simplification erases only syntactical duplicates, so two HasPolyC predicates remain. Constraint solving can take place on variable t_e, resulting in

$$\begin{array}{l}
data\ Poly^1 = Poly^1_1\ \ \hat{11} \mid Poly^1_2\ \ t_x \\
\\
\Lambda h_1.\textbf{let } id' = \lambda^D z'.z' \\
\quad \textbf{in } \lambda x'.\textbf{case fst}\,(id'@(Poly^1_1\ \ \bullet), id'@(Poly^1_2\ \ x'))\ \textbf{of} \\
\qquad\qquad Poly^1_1\ \ y'\ \rightarrow\ 11 \\
\qquad\qquad Poly^1_2\ \ y'\ \rightarrow\ h_1 \\
: \forall t_x.h_1 : \text{IsInt}\ t_x \Rightarrow t_x \rightarrow Int
\end{array}$$

where t_x generates a separate summand from $\hat{11}$. The problem would arise if t_x should be instantiated to $\hat{11}$ as well, — for example, if the function was applied to 11^S. In a context

[1] We consider that residual data declarations are inside the scope of the $\forall$ binder, even though they are displayed separate from the type of the expression. This needs further formalization; probably the grammar presented in definitions 4.3 and 4.5 needs to be changed. See discussion in section 6.2.1.

where constraint solving had to take place before the function application, there would be two constructors with the same residual argument. ◇

This issue is analogous to one cited by Martínez López when discussing constraint solving for polyvariant functions [2005, section 8.3.3]. One possibility to avoid generating duplicate arguments is to defer constraint solving of a sum type variable t until all the other variables in all the predicates where t appears are instantiated. However, as noted by Martínez López, this option contradicts our purpose of modularity. Another is to formalize a notion of *extensible sum type* so that a sum type with as few summands as possible could be defined, and extra summands added when they are needed.

It is important to note that this is a choice involving constraint solving exclusively — the type specialization system itself is not affected. This is one of the benefits of the principal type specialization approach, where the constraint generation and constraint solving aspects are separated. For a single well-defined specification, different implementations can be considered, and different heuristics can be tried.

Chapter 6

Conclusion

6.1 Related Work

6.1.1 Constructor specialization

Dealing with types in partial evaluation has been identified as a challenging problem since it first started being studied [Jones, 1988]. One of the open issues was if partial evaluation could be used to generate new specialized data types, just as it could generate new specialized functions. Indeed, new types were first generated only as simpler versions of the types in the source program [Launchbury, 1991], and later this limit was overcome by means of *constructor specialization*.

Constructor specialization was introduced by Torben Æ. Mogensen [1993] as a new mechanism for partial evaluation that lead to better results where the traditional methods failed to produce satisfactory specializations. Constructors are specialized with respect to the static part of their arguments, getting multiple residual versions of a single constructor just as partial evaluation can generate multiple copies of a single source function. Dynamic case expressions specialize to versions with as many branches as constructors are generated in the residual code, which in turn depend on the specialization of static arguments. Mogensen proposes several ways to achieve this:

- Regeneration of case expressions each time a new specialization is needed for a constructor.
- *Backpatching* earlier generated case expressions by destructive updating of the residual program.
- A multi-pass algorithm that first generates pieces of code for the case expression branches and then assembles these in the final residual program.

Our approach for specializing polyvariant sum types resembles this last option, where the first pass corresponds to specializing to a $\mathbf{polycase}_v$ construct and the second one to solving and reducing to the final case expression.

In his original presentation, Mogensen's discusses two limits of constructor specialization that are independent of how it is implemented. The first one is the fact that the residual program can have no more data type declarations than the source program, something that is against the general guiding rule of not letting the specialized result be limited by the structure

of the input. The second one, related to the former, involves the impossibility to separate the specialized constructors into different residual types. This means that every specialized constructor appears in every residual case expression that uses the original version, which can cause either run time errors or a lot of dead code in the residual program. This was later solved by enhancing binding time analysis [Dussart *et al.*, 1995].

None of the limits stated by Mogensen are an issue in our work. Multiple data type generation, separation of independent specializations, and dead code elimination were achieved for type specialization of regular dynamic sum types [Russo, 2004], and specialization of polyvariant sum types — the equivalent to constructor specialization itself — keeps both features.

6.1.2 John Hughes's polyvariant sums

The original formulation of type specialization [Hughes, 1996b; Hughes, 1996a; Hughes, 1998] includes a form of polyvariant sums. In Hughes's presentation, sum types are anonymous so they cannot be declared to be either monovariant or polyvariant. Instead, the special name *In* is used for identifying polyvariant constructors.

Anonymous sum types specialize to anonymous residual types, containing as many summands as necessary. Branch elimination and generation of independent residual data declarations are not an issue, since it is simply assumed that the residual type has exactly the required specialized constructors.

As happens with other features in this formulation, the specialization rules are not adequate for principal type specialization, and difficult to relate directly to an algorithm. A type specializer is implemented and described [Hughes, 1996b] where the residual types are represented by the notion of *open sets*: sets that hold the currently known summands and can be merged if it is found that two of them must unify.

Just as in principal type specialization, information constraining the types might depend on the context of use. In our formulation, we deal with this issue explicitly, using predicates, evidence expressions and the processes of constraint solving and residual term reduction. Hughes does not handle these problems in the specification of the system, but describes how they are dealt with in the implementation of his type specializer, with different mechanisms such as open sets, optimistic unification, and backtracking.

6.2 Future Work

There are several aspects of principal type specialization that can be improved, extended or strengthened. We begin by describing those directly related to our work, and then we briefly outline those for principal type specialization in general.

6.2.1 Work on polyvariant sum types

Improving constraint solving The meaning of the predicates we have introduced to describe polyvariant sums has been explained, and they all have a clear role in the specialization rules. However, during constraint solving, predicate HasMGC only needs to be checked for consistency — it does not contribute with evidence or any new information to the solving.

It is not clear if this check is in fact necessary or redundant. Given the way the predicates and their arguments are generated, it is possible that predicates of this form can never fail for

specializations resulting from a well-typed source expression. If this was the case, the need to verify these predicates could be relaxed, thus reducing the amount of work during constraint solving.

Formalizing constraint solving As observed by Martínez López [2005], constraint solving is the most involved part in our approach to principal type specialization — it is where the actual calculation of static data and information flow takes place. It is also without doubt the part that needs most of our efforts.

A first step to study it further would be to extend the current framework [Martínez López and Badenes, 2003], so that it includes the specification of the constraint solving rules we have described informally. This applies not only to polyvariant sums but also to other extensions to the basic formulation — static and dynamic sum-types, static functions and recursion, failure, etc.

Binding type variables in data declarations In both Russo's extension and ours, when solving a type variable representing a residual sum type, a data declaration is generated. The definition consists of a name and a set of summands: constructor names with an argument. Now there is no reason why these arguments should be solved before generating the data declaration — they could be type variables themselves.

Our current formulation presents data declarations as part of a residual term (see definitions 2.17 and 4.3):

$$
\begin{array}{lll}
e' & ::= & [\ ddcl'\ ]^*\ e'_p \\
ddcl' & ::= & \mathbf{data}\ D^n\ =\ cs' \mid \mathbf{data}\ Y^n\ =\ es' \\
cs' & ::= & \ldots \\
es' & ::= & \ldots \\
e'_p & ::= & \ldots
\end{array}
$$

We think they should probably be part of the type, with a grammar similar to the following (compare with definition 4.5):

$$
\begin{array}{lll}
\tau' & ::= & t \mid Int \mid \hat{n} \mid \tau' \to \tau' \mid (\tau', \ldots, \tau') \mid \mathbf{poly}\ \sigma \mid D^n \mid Y^n \\
\tau'_d & ::= & [\ ddcl'\ ]^*\ \tau' \\
\rho & ::= & \delta \Rightarrow \rho \mid \tau'_d \\
\sigma & ::= & s \mid \forall s.\sigma \mid \forall t.\sigma \mid \rho \\
\delta & ::= & \ldots
\end{array}
$$

Following this grammar, type variables in the summands of a data definition would be inside the scope of the type scheme σ, and properly constrained by predicates if necessary.

This alternative would not constitute a great alteration to our system — in fact, residual data declarations appear only in the constraint solving phase, leaving the specialization system intact — so it is probably a straightforward enhancement. However, some thought is required on how this new element would interact with polyvariance, and how types of the form **poly** σ would be solved.

Parametric data types An option that has not yet been considered for dynamic sum types is that of parametric data types. Parametric source data declarations can be handled as syntactic sugar for multiple data definitions. The most interesting aspect is that of generating

parametric residual sum types as an alternative to multiple residual type declarations. That is, instead of allowing for a single source data type to specialize to many residual versions, we could generate a single residual sum type and make it parametric. Where we currently have types $Y^1, \ldots, Y^n$, we could have a single data definition $Y\ t_1 \ldots t_k$ with as many arguments as needed to identify each former Y^i as $Y\ \tau_1 \ldots \tau_k$. This applies to regular dynamic sum types as well.

Recursion John Hughes's type specializer includes specialization of recursive dynamic data types, a feature still missing in principal type specialization, and without which certain essential data structures — such as lists — cannot be specialized.

It is not clear if recursive polyvariant sum types are of any use, so some analysis is required before an attempt to model it.

6.2.2 Work on principal type specialization

In his original presentation, Martínez López established a number of lines of work that remain open. They are briefly outlined below.

Dynamic recursion The interaction between dynamic recursive expressions and polyvariance is perhaps the most difficult problem in type specialization in general, and in principal type specialization in particular. Due to the way polyvariant expressions are specialized, recursive structures lead to predicates of the form IsMG $s\ \sigma$ where s appears in σ. This cannot be handled by the constraint solving mechanisms currently available, and needs further study.

Extending the source language Martínez López mentions a few important extensions to improve the level of expressiveness of the system. One of them is that of polyvariant sums, contributed by this work. Dynamic recursion, as was mentioned before, is essential, and specialization of imperative-like programs (monads) as was achieved for the original formulation of type specialization [Dussart *et al.*, 1997] is also a possible feature.

This framework also enables thinking about advanced features of modern languages, that have not been considered for any other known approach to program specialization. Examples of these are parametric polymorphism (in the source language), ad-hoc polymorphism, or overloading (type classes, for example), and programs with lazy behavior.

Better implementation Working with a complex theory like the one we have presented cannot be complete without an appropriate tool to test the ideas; additionally, as the final goal is to automatically produce programs, a proper implementation is a must.

The prototype designed so far is very naive, and no attempts has been made to make it efficient. As a result, only small examples can be tested — bigger ones take too much time.

There are many opportunities for improvement. The most time consuming part is that of constraint solving, so here is another reason for working on better algorithms as was mentioned before. However, there are also implementation enhancements that can be done: better representations could be found for predicates, terms, types and conversions to speed up some currently slow processes such as choosing a type variable to solve, performing substitutions, and comparing for equality.

Binding time assistant One important difference between type specialization and partial evaluation is the role of binding time annotations — annotations making an expression either static or dynamic. In partial evaluation, they can be deduced by a *binding time analyzer*. Type specialization, in contrast, has more flexibility, and can handle combinations of annotations that are not allowed in partial evaluation, so it is not possible to calculate them automatically, as shown by Martínez López [2005, chapter 4].

However, it has been noted that there are usually some rules of thumb on how to annotate a given program, and perhaps it would possible to construct a tool for assisting the annotation process. Such an assistant would suggest program points where it may be a good idea to make an expression polyvariant, where some variable would be better considered as dynamic, etc. It would also calculate annotations that depend on a particular choice, once it is made by the programmer.

A binding time assistant would be an excellent complement in an environment for automatic program production.

6.3 Concluding Remarks

In this thesis, we have studied the Principal Type Specialization system and extended it with polyvariant sums. Our contribution is a little step toward a powerful type specializer that can deal with features of a real programming language.

Working in this framework has been a challenging task, from which some observations and also a few questions have arisen. The Principal Type Specialization system is complex, and fully understanding it involves getting acquainted with a quite a number of elements:

- A language of *source* terms and types with *annotations*.
- A language of *residual* terms and types.
- A language of *predicates* and *evidence*, that are part of the residual types and terms respectively.
- An *entailment* relation, capturing the meaning of the predicates and their evidence, and, from a more practical point of view, specifying how evidence can be built.
- A *residual typing* system, suitable for proving that the residual language is typed.
- A *specialization* system, the core of this framework, specifying how typed source expressions are specialized to typed residual terms.
- A *source-residual* relation, similar to the one above but relating only source to residual types.
- An *algorithm*, composed itself by a number of subsystems, essential to proving the property of principality.

> *My dear young man, don't take it too hard. Your work is ingenious. It's quality work. And there are simply* ***too many notes****, that's all. Just cut a few and it will be perfect.*
>
> Emperor Joseph II to Mozart, in *Amadeus* (1984)

One cannot help but wonder why this framework seems so intricate. Why so many constructs, subsystems and rules? This thesis has taken around two years, the first of which was entirely devoted to learning the system, getting familiar with it and analyzing the elements present up to that moment. Indeed, this is a very elaborate system — the number of components and its many subtleties make it hard to master, which is certainly a disadvantage.

I believe part of the answer to this issue may be: principal type specialization takes care of detail. Besides the presence of a two-level (annotated) source language and a residual counterpart — which are features common to most program specializers — these many items make up a system that models *exactly* the specializations considered valid and useful for each source expression. Moreover, it defines a specialization that represents them all (the *principal* specialization), and specifies an algorithm for computing it. Despite being still incomplete, it does provide solutions to some difficult problems, such as polyvariance, lifting and specializing partially static expressions. These are all complex points, so it is only natural that a system dealing with them in such detail should be complex as well.

Surely simpler presentations are possible — John Hughes's formulation certainly is. However, his specification is more casual, it lacks all the good properties that Martínez López later achieves, and it leaves most of the tricky aspects for an implementation at least as intricate as the principal type specialization system is.

In a certain way, this last point is illustrated by our extension of polyvariant sums. Once achieved the necessary level of comfort working with the system, extending it was quite a natural task. The 'constraint generation - constraint solving' schema led us along a clear path: gather the *specification* of the residual sum type (via predicates) first, and then build one that satisfies the specification. After trying a few options, the final structure of this specification came out in the form of upper bounds and requirements on the arguments of the constructors.

If we compare our solution to the one John Hughes has provided for his original type specializer [Hughes, 1996b] (see comments in section 6.1.2), we can find some similarities in the way the residual sum types are built. Hughes has used a notion of *open sets* that are incrementally built by adding summands, and closed when it is found that no more of them are needed. Open sets are analogous to our type variables, the operation of adding summands is analogous to adding a predicate HasPolyC (a *requirement*) to our specification and closing the set is analogous to solving the variable. So our approach is not really much more complicated than his presentation, which is only simpler in appearance. On the contrary, we have gained insight on the difficulties related to performing specialization of polyvariant sums, and formulated them in a more clear way. We have achieved an acceptable solution and left the door open for trying alternative ones.

Was it all worth it yeah yeah
Giving all my heart and soul staying up all night
Was it all worth it...
Queen, *Was it all worth it* (1989)

So it is possible that the complexity of our framework is well justified. But having said that, a question remains about the type specialization approach in general: is it worth the effort? Other more mature approaches to program specialization have achieved similar results, or can achieve them if they are complemented with different pre and post processing mechanisms. Martínez López [2005, chapter 13] mentions several of them, among which partial

evaluation is the most popular and best understood. Several program specializers have been built that are efficient and powerful enough to be useful in practice.

Type specialization, on the contrary, is not well known, and still far from being fully developed. One could claim it is just a different — and complicated! — way of performing program specialization, with a formulation that is elegant in principle but too difficult to put into practice. Surely we can deal with most of the problems solved by type specialization with alternative, full-grown tools?

I cannot really venture a definitive answer to this question. With just a small subset of a language, type specialization has proved to be a powerful, flexible approach with a lot of potential. It can solve many of the problems currently handled with a combination of different techniques, and it leaves the door open for tackling features not considered so far. In order to grow, it still needs a great deal of work, and probably also the contribution of a greater number of people. But if it does, it can certainly become the favorite framework for program specialization.

Appendix A

Auxiliary systems and definitions

A.1 System RT

System RT is a type-checking system for residual terms. Judgments are of the form

$$\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma$$

meaning residual expression e' has type σ under predicate context Δ and context Γ_{R}, where $\Gamma_{\mathrm{R}} = \{x'_i : \tau_i \mid i = 1, \ldots, n\}$ maps residual variables to types.

The following are the residual typing rules for the expressions presented in section 2.2.

$$\text{(RT-VAR)}\quad \frac{x' : \tau' \in \Gamma_{\mathrm{R}}}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} x' : \tau'} \qquad \text{(RT-APP)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'_1 : \tau'_2 \to \tau'_1 \quad \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'_2 : \tau'_2}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'_1 @ e'_2 : \tau'_1}$$

$$\text{(RT-DINT)}\quad \frac{\Delta \Vdash v : \mathrm{IsInt}\ \tau'}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} v : \mathit{Int}} \qquad \text{(RT-LAM)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}}, x' : \tau'_2 \vdash_{\mathrm{RT}} e' : \tau'_1}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \lambda x'.e' : \tau'_2 \to \tau'_1}$$

$$\text{(RT-D+)}\quad \frac{(\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'_i : \mathit{Int})_{i=1,2}}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'_1 + e'_2 : \mathit{Int}} \qquad \text{(RT-TUPLE)}\quad \frac{(\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'_i : \tau'_i)_{i=1,..,n}}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} (e'_1, \ldots, e'_n) : (\tau'_1, \ldots, \tau'_n)}$$

$$\text{(RT-SINT)}\quad \frac{\Delta \Vdash v : \mathrm{IsInt}\ \tau'}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \bullet : \tau'} \qquad \text{(RT-PRJ)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : (\tau'_1, \ldots, \tau'_n)}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \pi_{i,n}\, e' : \tau'_i}$$

$$\text{(RT-LET)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'_2 : \tau'_2 \quad \Delta \mid \Gamma_{\mathrm{R}}, x' : \tau'_2 \vdash_{\mathrm{RT}} e'_1 : \tau'_1}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \mathbf{let}\ x' = e'_2\ \mathbf{in}\ e'_1 : \tau'_1}$$

$$\text{(RT-POLY)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma' \quad \Delta \Vdash v : \mathrm{IsMG}\ \sigma'\ \sigma}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} v[e'] : \mathbf{poly}\ \sigma}$$

$$\text{(RT-SPEC)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \mathbf{poly}\ \sigma \quad \Delta \Vdash v : \mathrm{IsMG}\ \sigma\ \tau'}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} v[e'] : \tau'}$$

$$\text{(RT-QIN)}\quad \frac{\Delta, \bar{h} : \delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \rho}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \Lambda h.e' : \delta \Rightarrow \rho} \qquad \text{(RT-GEN)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \forall\alpha.\sigma}\ \left(\alpha \notin FV(\Delta) \cup FV\left(\Gamma_{\mathrm{R}}\right)\right)$$

$$\text{(RT-QOUT)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \delta \Rightarrow \rho \quad \Delta \Vdash v : \delta}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'(\!(v)\!) : \rho} \qquad \text{(RT-INST)}\quad \frac{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \forall\alpha.\sigma}{\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : S\,\sigma}\ (\mathrm{dom}(S)=\alpha)$$

A.2 Computing principal type specializations

The systems we present below correspond to the description in section 2.4.

A.2.1 System W

System W describes an algorithm to compute principal type specializations. Judgments are of the form

$$\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \ \hookrightarrow\ e' : \sigma'$$

with the same meaning as in system P, where S is a substitution on the types of Γ. The rules can be interpreted as a grammar where Γ, e and τ are inherited attributes (i.e. input of the algorithm) and Δ, S, e' and σ' are synthesized (i.e. output).

$$\text{(W-VAR)}\quad \frac{x : \tau \hookrightarrow e' : \tau' \in \Gamma}{\emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} x : \tau \ \hookrightarrow\ e' : \tau'}$$

$$\text{(W-DINT)}\quad \emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} n^D : \mathit{Int}^D \ \hookrightarrow\ n : \mathit{Int}$$

$$\text{(W-D+)}\quad \frac{\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \mathit{Int}^D \ \hookrightarrow\ e_1' : \mathit{Int} \quad \Delta_2 \mid S_2\,(S_1\,\Gamma) \vdash_{\mathrm{W}} e_2 : \mathit{Int}^D \ \hookrightarrow\ e_2' : \mathit{Int}}{S_2\,\Delta_1, \Delta_2 \mid S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 +^D e_2 : \mathit{Int}^D \ \hookrightarrow\ e_1' + e_2' : \mathit{Int}}$$

$$\text{(W-LIFT)}\quad \frac{\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \mathit{Int}^S \ \hookrightarrow\ e' : \tau' \quad \Delta' \mid \Delta \Vdash_{\mathrm{W}} v : \mathrm{IsInt}\ \tau'}{\Delta', \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} \mathbf{lift}\ e : \mathit{Int}^D \ \hookrightarrow\ v : \mathit{Int}}$$

$$\text{(W-SINT)}\quad \emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} n^S : \mathit{Int}^S \ \hookrightarrow\ \bullet : \hat{n}$$

$$\text{(W-S+)}\quad \frac{\begin{array}{c}\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \mathit{Int}^S \ \hookrightarrow\ e_1' : \tau_1' \\ \Delta_2 \mid S_2\,(S_1\,\Gamma) \vdash_{\mathrm{W}} e_2 : \mathit{Int}^S \ \hookrightarrow\ e_2' : \tau_2' \\ \Delta \mid S_2\,\Delta_1, \Delta_2 \Vdash_{\mathrm{W}} v : t := S_2\,\tau_1' + \tau_2'\end{array}}{\Delta, S_2\,\Delta_1, \Delta_2 \mid S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 +^S e_2 : \mathit{Int}^S \ \hookrightarrow\ \bullet : t}\ (t \textit{ fresh})$$

$$\text{(W-DLAM)}\quad \frac{\Delta \vdash_{\text{W-SR}} \tau_2 \ \hookrightarrow\ \tau_2' \quad \Delta' \mid S\,(\Gamma, x : \tau_2 \hookrightarrow x' : \tau_2') \vdash_{\mathrm{W}} e : \tau_1 \ \hookrightarrow\ e' : \tau_1'}{\Delta', S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} \lambda^D x.e : \tau_2 \rightarrow^D \tau_1 \ \hookrightarrow\ \lambda x'.e' : S\,\tau_2' \rightarrow \tau_1'}\ (x' \textit{ fresh})$$

$$\text{(W-DAPP)}\quad \frac{\begin{array}{c}\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \tau_2 \to^D \tau_1 \;\hookrightarrow\; e_1' : \tau_1' \\ \Delta_2 \mid S_2\,(S_1\,\Gamma) \vdash_{\mathrm{W}} e_2 : \tau_2 \;\hookrightarrow\; e_2' : \tau_2' \qquad S_2\,\tau_1' \sim^U \tau_2' \to t\end{array}}{U S_2\,\Delta_1, U\,\Delta_2 \mid U S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 \mathbin{@^D} e_2 : \tau_1 \;\hookrightarrow\; e_1' @ e_2' : U\,t}\ (t \textit{ fresh})$$

$$\text{(W-POLY)}\quad \frac{\begin{array}{c}\bar{h} : \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \;\hookrightarrow\; e' : \tau' \\ \Delta' \mid \emptyset \Vdash_{\mathrm{W}} v : \mathrm{IsMG}\ (\mathrm{Gen}_{S\,\Gamma,\emptyset}(\Delta \Rightarrow \tau'))\ s\end{array}}{\Delta' \mid S\,\Gamma \vdash_{\mathrm{W}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \;\hookrightarrow\; v[\Lambda \bar{h}.e'] : \mathbf{poly}\ s}\ (s \textit{ fresh})$$

$$\text{(W-SPEC)}\quad \frac{\begin{array}{c}\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \mathbf{poly}\ \tau \;\hookrightarrow\; e' : \tau_\sigma' \qquad \tau_\sigma' \sim^U \mathbf{poly}\ s \\ \Delta' \vdash_{\text{W-SR}} \tau \;\hookrightarrow\; \tau' \qquad \Delta'' \mid U\,\Delta, \Delta' \Vdash_{\mathrm{W}} v : \mathrm{IsMG}\ (U\,s)\ \tau'\end{array}}{\Delta'', U\,\Delta, \Delta' \mid U S\,\Gamma \vdash_{\mathrm{W}} \mathbf{spec}\ e : \tau \;\hookrightarrow\; v[e'] : \tau'}\ (s \textit{ fresh})$$

$$\text{(W-DLET)}\quad \frac{\begin{array}{c}\Delta_2 \mid S_2\,\Gamma \vdash_{\mathrm{W}} e_2 : \tau_2 \;\hookrightarrow\; e_2' : \tau_2' \\ \Delta_1 \mid S_1\,(S_2\,\Gamma, x : \tau_2 \hookrightarrow x' : \tau_2') \vdash_{\mathrm{W}} e_1 : \tau_1 \;\hookrightarrow\; e_1' : \tau_1'\end{array}}{\begin{array}{l}S_1\,\Delta_2, \Delta_1 \mid S_1 S_2\,\Gamma \vdash_{\mathrm{W}} \mathbf{let}^D\ x = e_2\ \mathbf{in}\ e_1 : \tau_1 \\ \qquad\qquad \hookrightarrow\ \mathbf{let}\ x' = e_2'\ \mathbf{in}\ e_1' : \tau_1'\end{array}}\ (x' \textit{ fresh})$$

$$\text{(W-DTUPLE)}\quad \frac{\begin{array}{c}\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \tau_1 \;\hookrightarrow\; e_1' : \tau_1' \\ \ldots \qquad \Delta_n \mid S_n\,S_{n-1} \ldots S_1\,\Gamma \vdash_{\mathrm{W}} e_n : \tau_n \;\hookrightarrow\; e_n' : \tau_n'\end{array}}{\begin{array}{l}S_n \ldots S_2\,\Delta_1, \ldots, \Delta_n \mid S_n \ldots S_1\,\Gamma \\ \quad \vdash_{\mathrm{W}} (e_1, \ldots, e_n)^{\,D} : (\tau_1, \ldots, \tau_n)^{\,D} \\ \quad\quad \hookrightarrow\ (e_1', \ldots, e_n') : (S_n \ldots S_2\,\tau_1', S_n \ldots S_3\,\tau_2', \ldots, \tau_n')\end{array}}$$

$$\text{(W-DPRJ)}\quad \frac{\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : (\tau_1, \ldots, \tau_n)^{\,D} \;\hookrightarrow\; e' : \tau' \quad \tau' \sim^U (t_1, \ldots, t_n)}{U\,\Delta \mid U S\,\Gamma \vdash_{\mathrm{W}} \pi_{i,n}^D\,e : \tau_i \;\hookrightarrow\; \pi_{i,n}\,e' : U\,t_i}\ (t_1,\ldots,t_n \textit{ fresh})$$

A.2.2 Unification

The algorithm calculating a most general unifier for two residual types is described by rules of the form $\sigma_1 \sim^U \sigma_2$, where σ_1 and σ_2 are input and U is output, if it exists.

$$c \sim^{\mathrm{Id}} c \qquad \hat{n} \sim^{\mathrm{Id}} \hat{n} \qquad \mathit{Int} \sim^{\mathrm{Id}} \mathit{Int}$$

$$\alpha \sim^{\mathrm{Id}} \alpha \qquad \frac{\alpha \notin FV(\sigma)}{\alpha \sim^{[\alpha/\sigma]} \sigma} \qquad \frac{\tau_1' \sim^T \tau_1'' \quad T\,\tau_2' \sim^U T\,\tau_2''}{\tau_1' \to \tau_2' \sim^{UT} \tau_1'' \to \tau_2''}$$

$$\frac{\tau_{11}' \sim^{T_1} \tau_{21}' \quad T_1\,\tau_{12}' \sim^{T_2} T_1\,\tau_{22}' \quad \ldots \quad T_{n-1} \ldots T_1\,\tau_{1n}' \sim^{T_n} T_{n-1} \ldots T_1\,\tau_{2n}'}{(\tau_{11}', \ldots, \tau_{1n}') \sim^{T_n \ldots T_1} (\tau_{21}', \ldots, \tau_{2n}')}$$

$$\frac{\sigma \sim^U \sigma'}{\mathbf{poly}\ \sigma \sim^U \mathbf{poly}\ \sigma'} \qquad \frac{\sigma[\alpha/c] \sim^U \sigma'[\alpha'/c]}{\forall\alpha.\sigma \sim^U \forall\alpha'.\sigma'}\ (c \text{ fresh})$$

$$\frac{\delta \sim^U \delta' \quad \rho \sim^U \rho'}{\delta \Rightarrow \rho \sim^U \delta' \Rightarrow \rho'} \qquad \frac{\tau \sim^T \tau' \quad T\,\tau_1 \sim^U T\,\tau_1' \quad UT\,\tau_2 \sim^V UT\,\tau_2'}{\tau := \tau_1 + \tau_2 \sim^{VUT} \tau' := \tau_1' + \tau_2'}$$

$$\frac{\tau \sim^U \tau'}{\text{IsInt}\ \tau \sim^U \text{IsInt}\ \tau'} \qquad \frac{\sigma_1 \sim^T \sigma_2 \quad T\,\sigma_1' \sim^U T\,\sigma_2'}{\text{IsMG}\ \sigma_1\ \sigma_1' \sim^{UT} \text{IsMG}\ \sigma_2\ \sigma_2'}$$

A.2.3 System W-SR

System W-SR describes an algorithm for computing the source-residual relationship. Judgments are of the form $\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'$ with the same meaning as system SR, where τ is input and Δ and τ' are output.

$$h : \text{IsInt}\ t \vdash_{\text{W-SR}} Int^S \hookrightarrow t \quad (t \text{ fresh})$$

$$\emptyset \vdash_{\text{W-SR}} Int^D \hookrightarrow Int$$

$$\frac{\Delta_1 \vdash_{\text{W-SR}} \tau_1 \hookrightarrow \tau_1' \quad \Delta_2 \vdash_{\text{W-SR}} \tau_2 \hookrightarrow \tau_2'}{\Delta_1, \Delta_2 \vdash_{\text{W-SR}} \tau_2 \to^D \tau_1 \hookrightarrow \tau_2' \to \tau_1'}$$

$$\frac{(\Delta_i \vdash_{\text{W-SR}} \tau_i \hookrightarrow \tau_i')_{i=1,\ldots,n}}{\Delta_1, \ldots, \Delta_n \vdash_{\text{W-SR}} (\tau_1, \ldots, \tau_n)^D \hookrightarrow (\tau_1', \ldots, \tau_n')}$$

$$\frac{\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'}{\text{IsMG}\ \sigma\ s \vdash_{\text{W-SR}} \mathbf{poly}\ \tau \hookrightarrow \mathbf{poly}\ s}\ \left(\sigma = \text{Gen}_{\emptyset,\emptyset}(\Delta \Rightarrow \tau') \text{ and } s \text{ fresh}\right)$$

A.3 Extending system RT for sum types

Martínez López's extension [2005] does not include rules to type residual terms derived from static sums. These are straightforward and can be deduced from the specialization rules.

The rules for system RT involving dynamic sums are the following:

$$\text{(RT-DCONSTR)}\quad \frac{\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : \tau_j' \qquad \Delta \Vdash \text{IsSum}\ \tau' \quad \Delta \Vdash v_j : \text{HasC}\ \tau'\ K_j\ \tau_j'}{\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} K_j^{v_j}\ e' : \tau'}$$

$$\text{(RT-DCASE)}\quad \frac{\begin{array}{l}\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_e \\ \Delta \Vdash v^d : \mathrm{IsSum}\ \tau'_e \\ \left(\begin{array}{l}\bar{h}_k : \Delta_k \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \lambda x'_k.e'_k : \tau'_k \to \tau' \\ \Delta \Vdash \bar{v}_k : K_k \in \tau'_e \,?\, \Delta_k \\ \Delta \Vdash w_k : K_k \in \tau'_e \,?\, \mathrm{HasC}\ \tau'_e\ \ K_k\ \ \tau'_k\end{array}\right)_{k\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}}\ \mathbf{protocase}_v\ e'\ \mathbf{with}\ v^d\ \mathbf{of} \\ \qquad\qquad (K_k^{w_k}\ x'_k\ \to\ e'_k[\bar{v}_k/\bar{h}_k])_{k\in B} : \tau'\end{array}}$$

A.4 Substitution of evidence variables

We extend the definition of substitution of evidence variables in residual expressions for the new constructs presented in section 4.2.

$$\begin{array}{lcl}
\left(L_k^{v_k}\ e'\right)[\bar{v}/\bar{h}] & = & L_k^{v_k[\bar{v}/\bar{h}]}\ e'[\bar{v}/\bar{h}] \\[2ex]
\left(\begin{array}{l}\mathbf{case}\ e'\ \mathbf{of} \\ \ \left(L_k^{v_k}\ x'_k\ \to\ e'_k\right)_{k\in B}\end{array}\right)[\bar{v}/\bar{h}] & = & \begin{array}{l}\mathbf{case}\ e'[\bar{v}/\bar{h}]\ \mathbf{of} \\ \ \left(L_k^{v_k[\bar{v}/\bar{h}]}\ x'_k\ \to\ e'_k[\bar{v}/\bar{h}]\right)_{k\in B}\end{array} \\[2ex]
\left(\begin{array}{l}\mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y \\ \ \mathbf{and}\ (w_k)_{k\in B}\ \mathbf{of} \\ \ (L_k\ \to\ e'_k)_{k\in B}\end{array}\right)[\bar{v}/\bar{h}] & = & \begin{array}{l}\mathbf{polycase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^y[\bar{v}/\bar{h}] \\ \ \mathbf{and}\ \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B}\ \mathbf{of} \\ \ \left(L_k\ \to\ e'_k[\bar{v}/\bar{h}]\right)_{k\in B}\end{array} \\[2ex]
(\langle n, w_k\rangle)[\bar{v}/\bar{h}] & = & \left\langle n, \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B}\right\rangle
\end{array}$$

A.5 Substitution of type variables in predicates

We extend the definition of type variable substitutions for the predicates we have introduced in section 4.2.2.

$$\begin{array}{lcl}
(\mathrm{IsPolySum}\ \tau)[\tau'/t] & = & \mathrm{IsPolySum}\ \tau[\tau'/t] \\
(\mathrm{HasMGC}\ \tau\ \ L_k\ \ \sigma)[\tau'/t] & = & \mathrm{HasMGC}\ \tau[\tau'/t]\ \ L_k\ \ \sigma[\tau'/t] \\
(\mathrm{HasMGBr}\ \tau\ \ L_k\ \ \sigma\ \ \tau_{br})[\tau'/t] & = & \mathrm{HasMGBr}\ \tau[\tau'/t]\ \ L_k\ \ \sigma[\tau'/t]\ \ \tau_{br}[\tau/t] \\
(\mathrm{HasPolyC}\ \tau_1\ \ L_k\ \ \tau_2)[\tau'/t] & = & \mathrm{HasPolyC}\ \tau_1[\tau'/t]\ \ L_k\ \ \tau_2[\tau'/t] \\
(L \in \tau \,?\, \delta)[\tau'/t] & = & L \in \tau[\tau'/t] \,?\, \delta[\tau'/t]
\end{array}$$

A.6 Extending the definition of equivalence of residual terms

The definition of equivalence of residual terms is extended to handle $\mathbf{polycase}_v$ expressions with evidence variables. The equivalence (=) relation is defined as the smallest congruence containing the reduction rules and the following:

$$\begin{array}{l}\textbf{polycase}_v\ e'\ \textbf{with}\ h\ \textbf{and} \\ \quad (w_k)_{k\in B}\ \textbf{of} \\ \quad (L_k\ \rightarrow\ e'_k)_{k\in B}\end{array} \quad = \quad \begin{array}{l}\textbf{polycase}_v\ e''\ \textbf{with}\ h'\ \textbf{and} \\ \quad (w'_k)_{k\in B}\ \textbf{of} \\ \quad (L_k\ \rightarrow\ e''_k)_{k\in B}\end{array}$$

if and only if $\forall Y = \{L_{k,i}\}_{k\in I, i\in I'_k}$

$$\begin{array}{l}\textbf{polycase}_v\ e'[Y/h]\ \textbf{with}\ Y\ \textbf{and} \\ \quad (w_k[Y/h])_{k\in B}\ \textbf{of} \\ \quad (L_k\ \rightarrow\ e'_k[Y/h])_{k\in B}\end{array} \quad = \quad \begin{array}{l}\textbf{polycase}_v\ e''[Y/h']\ \textbf{with}\ Y\ \textbf{and} \\ \quad (w'_k[Y/h'])_{k\in B}\ \textbf{of} \\ \quad (L_k\ \rightarrow\ e''_k[Y/h'])_{k\in B}\end{array}$$

$$\begin{array}{l}\textbf{polycase}_v\ e'\ \textbf{with}\ v^y\ \textbf{and} \\ \quad (w_1, \ldots, h_i, \ldots, w_k)\ \textbf{of} \\ \quad (L_k\ \rightarrow\ e'_k)_{k\in B}\end{array} \quad = \quad \begin{array}{l}\textbf{polycase}_v\ e''\ \textbf{with}\ v^y\ \textbf{and} \\ \quad (w'_1, \ldots, h'_i, \ldots, w'_k)\ \textbf{of} \\ \quad (L_k\ \rightarrow\ e''_k)_{k\in B}\end{array}$$

if and only if $\forall u = \left\langle n, \left(v'_j\right)_{j\in I}\right\rangle$

$$\begin{array}{l}\textbf{polycase}_v\ e'[u/h_i]\ \textbf{with}\ v^y\ \textbf{and} \\ \quad (w_1[u/h_i], \ldots, u, \ldots, w_k[u/h_i])\ \textbf{of} \\ \quad (L_k\ \rightarrow\ e'_k[u/h_i])_{k\in B}\end{array} \quad = \quad \begin{array}{l}\textbf{polycase}_v\ e''[u/h'_i]\ \textbf{with}\ v^y\ \textbf{and} \\ \quad (w'_1[u/h'_i], \ldots, u, \ldots, w'_k[u/h'_i])\ \textbf{of} \\ \quad (L_k\ \rightarrow\ e''_k[u/h'_i])_{k\in B}\end{array}$$

Appendix B

Proofs

Most proofs are extensions to the proofs given in Pablo E. Martínez López's PhD thesis [2005] and Alejandro Russo's graduate thesis [2004].

All of them are proofs by induction on the structure of a derivation, and many follow the same argument pattern. We have kept references to similar proofs to a minimum, so that each one could be read independently. As a result, they may be found repetitive if read straight through — this appendix is meant to be consulted for individual proofs instead.

We have marked with a star ($\star$) the proofs we consider most interesting or complex.

B.1 Proof of lemma 4.6, section 4.2

Lemma 4.6 *If* $\bar{h} : \Delta \Vdash v^y : \text{IsPolySum } \tau'$,
$\bar{h}_1 : \Delta'_1 \Vdash \bar{v}_2 : \Delta'_2$ *and*
$\bar{h} : \Delta \Vdash \bar{v}_1 : L_k \in \tau' \,?\, \Delta'_1$
then $\bar{h} : \Delta \Vdash \bar{v}' : L_k \in \tau' \,?\, \Delta'_2$
where $\bar{v}' = \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ \bar{v}_2[\bar{v}_1/\bar{h}_1]\ \mathbf{else}\ \bullet$

Proof: We know that

$$\bar{h} : \Delta \Vdash v^y : \text{IsPolySum } \tau'$$
$$\bar{h}_1 : \Delta'_1 \Vdash \bar{v}_2 : \Delta'_2$$

So by rule (IsPolySum-Guard),

$$\Delta, \bar{h}_1 : L_k \in \tau' \,?\, \Delta'_1 \Vdash \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ \bar{v}_2\ \mathbf{else}\ \bullet : L_k \in \tau' \,?\, \Delta'_2$$

Let us call $\bar{v}'_1 = \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ \bar{v}_2\ \mathbf{else}\ \bullet$, so now we have:

$$\bar{h} : \Delta, \bar{h}_1 : L_k \in \tau' \,?\, \Delta'_1 \Vdash \bar{v}'_1 : L_k \in \tau' \,?\, \Delta'_2 \tag{B.1}$$

We also know by hypothesis that

$$\bar{h} : \Delta \Vdash \bar{v}_1 : L_k \in \tau' \,?\, \Delta'_1 \tag{B.2}$$

By rule (Cut) on B.2 and B.1,

$$\bar{h} : \Delta \Vdash \bar{v}'_1[\bar{v}_1/\bar{h}_1] : L_k \in \tau' \,?\, \Delta'_2$$

Let us take $\bar{v}' = \bar{v}'_1[\bar{v}_1/\bar{h}_1]$. By evidence substitution,

$$\bar{v}' = (\mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ \bar{v}_2\ \mathbf{else}\ \bullet)\,[\bar{v}_1/\bar{h}_1] = \mathbf{if}_v\ L_k \in v^y[\bar{v}_1/\bar{h}_1]\ \mathbf{then}\ \bar{v}_2[\bar{v}_1/\bar{h}_1]\ \mathbf{else}\ \bullet\,[\bar{v}_1/\bar{h}_1]$$

Now by alpha conversion we can assume $\bar{h}$ and $\bar{h}_1$ are disjoint. By rule (Evars), $EV(v^y) \subseteq \bar{h}$, so $\bar{h}_1$ do not appear free in v^y and we finally get

$$\bar{v}' = \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ \bar{v}_2[\bar{v}_1/\bar{h}_1]\ \mathbf{else}\ \bullet$$

which completes our proof.

B.2 Proof of lemma 4.8, section 4.3

Lemma 4.8 *If* $\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma$
then $EV(e') \subseteq \bar{h}$

Proof: By induction on the RT derivation.

Case (RT-VAR): We have a derivation of the form

$$\frac{x' : \tau' \in \Gamma_{\mathrm{R}}}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} x' : \tau'}$$

We need to prove that $EV(x') \subseteq \bar{h}$, which is trivial, since $EV(x') = \emptyset$.

Case (RT-DINT): We have a derivation of the form

$$\frac{\bar{h} : \Delta \Vdash v : \mathrm{IsInt}\ \tau'}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} v : \mathit{Int}}$$

We need to prove that $EV(v) \subseteq \bar{h}$, which follows directly from rule (Evars) on the premise of the rule.

Case (RT-D+): We have a derivation of the form

$$\frac{(\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'_i : \mathit{Int})_{i=1,2}}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'_1 + e'_2 : \mathit{Int}}$$

We need to prove that $EV(e'_1 + e'_2) \subseteq \bar{h}$.

Now by inductive hypothesis, we have that $EV(e'_1) \subseteq \bar{h}$ and $EV(e'_2) \subseteq \bar{h}$, so

$$EV\left(e'_1 + e'_2\right) = EV\left(e'_1\right) \cup EV\left(e'_2\right) \subseteq \bar{h}$$

Case (RT-SINT): We have a derivation of the form

$$\frac{\bar{h} : \Delta \Vdash v : \text{IsInt } \tau'}{\bar{h} : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} \bullet : \tau'}$$

We need to prove that $EV(\bullet) \subseteq \bar{h}$, which is trivial, since $EV(\bullet) = \emptyset$.

Case (RT-TUPLE): We have a derivation of the form

$$\frac{(\bar{h} : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e'_i : \tau'_i)_{i=1,..,n}}{\bar{h} : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} (e'_1, \ldots, e'_n) : (\tau'_1, \ldots, \tau'_n)}$$

We need to prove that $EV((e'_1, \ldots, e'_n)) \subseteq \bar{h}$. Now for all $i = 1, \ldots, n$, we have by inductive hypothesis that $EV(e'_i) \subseteq \bar{h}$. So

$$EV\big((e'_1, \ldots, e'_n)\big) = EV\big(e'_1\big) \cup \ldots \cup EV\big(e'_n\big) \subseteq \bar{h}$$

Case (RT-PRJ): We have a derivation of the form

$$\frac{\bar{h} : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : (\tau'_1, \ldots, \tau'_n)}{\bar{h} : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} \pi_{i,n}\, e' : \tau'_i}$$

We want to prove that $EV(\pi_{i,n}\, e') \subseteq \bar{h}$. This follows from the inductive hypothesis, by wich $EV(e') \subseteq \bar{h}$, and from the fact that $EV(\pi_{i,n}\, e') = EV(e')$.

Case (RT-LAM): We have a derivation of the form

$$\frac{\bar{h} : \Delta \mid \Gamma_{\text{R}}, x' : \tau'_2 \vdash_{\text{RT}} e' : \tau'_1}{\bar{h} : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} \lambda x'.e' : \tau'_2 \to \tau'_1}$$

We want to show that $EV(\lambda x'.e') \subseteq \bar{h}$. This follows directly from the inductive hypothesis, by which $EV(e') \subseteq \bar{h}$, and from the fact that $EV(\lambda x'.e') = EV(e')$.

Case (RT-APP): We have a derivation of the form

$$\text{(RT-APP)} \quad \frac{\bar{h} : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e'_1 : \tau'_2 \to \tau'_1 \quad \bar{h} : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e'_2 : \tau'_2}{\bar{h} : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e'_1 @ e'_2 : \tau'_1}$$

We need to prove that $EV(e'_1 @ e'_2) \subseteq \bar{h}$. Now by inductive hypothesis, we have that $EV(e'_1) \subseteq \bar{h}$ and $EV(e'_2) \subseteq \bar{h}$. So

$$EV\big(e'_1 @ e'_2\big) = EV\big(e'_1\big) \cup EV\big(e'_2\big) \subseteq \bar{h}$$

Case (RT-LET): We have a derivation of the form

$$\frac{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e_2' : \tau_2' \quad \bar{h} : \Delta \mid \Gamma_{\mathrm{R}}, x' : \tau_2' \vdash_{\mathrm{RT}} e_1' : \tau_1'}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \mathbf{let}\ x' = e_2'\ \mathbf{in}\ e_1' : \tau_1'}$$

We want to prove that $EV(\mathbf{let}\ x' = e_2'\ \mathbf{in}\ e_1') \subseteq \bar{h}$. Now by inductive hypothesis, $EV(e_1') \subseteq \bar{h}$ and $EV(e_2') \subseteq \bar{h}$. So

$$EV\big(\mathbf{let}\ x' = e_2'\ \mathbf{in}\ e_1'\big) = EV\big(e_1'\big) \cup EV\big(e_2'\big) \subseteq \bar{h}$$

Case (RT-POLY): We have a derivation of the form

$$\frac{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma' \quad \Delta \Vdash v : \mathrm{IsMG}\ \sigma'\ \sigma}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} v[e'] : \mathbf{poly}\ \sigma}$$

We need to show that $EV(v[e']) \subseteq \bar{h}$. Now by inductive hypothesis on the first premise of the rule, we have that $EV(e') \subseteq \bar{h}$. Also, by rule (Evars) on the second premise, $EV(v) \subseteq \bar{h}$. So

$$EV\big(v[e']\big) = EV(v) \cup EV\big(e'\big) \subseteq \bar{h}$$

Case (RT-SPEC): We have a derivation of the form

$$\frac{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \mathbf{poly}\ \sigma \quad \bar{h} : \Delta \Vdash v : \mathrm{IsMG}\ \sigma\ \tau'}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} v[e'] : \tau'}$$

We need to show that $EV(v[e']) \subseteq \bar{h}$. Now by inductive hypothesis on the first premise of the rule, we have that $EV(e') \subseteq \bar{h}$. Also, by rule (Evars) on the second premise, $EV(v) \subseteq \bar{h}$. So

$$EV\big(v[e']\big) = EV(v) \cup EV\big(e'\big) \subseteq \bar{h}$$

Case (RT-QIN): We have a derivation of the form

$$\frac{\bar{h} : \Delta, h' : \delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \rho}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \Lambda h'.e' : \delta \Rightarrow \rho}$$

We want to show that $EV(\Lambda h'.e') \subseteq \bar{h}$. By inductive hypothesis, we know that $EV(e') \subseteq \bar{h} \cup \{h'\}$. So

$$EV\big(\Lambda h'.e'\big) = EV\big(e'\big) \setminus \{h'\} \subseteq \bar{h}$$

Case (RT-QOUT): We have a derivation of the form

$$\frac{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \delta \Rightarrow \rho \quad \bar{h} : \Delta \Vdash v : \delta}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'(\!(v)\!) : \rho}$$

We need to prove that $EV(e'(\!(v)\!)) \subseteq \bar{h}$. By inductive hypothesis on the first premise, $EV(e') \subseteq \bar{h}$, and by rule (Evars) on the second premise, $EV(v) \subseteq \bar{h}$. So

$$EV\big(e'(\!(v)\!)\big) = EV\big(e'\big) \cup EV(v) \subseteq \bar{h}$$

Case (RT-GEN): We have a derivation of the form

$$\text{(RT-GEN)} \quad \frac{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \forall \alpha.\sigma} \left(\alpha \notin FV(\Delta) \cup FV\big(\Gamma_{\mathrm{R}}\big)\right)$$

We need to prove that $EV(e') \subseteq \bar{h}$, which holds directly from the inductive hypothesis.

Case (RT-INST): We have a derivation of the form

$$\frac{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \forall \alpha.\sigma}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : S\,\sigma} \ (\mathrm{dom}(S) = \alpha)$$

We need to prove that $EV(e') \subseteq \bar{h}$, which holds directly from the inductive hypothesis.

Case (RT-DCONSTR): We have a derivation of the form

$$\frac{\begin{array}{c}\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_j \\ \bar{h} : \Delta \Vdash \mathrm{IsSum}\ \tau' \qquad \bar{h} : \Delta \Vdash v_j : \mathrm{HasC}\ \tau'\ \ K_j\ \ \tau'_j\end{array}}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} K_j^{v_j}\ e' : \tau'}$$

We need to prove that $EV\left(K_j^{v_j}\ e'\right) \subseteq \bar{h}$. Now by inductive hypothesis on the first premise, $EV(e') \subseteq \bar{h}$, and by rule (Evars) on the third premise, $EV(v_j) \subseteq \bar{h}$. So

$$EV\left(K_j^{v_j}\ e'\right) = EV(v_j) \cup EV\big(e'\big) \subseteq \bar{h}$$

Case (RT-DCASE): We have a derivation of the form

$$\frac{\begin{array}{l}\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_e \\ \bar{h} : \Delta \Vdash v^d : \mathrm{IsSum}\ \tau'_e \\ \left(\begin{array}{l}\bar{h}_k : \Delta_k \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \lambda x'_k.e'_k : \tau'_k \to \tau' \\ \bar{h} : \Delta \Vdash \bar{v}_k : K_k \in \tau'_e\,?\,\Delta_k \\ \bar{h} : \Delta \Vdash w_k : K_k \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_k\ \ \tau'_k\end{array}\right)_{k \in B}\end{array}}{\begin{array}{l}\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \ \mathbf{protocase}_v\ e'\ \mathbf{with}\ v^d\ \mathbf{of} \\ \qquad\qquad (K_k^{w_k}\ x'_k\ \to\ e'_k[\bar{v}_k/\bar{h}_k])_{k \in B} : \tau'\end{array}}$$

We need to prove that

$$EV\left(\begin{array}{l}\textbf{protocase}_v\ e'\ \textbf{with}\ v^d\ \textbf{of}\\ \left(K_k^{w_k}\ x'_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B}\end{array}\right) \subseteq \bar{h}$$

Now by inductive hypothesis on the first and third premises

$$EV\left(e'\right) \subseteq \bar{h} \tag{B.3}$$
$$EV\left(e'_k\right) \subseteq \bar{h}_k\ \forall k \in B \tag{B.4}$$

By rule (Evars) on the remaining premises

$$EV\left(v^d\right) \subseteq \bar{h} \tag{B.5}$$
$$EV(\bar{v}_k) \subseteq \bar{h}\ \forall k \in B \tag{B.6}$$
$$EV(w_k) \subseteq \bar{h}\ \forall k \in B \tag{B.7}$$

From B.4 we can conclude that $EV\left(e'_k[\bar{v}_k/\bar{h}_k]\right) = EV(\bar{v}_k)$ by substitution. So from B.6,

$$EV\left(e'_k[\bar{v}_k/\bar{h}_k]\right) = EV(\bar{v}_k) \subseteq \bar{h} \tag{B.8}$$

Finally, from B.3, B.5, B.7 and B.8 we can conclude

$$EV\left(\begin{array}{l}\textbf{protocase}_v\ e'\ \textbf{with}\ v^d\ \textbf{of}\\ \left(K_k^{w_k}\ x'_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B}\end{array}\right) \quad =$$

$$EV(e') \cup EV\left(v^d\right) \cup \bigcup_{k\in B}\left(EV(w_k) \cup EV\left(e'_k[\bar{v}_k/\bar{h}_k]\right)\right) \quad \subseteq \quad \bar{h}$$

As we wanted to prove.

Case (RT-DCASE-2): We have a derivation of the form

$$\frac{\begin{array}{l}\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_e\\ \bar{h} : \Delta \Vdash v^d : \mathrm{IsSum}\ \tau'_e\\ \left(\begin{array}{l}\bar{h}_j : \Delta_j \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \lambda x'_j.e'_j : \tau'_j \rightarrow \tau''_j\\ \bar{h} : \Delta \Vdash \bar{v}_j : K_j \in \tau'_e\,?\,\Delta_j\\ \bar{h} : \Delta \Vdash w_j : K_j \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j\\ \bar{h} : \Delta \Vdash K_j \in \tau'_e\,?\,\tau''_j \sim \tau'\end{array}\right)_{j\in B}\end{array}}{\begin{array}{l}\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}}\ \ \textbf{protocase}_v\ e'\ \textbf{with}\ v^d\ \textbf{of}\\ \qquad\qquad\qquad \left(K_j^{w_j}\ x'_j\ \rightarrow\ e'_j[\bar{v}_j/\bar{h}_j]\right)_{j\in B} : \tau'\end{array}}$$

The proof that

$$EV\left(\begin{array}{l}\textbf{protocase}_v\ e'\ \textbf{with}\ v^d\ \textbf{of}\\ \left(K_k^{w_k}\ x'_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B}\end{array}\right) \subseteq \bar{h}$$

is completely analogous to the previous case.

Case (RT-POLYCONSTR): We have a derivation of the form

$$\frac{\begin{array}{c}\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_j \\ \bar{h} : \Delta \Vdash \mathrm{IsPolySum}\ \tau' \quad \bar{h} : \Delta \Vdash v_j : \mathrm{HasPolyC}\ \tau'\ \ L_j\ \ \tau'_j\end{array}}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} L_j^{\,v_j}\ e' : \tau'}$$

We need to prove that $EV\left(L_j^{\,v_j}\ e'\right) \subseteq \bar{h}$, which can be done the same way as in case (RT-DCONSTR).

Case (RT-POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l}\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_e \\ \bar{h} : \Delta \Vdash v^y : \mathrm{IsPolySum}\ \tau'_e \\ \left(\begin{array}{l}\bar{h}_k : \Delta_k \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e''_k : \sigma_k \\ \bar{h} : \Delta \Vdash \bar{v}_k : L_k \in \tau'_e\,?\,\Delta_k \\ h : \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'\end{array}\right)_{k \in B}\end{array}}{\begin{array}{l}h : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}}\ \ \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and} \\ \qquad\qquad (w_k)_{k \in B}\ \mathbf{of}\ (L_k\ \rightarrow\ e''_k[\bar{v}_k/\bar{h}_k])_{k \in B} : \tau'\end{array}}$$

We need to prove that

$$EV\left(\begin{array}{l}\mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and} \\ \quad (w_k)_{k \in B}\ \mathbf{of}\ (L_k\ \rightarrow\ e''_k[\bar{v}_k/\bar{h}_k])_{k \in B}\end{array}\right) \subseteq \bar{h}$$

Now by inductive hypothesis on the first and third premises

$$EV\left(e'\right) \subseteq \bar{h} \tag{B.9}$$

$$EV\left(e'_k\right) \subseteq \bar{h}_k\ \forall k \in B \tag{B.10}$$

By rule (Evars) on the remaining premises

$$EV\left(v^d\right) \subseteq \bar{h} \tag{B.11}$$

$$EV(\bar{v}_k) \subseteq \bar{h}\ \forall k \in B \tag{B.12}$$

$$EV(w_k) \subseteq \bar{h}\ \forall k \in B \tag{B.13}$$

From B.10 we can conclude that $EV\left(e'_k[\bar{v}_k/\bar{h}_k]\right) = EV(\bar{v}_k)$ by substitution. Then, from B.12,

$$EV\left(e'_k[\bar{v}_k/\bar{h}_k]\right) = EV(\bar{v}_k) \subseteq \bar{h} \tag{B.14}$$

Finally, from B.9, B.11, B.13 and B.14 we can conclude

$$EV\left(\begin{array}{l}\mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and} \\ \quad (w_k)_{k \in B}\ \mathbf{of}\ (L_k\ \rightarrow\ e''_k[\bar{v}_k/\bar{h}_k])_{k \in B}\end{array}\right) \quad =$$

$$EV(e') \cup EV(v^y) \cup \textstyle\bigcup_{k \in B} \left(EV(w_k) \cup EV\left(e'_k[\bar{v}_k/\bar{h}_k]\right)\right) \quad \subseteq \quad \bar{h}$$

As we wanted to prove.

B.3 Proof of proposition 4.9, section 4.3

Proposition 4.9 *If* $\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma$ *and* $\Delta' \Vdash \bar{v} : \Delta$ *then* $\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'[\bar{v}/\bar{h}] : \sigma$

Proof: By induction on the RT derivation.
Extending proofs of propositions by Martínez López [2005, 6.11] and Russo [2004, 3.7].

Case (RT-DCASE-2): We have a derivation of the form

$$\frac{\begin{array}{l}\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_e \\ \bar{h} : \Delta \Vdash v^d : \mathrm{IsSum}\ \tau'_e \\ \left(\begin{array}{l}\bar{h}_j : \Delta_j \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \lambda x'_j.e'_j : \tau'_j \to \tau''_j \\ \bar{h} : \Delta \Vdash \bar{v}_j : K_j \in \tau'_e\,?\,\Delta_j \\ \bar{h} : \Delta \Vdash w_j : K_j \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \\ \bar{h} : \Delta \Vdash K_j \in \tau'_e\,?\,\tau''_j \sim \tau'\end{array}\right)_{j\in B}\end{array}}{\begin{array}{l}\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \ \mathbf{protocase}_v\ e'\ \mathbf{with}\ v^d\ \mathbf{of} \\ \qquad\qquad (K_j^{\,w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : \tau'\end{array}}$$

We need to prove that

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \left(\begin{array}{l}\mathbf{protocase}_v\ e'\ \mathbf{with}\ v^d\ \mathbf{of} \\ \quad (K_j^{\,w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B}\end{array}\right)[\bar{v}/\bar{h}] : \tau'$$

By the definition of substitution on a $\mathbf{protocase}_v$ expression, this is the same as proving that

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \left(\begin{array}{l}\mathbf{protocase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^d[\bar{v}/\bar{h}]\ \mathbf{of} \\ \quad (K_j^{\,w_j[\bar{v}/\bar{h}]}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j][\bar{v}/\bar{h}])_{j\in B}\end{array}\right) : \tau'$$

We can also assume (by alpha conversion) that $\bar{h}_j$ and $\bar{h}$ are disjoint for all j. Since by lemma 4.8 on the third premise, $EV\left(e'_j\right) = EV\left(\lambda x'_j.e'_j\right) \subseteq \bar{h}_j$, evidence $\bar{h}$ do not appear free in e'_j and the judgement above is equivalent to

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \left(\begin{array}{l}\mathbf{protocase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^d[\bar{v}/\bar{h}]\ \mathbf{of} \\ \quad (K_j^{\,w_j[\bar{v}/\bar{h}]}\ x'_j\ \to\ e'_j[\bar{v}_j[\bar{v}/\bar{h}]/\bar{h}_j])_{j\in B}\end{array}\right) : \tau'$$

Now one of our hypothesis is $\Delta' \Vdash \bar{v} : \Delta$, so by rule (Trans) on premises 2, 4, 5 and 6:

$$\begin{array}{l}\Delta' \Vdash v^d[\bar{v}/\bar{h}] : \mathrm{IsSum}\ \tau'_e \\ \left(\begin{array}{l}\Delta' \Vdash \bar{v}_j[\bar{v}/\bar{h}] : K_j \in \tau'_e\,?\,\Delta_j \\ \Delta' \Vdash w_j[\bar{v}/\bar{h}] : K_j \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \\ \Delta' \Vdash K_j \in \tau'_e\,?\,\tau''_j \sim \tau'\end{array}\right)_{j\in B}\end{array} \tag{B.15}$$

Also by inductive hypothesis on the first premise:

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'[\bar{v}/\bar{h}] : \tau'_e \tag{B.16}$$

Then, applying rule (RT-DCASE-2) to B.15, B.16 and premise 3, we can conclude that

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \left(\begin{array}{c} \mathbf{protocase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^d[\bar{v}/\bar{h}]\ \mathbf{of} \\ (K_j^{w_j[\bar{v}/\bar{h}]}\ x'_j\ \rightarrow\ e'_j[\bar{v}_j[\bar{v}/\bar{h}]/\bar{h}_j])_{j\in B} \end{array} \right) : \tau'$$

as we wanted to show.

Case (RT-POLYCONSTR): We have a derivation of the form

$$\frac{\begin{array}{c} \bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_j \\ \bar{h} : \Delta \Vdash \mathrm{IsPolySum}\ \tau' \qquad \bar{h} : \Delta \Vdash v_j : \mathrm{HasPolyC}\ \tau'\ \ L_j\ \ \tau'_j \end{array}}{\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} L_j^{v_j}\ e' : \tau'}$$

We need to prove that

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} (L_j^{v_j}\ e')[\bar{v}/\bar{h}] : \tau'$$

By the definition of substitution on a tagged expression, this is the same as proving that

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} (L_j^{v_j[\bar{v}/\bar{h}]}\ e'[\bar{v}/\bar{h}]) : \tau'$$

Now one of our hypothesis is $\Delta' \Vdash \bar{v} : \Delta$, so by rule (Trans) on the second and third premises:

$$\begin{array}{l} \Delta' \Vdash \mathrm{IsPolySum}\ \tau' \\ \Delta' \Vdash v_j[\bar{v}/\bar{h}] : \mathrm{HasPolyC}\ \tau'\ \ L_j\ \ \tau'_j \end{array} \tag{B.17}$$

Also by inductive hypothesis on the first premise:

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'[\bar{v}/\bar{h}] : \tau'_j \tag{B.18}$$

Applying rule (RT-POLYCONSTR) on B.17 and B.18, we can conclude

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} (L_j^{v_j[\bar{v}/\bar{h}]}\ e'[\bar{v}/\bar{h}]) : \tau'$$

as we wanted to show.

Case (RT-POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l}\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_e \\ \bar{h} : \Delta \Vdash v^y : \mathrm{IsPolySum}\ \tau'_e \\ \left(\begin{array}{l}\bar{h}_k : \Delta_k \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e''_k : \sigma_k \\ \bar{h} : \Delta \Vdash \bar{v}_k : L_k \in \tau'_e\,?\,\Delta_k \\ h : \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'\end{array}\right)_{k\in B}\end{array}}{\begin{array}{l}h : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}}\ \ \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and} \\ \qquad\qquad (w_k)_{k\in B}\ \ \mathbf{of}\ (L_k\ \rightarrow\ e''_k[\bar{v}_k/\bar{h}_k])_{k\in B} : \tau'\end{array}}$$

We need to prove that

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \left(\begin{array}{l}\mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and} \\ \quad (w_k)_{k\in B}\ \ \mathbf{of}\ (L_k\ \rightarrow\ e''_k[\bar{v}_k/\bar{h}_k])_{k\in B}\end{array}\right)[\bar{v}/\bar{h}] : \tau'$$

By the definition of substitution on a $\mathbf{polycase}_v$ expression, this is the same as proving that

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \left(\begin{array}{l}\mathbf{polycase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^y[\bar{v}/\bar{h}]\ \mathbf{and} \\ \quad \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B}\ \ \mathbf{of}\ (L_k\ \rightarrow\ e''_k[\bar{v}_k/\bar{h}_k][\bar{v}/\bar{h}])_{k\in B}\end{array}\right) : \tau'$$

We can also assume (by alpha conversion) that $\bar{h}_k$ and $\bar{h}$ are disjoint for all k. Since by lemma 4.8 on the third premise, $EV(e''_k) \subseteq \bar{h}_k$, evidence $\bar{h}$ do not appear free in e''_k and the judgment above is equivalent to

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \left(\begin{array}{l}\mathbf{polycase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^y[\bar{v}/\bar{h}]\ \mathbf{and} \\ \quad \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B}\ \ \mathbf{of}\ (L_k\ \rightarrow\ e''_k[\bar{v}_k[\bar{v}/\bar{h}]/\bar{h}_k])_{k\in B}\end{array}\right) : \tau'$$

Now one of our hypothesis is $\Delta' \Vdash \bar{v} : \Delta$, so by rule (Trans) on premises 2, 4 and 5:

$$\begin{array}{l}\Delta' \Vdash v^y[\bar{v}/\bar{h}] : \mathrm{IsPolySum}\ \tau'_e \\ \left(\begin{array}{l}\Delta' \Vdash \bar{v}_k[\bar{v}/\bar{h}] : L_k \in \tau'_e\,?\,\Delta_k \\ \Delta' \Vdash w_k[\bar{v}/\bar{h}] : L_k \in \tau'_e\,?\,\mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'\end{array}\right)_{k\in B}\end{array} \tag{B.19}$$

Also by inductive hypothesis on the first premise:

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'[\bar{v}/\bar{h}] : \tau'_e \tag{B.20}$$

Then, applying rule (RT-POLYCASE) to B.19, B.20 and premise 3, we can conclude that

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \left(\begin{array}{l}\mathbf{polycase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^y[\bar{v}/\bar{h}]\ \mathbf{and} \\ \quad \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B}\ \ \mathbf{of}\ (L_k\ \rightarrow\ e''_k[\bar{v}_k[\bar{v}/\bar{h}]/\bar{h}_k])_{k\in B}\end{array}\right) : \tau'$$

as we wanted to show.

B.4 Proof of theorem 4.10, section 4.3

Theorem 4.10 *If* $\bar{h} : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma$, *and* $C : (\bar{h} : \Delta \mid \sigma) \geq (\bar{h}' : \Delta' \mid \sigma')$ *then* $\bar{h}' : \Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} C[e'] : \sigma'$

Proof: The proof of this theorem, presented by Martínez López [2005, 6.12], does not depend on the structure of the RT derivation but on the definition of conversions, so the extensions we have made to the system do not modify it.

B.5 Proof of proposition 4.11, section 4.4

Proposition 4.11 *If* $\Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow \sigma$ *then* $S\,\Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow S\,\sigma$

Proof: By induction on the SR relation.
Extending proofs of propositions by Martínez López [2005, 6.13] and Russo [2004, 3.11].

Case (SR-POLYDATA): We have a derivation of the form

$$\frac{\begin{array}{l}\Delta \Vdash \mathrm{IsPolySum}\ \tau' \\ \left(\begin{array}{l}\Delta_k \vdash_{\mathrm{SR}} Y(L_k) \hookrightarrow \sigma_k \\ \Delta \Vdash L_k \in \tau' \,?\, \Delta_k \\ \Delta \Vdash L_k \in \tau' \,?\, \mathrm{HasMGC}\ \tau'\ L_k\ \sigma_k\end{array}\right)_{L_k \in Y}\end{array}}{\Delta \vdash_{\mathrm{SR}} Y^D \hookrightarrow \tau'}$$

We need to prove that

$$S\,\Delta \vdash_{\mathrm{SR}} Y^D \hookrightarrow S\,\tau'$$

By rule (Close) on all the premises involving entailment:

$$\begin{array}{l}S\,\Delta \Vdash S\,(\mathrm{IsPolySum}\ \tau') \\ \left(\begin{array}{l}S\,\Delta_k \Vdash S\,(\mathrm{IsMG}\ \sigma'_k\ \sigma_k) \\ S\,\Delta \Vdash S\,(L_k \in \tau' \,?\, \Delta_k) \\ S\,\Delta \Vdash S\,(L_k \in \tau' \,?\, \mathrm{HasMGC}\ \tau'\ L_k\ \sigma_k)\end{array}\right)_{L_k \in Y}\end{array}$$

Which is equivalent to

$$\begin{array}{l}S\,\Delta \Vdash \mathrm{IsPolySum}\ (S\,\tau') \\ \left(\begin{array}{l}S\,\Delta_k \Vdash \mathrm{IsMG}\ (S\,\sigma'_k)\ (S\,\sigma_k) \\ S\,\Delta \Vdash L_k \in (S\,\tau') \,?\, (S\,\Delta_k) \\ S\,\Delta \Vdash L_k \in (S\,\tau') \,?\, \mathrm{HasMGC}\ (S\,\tau')\ L_k\ (S\,\sigma_k)\end{array}\right)_{L_k \in Y}\end{array} \tag{B.21}$$

Also by inductive hypothesis on the SR premise:

$$\left(S\,\Delta_k \vdash_{\mathrm{SR}} Y(L_k) \hookrightarrow S\,\sigma'_k\right)_{L_k \in Y} \tag{B.22}$$

From B.21 and B.22 by rule (SR-POLYDATA) we can conclude

$$S\,\Delta \vdash_{\text{SR}} Y^D \;\hookrightarrow\; S\,\tau'$$

as we wanted to prove.

B.6 Proof of proposition 4.12, section 4.4

Proposition 4.12 *If* $\Delta \vdash_{\text{SR}} \tau \;\hookrightarrow\; \sigma$ *and* $\Delta' \Vdash \Delta$ *then* $\Delta' \vdash_{\text{SR}} \tau \;\hookrightarrow\; \sigma$

Proof: By induction on the SR derivation.
Extending proofs of propositions by Martínez López [2005, 6.14] and Russo [2004, 3.12].

Case (SR-POLYDATA): We have a derivation of the form

$$\frac{\begin{array}{l}\Delta \Vdash \text{IsPolySum}\;\tau' \\ \left(\begin{array}{l}\Delta_k \vdash_{\text{SR}} Y(L_k) \;\hookrightarrow\; \sigma_k \\ \Delta \Vdash L_k \in \tau' \,?\, \Delta_k \\ \Delta \Vdash L_k \in \tau' \,?\, \text{HasMGC}\;\tau'\;\; L_k\;\; \sigma_k\end{array}\right)_{L_k \in Y}\end{array}}{\Delta \vdash_{\text{SR}} Y^D \;\hookrightarrow\; \tau'}$$

We need to show that

$$\Delta' \vdash_{\text{SR}} Y^D \;\hookrightarrow\; \tau'$$

By hypothesis, $\Delta' \Vdash \Delta$, so by rule (Trans) on all the entailment premises involving Δ

$$\begin{array}{l}\Delta' \Vdash \text{IsPolySum}\;\tau' \\ \left(\begin{array}{l}\Delta' \Vdash L_k \in \tau' \,?\, \Delta_k \\ \Delta' \Vdash L_k \in \tau' \,?\, \text{HasMGC}\;\tau'\;\; L_k\;\; \sigma_k\end{array}\right)_{L_k \in Y}\end{array} \tag{B.23}$$

From B.23 and the remaining premises, we can conclude by rule (SR-POLYDATA)

$$\Delta' \vdash_{\text{SR}} Y^D \;\hookrightarrow\; \tau'$$

As we wanted.

B.7 Proof of theorem 4.13, section 4.4

Theorem 4.13 *If* $\Delta \vdash_{\text{SR}} \tau \;\hookrightarrow\; \sigma$ *and* $C : (\bar{h} : \Delta \mid \sigma) \geq (\bar{h}' : \Delta' \mid \sigma')$ *then* $\Delta' \vdash_{\text{SR}} \tau \;\hookrightarrow\; \sigma'$

Proof: The proof of this theorem, presented by Martínez López [2005, 6.15], does not depend on the structure of the SR derivation but on the definition of conversions, so the extensions we have made to the system do not modify it.

B.8 Proof of lemma 4.14, section 4.4

Lemma 4.14 *If* $\Delta \vdash_{\text{SR}} Y^D \hookrightarrow \sigma$
then $\sigma = \forall\bar{\beta}.\Delta' \Rightarrow \tau$ *and*
$\Delta, \Delta' \Vdash \text{IsPolySum } \tau$

Proof: By induction on the SR derivation.

Case (SR-POLYDATA): We have a derivation of the form

$$\frac{\Delta \Vdash \text{IsPolySum } \tau' \quad \left(\begin{array}{l} \Delta_k \vdash_{\text{SR}} Y(L_k) \hookrightarrow \sigma_k \\ \Delta \Vdash L_k \in \tau' \,?\, \Delta_k \\ \Delta \Vdash L_k \in \tau' \,?\, \text{HasMGC } \tau' \;\; L_k \;\; \sigma_k \end{array}\right)_{L_k \in Y}}{\Delta \vdash_{\text{SR}} Y^D \hookrightarrow \tau'}$$

We need to show that $\tau' = \forall\bar{\beta}.\Delta' \Rightarrow \tau$ such that $\Delta, \Delta' \Vdash \text{IsPolySum } \tau$. This holds taking, $\beta, \Delta' = \emptyset$ and $\tau' = \tau$; $\Delta \Vdash \text{IsPolySum } \tau'$ holds by the first premise of the rule.

Case (SR-QIN): We have a derivation of the form

$$\frac{\Delta, \delta \vdash_{\text{SR}} \tau \hookrightarrow \rho}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \delta \Rightarrow \rho}$$

Taking $\bar{\beta} = \emptyset$, $\rho = \Delta'' \Rightarrow \tau$, we have $\Delta' = \delta, \Delta''$. By inductive hypothesis,

$$\Delta, \delta, \Delta'' \Vdash \text{IsPolySum } \tau$$

so $\Delta, \Delta' \Vdash \text{IsPolySum } \tau$ follows trivially.

Case (SR-QOUT): We have a derivation of the form

$$\frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \delta \Rightarrow \rho \quad \Delta \Vdash \delta}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \rho}$$

Let us take $\bar{\beta} = \emptyset$, $\rho = \Delta' \Rightarrow \tau$. We need to show that $\Delta, \Delta' \Vdash \text{IsPolySum } \tau$. Now by inductive hypothesis on the first premise, $\Delta, \delta, \Delta' \Vdash \text{IsPolySum } \tau$, and since $\Delta \Vdash \delta$, by rule (Cut) we can conclude $\Delta, \Delta' \Vdash \text{IsPolySum } \tau$.

Case (SR-GEN): We have a derivation of the form

$$\frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \forall\alpha.\sigma}\ (\alpha \notin FV(\Delta))$$

We need to show that $\forall\alpha.\sigma = \forall\bar{\beta}.\Delta' \Rightarrow \tau$ such that $\Delta, \Delta' \Vdash \text{IsPolySum } \tau$.

Let us take $\sigma = \forall\bar{\alpha}'.\Delta' \Rightarrow \tau$, that is $\bar{\beta} = \alpha, \bar{\alpha}'$. $\Delta, \Delta' \Vdash \text{IsPolySum } \tau$ follows directly from the inductive hypothesis.

Case (SR-INST): We have a derivation of the form

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

We want to prove that $S\,\sigma = \forall\bar{\beta}.\Delta' \Rightarrow \tau$ such that $\Delta, \Delta' \Vdash \text{IsPolySum } \tau$.

Let us take $\forall\alpha.\sigma = \forall\alpha, \bar{\alpha}'.\Delta' \Rightarrow \tau$, so $S\,\sigma = \forall\bar{\alpha}'.S\,\Delta' \Rightarrow S\,\tau$. Then we need to prove that

$$\Delta, S\,\Delta' \Vdash \text{IsPolySum } S\,\tau$$

By inductive hypothesis,

$$\Delta, \Delta' \Vdash \text{IsPolySum } \tau$$

Applying rule (Close),

$$S\,\Delta, S\,\Delta' \Vdash \text{IsPolySum } S\,\tau$$

Now $\text{dom}(S) = \alpha$, and by alpha-conversion we can assume $\alpha \notin FV(\Delta)$, so $S\,\Delta = \Delta$ and

$$\Delta, S\,\Delta' \Vdash \text{IsPolySum } S\,\tau$$

as we needed to show.

Given our hypothesis $\Delta \vdash_{\text{SR}} Y^D \ \hookrightarrow \ \sigma$, no other SR rules apply, so this completes our proof.

B.9 Proof of theorem 4.20, section 4.4

Theorem 4.20 *If* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \ \hookrightarrow \ e' : \sigma$
then $\Delta \mid \Gamma_{(\text{RT})} \vdash_{\text{RT}} e' : \sigma$
where $\Gamma_{(\text{RT})} = \{x'_i : \tau_i \mid i = 1, \ldots, n\}$
if $\Gamma = \{x_i : \tau_i \hookrightarrow x'_i : \tau_i \mid i = 1, \ldots, n\}$

Proof: By induction on the P derivation.
Extending proofs of theorems by Martínez López [2005, 6.20] and Russo [2004, 3.15].

Case (DCASE-2): We have a derivation of the form

$$\frac{\begin{array}{l}1)\ \Delta \mid \Gamma \vdash_{\mathrm{P}} e : D^D\ \hookrightarrow\ e' : \tau'_e \\ 2)\ \Delta \Vdash v^d : \mathrm{IsSum}\ \tau'_e \\ 3)\ \Delta \vdash_{\mathrm{SR}} \tau\ \hookrightarrow\ \tau' \\ \left(\begin{array}{l} 4)\ \bar{h}_j : \Delta_j \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_j.e_j : D^D\,(K_j) \to^D \tau\ \hookrightarrow\ \lambda x'_j.e'_j : \tau'_j \to \tau''_j \\ 5)\ \Delta \Vdash \bar{v}_j : K_j \in \tau'_e\,?\,\Delta_j \\ 6)\ \Delta \Vdash w_j : K_j \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \\ 7)\ \Delta \Vdash K_j \in \tau'_e\,?\,\tau''_j \sim \tau' \end{array}\right)_{j\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^D\, e\ \mathbf{of}\ (K_j^{\ D}\ x_j\ \to\ e_j)_{j\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{protocase}_v\ e'\ \mathbf{with}\ v^d\ \mathbf{of} \\ \qquad\qquad\quad (K_j^{\ w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : \tau'\end{array}}$$

We want to show that

$$\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} \mathbf{protocase}_v\ e'\ \mathbf{with}\ v^d\ \mathbf{of}\ (K_j^{\ w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : \tau'$$

By inductive hypothesis on premises 1 and 4

$$\begin{array}{l}\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} e' : \tau'_e \\ \left(\bar{h}_j : \Delta_j \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} \lambda x'_j.e'_j : \tau'_j \to \tau''_j\right)_{j\in B}\end{array} \tag{B.24}$$

From B.24 and premises 2, 5, 6 and 7 by (RT-DCASE-2)

$$\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} \mathbf{protocase}_v\ e'\ \mathbf{with}\ v^d\ \mathbf{of}\ (K_j^{\ w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : \tau'$$

As we needed to show.

Case (POLYCONSTR): We have a derivation of the form

$$\frac{\begin{array}{l}1)\ \Delta \vdash_{\mathrm{SR}} Y^D\ \hookrightarrow\ \tau'_e \\ 2)\ \Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y(L_j)\ \hookrightarrow\ e' : \tau'_j \\ 3)\ \Delta \Vdash v_j : \mathrm{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j\end{array}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} L_j^{\ D}\ e : Y^D\ \hookrightarrow\ L_j^{\ v_j}\ e' : \tau'_e}$$

We want to show that

$$\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} L_j^{\ v_j}\ e' : \tau'_e$$

Premise 1 says that $\Delta \vdash_{\mathrm{SR}} Y^D\ \hookrightarrow\ \tau'_e$, so by lemma 4.14 we can conclude

$$\Delta \Vdash \mathrm{IsPolySum}\ \tau'_e \tag{B.25}$$

Also by inductive hypothesis on premise 1

$$\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} e' : \tau'_j \tag{B.26}$$

From B.25, B.26 and premise 3 of the derivation rule, applying (RT-POLYCONSTR)

$$\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} L_j^{\ v_j}\ e' : \tau'_e$$

As we needed to show.

Case (POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l}1)\ \Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y^{D} \ \hookrightarrow\ e' : \tau'_e \\ 2)\ \Delta \Vdash v^{y} : \mathrm{IsPolySum}\ \tau'_e \\ 3)\ \Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \tau' \\ \left(\begin{array}{l} 4)\ \bar{h}_k : \Delta_k \mid \Gamma \vdash_{\mathrm{P}} \lambda^{D} x_k.e_k : Y(L_k) \to^{D} \tau \ \hookrightarrow\ e'_k : \sigma_k \\ 5)\ \Delta \Vdash \bar{v}_k : L_k \in \tau'_e\,?\,\Delta_k \\ 6)\ \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau' \end{array}\right)_{k\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \to\ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^{y}\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ \left(L_k\ \to\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B} : \tau'\end{array}}$$

We want to show that

$$\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^{y}\ \mathbf{and}\ (w_k)_{k\in B}\ \mathbf{of}\ (L_k\ \to\ e'_k[\bar{v}_k/\bar{h}_k])_{k\in B} : \tau'$$

By inductive hypothesis on premises 1 and 4

$$\begin{array}{l}\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} e' : \tau'_e \\ \left(\bar{h}_k : \Delta_k \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} e'_k : \sigma_k\right)_{k\in B}\end{array} \tag{B.27}$$

From B.27 and premises 2, 5, 6 and 7 by (RT-POLYCASE)

$$\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^{y}\ \mathbf{and}\ (w_k)_{k\in B}\ \mathbf{of}\ (L_k\ \to\ e'_k[\bar{v}_k/\bar{h}_k])_{k\in B} : \tau'$$

As we needed to show.

B.10 Proof of theorem 4.21, section 4.4

Theorem 4.21 *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \ \hookrightarrow\ e' : \sigma$
then $\Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \sigma$

Proof: By induction on the P derivation.
Extending proofs of theorems by Martínez López [2005, 6.19] and Russo [2004, 3.14].

Case (RT-DCASE-2): We have a derivation of the form

$$\frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} e : D^{D} \ \hookrightarrow\ e' : \tau'_e \\ \Delta \Vdash v^{d} : \mathrm{IsSum}\ \tau'_e \\ \Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \tau' \\ \left(\begin{array}{l} \bar{h}_j : \Delta_j \mid \Gamma \vdash_{\mathrm{P}} \lambda^{D} x_j.e_j : D^{D}\,(K_j) \to^{D} \tau \ \hookrightarrow\ \lambda x'_j.e'_j : \tau'_j \to \tau''_j \\ \Delta \Vdash \bar{v}_j : K_j \in \tau'_e\,?\,\Delta_j \\ \Delta \Vdash w_j : K_j \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \\ \Delta \Vdash K_j \in \tau'_e\,?\,\tau''_j \sim \tau' \end{array}\right)_{j\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^{D}\, e\ \mathbf{of}\ (K_j^{\ D}\ x_j\ \to\ e_j)_{j\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{protocase}_v\ e'\ \mathbf{with}\ v^{d}\ \mathbf{of} \\ \qquad\qquad\quad (K_j^{\ w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : \tau'\end{array}}$$

We want to prove that

$$\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'$$

which is trivial, since it is one of the premises of the rule.

Case (POLYCONSTR): We have a derivation of the form

$$\frac{\Delta \vdash_{\text{SR}} Y^D \hookrightarrow \tau'_e \qquad \Delta \mid \Gamma \vdash_{\text{P}} e : Y(L_j) \hookrightarrow e' : \tau'_j \qquad \Delta \Vdash v_j : \text{HasPolyC}\ \tau'_e\ L_j\ \tau'_j}{\Delta \mid \Gamma \vdash_{\text{P}} L_j^D\ e : Y^D \hookrightarrow L_j^{v_j}\ e' : \tau'_e}$$

We want to prove that

$$\Delta \vdash_{\text{SR}} Y^D \hookrightarrow \tau'_e$$

which is trivial, since it is one of the premises of the rule.

Case (POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\text{P}} e : Y^D \hookrightarrow e' : \tau'_e \\ \Delta \Vdash v^y : \text{IsPolySum}\ \tau'_e \\ \Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau' \\ \left(\begin{array}{l}\bar{h}_k : \Delta_k \mid \Gamma \vdash_{\text{P}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \hookrightarrow e'_k : \sigma_k \\ \Delta \Vdash \bar{v}_k : L_k \in \tau'_e\,?\,\Delta_k \\ \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\text{HasMGBr}\ \tau'_e\ L_k\ \sigma_k\ \tau'\end{array}\right)_{k\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\text{P}} \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^D\ x_k \to e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ \left(L_k \to e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B} : \tau'\end{array}}$$

We want to prove that

$$\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'$$

which is trivial, since it is one of the premises of the rule.

B.11 Proof of proposition 4.22, section 4.4

Proposition 4.22 ⋆ *If* $\bar{h} : \Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma'$ *and* $\Delta' \Vdash \bar{v} : \Delta$ *then* $\Delta' \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e'[\bar{v}/\bar{h}] : \sigma'$

Proof: By induction on the P derivation.
Extending proofs of propositions by Martínez López [2005, 6.21] and Russo [2004, 3.16].

Case (DCASE-2): We have a derivation of the form

$$\frac{\begin{array}{l}1)\ \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : D^D \ \hookrightarrow\ e' : \tau'_e \\ 2)\ \bar{h} : \Delta \Vdash v^d : \mathrm{IsSum}\ \tau'_e \\ 3)\ \bar{h} : \Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \tau' \\ \left(\begin{array}{l} 4)\ \bar{h}_j : \Delta_j \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_j.e_j : D^D\,(K_j) \to^D \tau \ \hookrightarrow\ \lambda x'_j.e'_j : \tau'_j \to \tau''_j \\ 5)\ \bar{h} : \Delta \Vdash \bar{v}_j : K_j \in \tau'_e\,?\,\Delta_j \\ 6)\ \bar{h} : \Delta \Vdash w_j : K_j \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \\ 7)\ \bar{h} : \Delta \Vdash K_j \in \tau'_e\,?\,\tau''_j \sim \tau' \end{array}\right)_{j\in B}\end{array}}{\begin{array}{l}\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^D\, e\ \mathbf{of}\ (K_j^{\ D}\ x_j\ \to\ e_j)_{j\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{protocase}_v\ e'\ \mathbf{with}\ v^d\ \mathbf{of} \\ \qquad\qquad\quad (K_j^{\ w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : \tau'\end{array}}$$

We need to prove that

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^D\, e\ \mathbf{of}\ (K_j^{\ D}\ x_j\ \to\ e_j)_{j\in B} : \tau\ \hookrightarrow \left(\begin{array}{l}\mathbf{protocase}_v\ e'\ \mathbf{with}\ v^d\ \mathbf{of} \\ \quad (K_j^{\ w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B}\end{array}\right)[\bar{v}/\bar{h}] : \tau'$$

Which, by definition of substitutions on a $\mathbf{protocase}_v$ expression, is equivalent to proving

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^D\, e\ \mathbf{of}\ (K_j^{\ D}\ x_j\ \to\ e_j)_{j\in B} : \tau\ \hookrightarrow \begin{array}{l}\mathbf{protocase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^d[\bar{v}/\bar{h}]\ \mathbf{of} \\ \quad (K_j^{\ w_j[\bar{v}/\bar{h}]}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j][\bar{v}/\bar{h}])_{j\in B}\end{array} : \tau'$$

By alpha conversion we can assume $\bar{h}$ and $\bar{h}_j$ are disjoint. By lemma 4.24 on premise 4, $EV\left(e'_j\right) = EV\left(\lambda x'_j.e'_j\right) \subseteq \bar{h}_j$, so $\bar{h}$ do not appear free in e'_j. Therefore $e'_j[\bar{v}_j/\bar{h}_j][\bar{v}/\bar{h}] = e'_j[\bar{v}_j[\bar{v}/\bar{h}]/\bar{h}_j]$, so we finally need to prove that

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^D\, e\ \mathbf{of}\ (K_j^{\ D}\ x_j\ \to\ e_j)_{j\in B} : \tau\ \hookrightarrow \begin{array}{l}\mathbf{protocase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^d[\bar{v}/\bar{h}]\ \mathbf{of} \\ \quad (K_j^{\ w_j[\bar{v}/\bar{h}]}\ x'_j\ \to\ e'_j[\bar{v}_j[\bar{v}/\bar{h}]/\bar{h}_j])_{j\in B}\end{array} : \tau'$$

Now, as $\Delta' \Vdash \bar{v} : \Delta$, by rule (Trans) on premises 2, 5, 6 and 7

$$\begin{array}{l}\Delta' \Vdash v^d[\bar{v}/\bar{h}] : \mathrm{IsSum}\ \tau'_e \\ \left(\begin{array}{l}\Delta' \Vdash \bar{v}_j[\bar{v}/\bar{h}] : K_j \in \tau'_e\,?\,\Delta_j \\ \Delta' \Vdash w_j[\bar{v}/\bar{h}] : K_j \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \\ \Delta' \Vdash K_j \in \tau'_e\,?\,\tau''_j \sim \tau'\end{array}\right)_{j\in B}\end{array} \tag{B.28}$$

Also by inductive hypothesis on premise 1

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} e : D^D\ \hookrightarrow\ e'[\bar{v}/\bar{h}] : \tau'_e \tag{B.29}$$

Finally, by proposition 4.12 on premise 3

$$\Delta' \vdash_{\mathrm{SR}} \tau\ \hookrightarrow\ \tau' \tag{B.30}$$

From B.28, B.29, B.30 and premise 4, we can conclude by (DCASE-2)

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} \begin{array}{l} \mathbf{case}^D\, e\ \mathbf{of}\ (K_j^{\ D}\ x_j\ \rightarrow\ e_j)_{j\in B} : \tau\ \hookrightarrow \\ \quad \mathbf{protocase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^d[\bar{v}/\bar{h}]\ \mathbf{of} \\ \quad (K_j^{\ w_j[\bar{v}/\bar{h}]}\ x'_j\ \rightarrow\ e'_j[\bar{v}_j[\bar{v}/\bar{h}]/\bar{h}_j])_{j\in B} \end{array} : \tau'$$

As we wanted to show.

Case (POLYCONSTR): We have a derivation of the form

$$\frac{\begin{array}{l} 1)\ \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y(L_j)\ \hookrightarrow\ e' : \tau'_j \\ 2)\ \bar{h} : \Delta \vdash_{\mathrm{SR}} Y^D\ \hookrightarrow\ \tau'_e \\ 3)\ \bar{h} : \Delta \Vdash v_j : \mathrm{HasPolyC}\ \tau'_e\ L_j\ \tau'_j \end{array}}{\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} L_j^{\ D}\ e : Y^D\ \hookrightarrow\ L_j^{\ v_j}\ e' : \tau'_e}$$

We want to prove that

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} L_j^{\ D}\ e : Y^D\ \hookrightarrow\ (L_j^{\ v_j}\ e')[\bar{v}/\bar{h}] : \tau'$$

which, by definition of substitution on tagged expressions is equivalent to

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} L_j^{\ D}\ e : Y^D\ \hookrightarrow\ L_j^{\ v_j[\bar{v}/\bar{h}]}\ e'[\bar{v}/\bar{h}] : \tau'$$

Now by inductive hypothesis on premise 1

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} e : Y(L_j)\ \hookrightarrow\ e'[\bar{v}/\bar{h}] : \tau'_j \tag{B.31}$$

Also, as $\Delta' \Vdash \bar{v} : \Delta$, by proposition 4.12 on premise 2

$$\Delta' \vdash_{\mathrm{SR}} Y^D\ \hookrightarrow\ \tau'_e \tag{B.32}$$

Finally by (Trans) on premise 3

$$\Delta' \Vdash v_j[\bar{v}/\bar{h}] : \mathrm{HasPolyC}\ \tau'_e\ L_j\ \tau'_j \tag{B.33}$$

From B.31, B.32 and B.33 by rule (POLYCONSTR)

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} L_j^{\ D}\ e : Y^D\ \hookrightarrow\ L_j^{\ v_j[\bar{v}/\bar{h}]}\ e'[\bar{v}/\bar{h}] : \tau'$$

As we needed to prove.

Case (POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l} 1)\ \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y^D\ \hookrightarrow\ e' : \tau'_e \\ 2)\ \bar{h} : \Delta \Vdash v^y : \mathrm{IsPolySum}\ \tau'_e \\ 3)\ \bar{h} : \Delta \vdash_{\mathrm{SR}} \tau\ \hookrightarrow\ \tau' \\ \left(\begin{array}{l} 4)\ \bar{h}_k : \Delta_k \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_k.e_k : Y(L_k) \rightarrow^D \tau\ \hookrightarrow\ e'_k : \sigma_k \\ 5)\ \bar{h} : \Delta \Vdash \bar{v}_k : L_k \in \tau'_e\,?\,\Delta_k \\ 6)\ \bar{h} : \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\mathrm{HasMGBr}\ \tau'_e\ L_k\ \sigma_k\ \tau' \end{array}\right)_{k\in B} \end{array}}{\begin{array}{l} \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \qquad \hookrightarrow \\ \qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad \mathbf{of}\ \left(L_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B} : \tau' \end{array}}$$

We need to prove that

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau\ \hookrightarrow \left(\begin{array}{c}\mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B}\ \mathbf{of}\\ \left(L_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B}\end{array}\right)[\bar{v}/\bar{h}] : \tau'$$

Which, by definition of substitutions on a $\mathbf{polycase}_v$ expression, is equivalent to proving

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau\ \hookrightarrow \begin{array}{l}\mathbf{polycase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^y[\bar{v}/\bar{h}]\ \mathbf{and}\ \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B}\ \mathbf{of}\\ \left(L_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k][\bar{v}/\bar{h}]\right)_{k\in B}\end{array} : \tau'$$

By alpha conversion we can assume $\bar{h}$ and $\bar{h}_k$ are disjoint. By lemma 4.24 on premise 4, $EV(e'_k) \subseteq \bar{h}_k$, so $\bar{h}$ do not appear free in e'_k. Therefore $e'_k[\bar{v}_k/\bar{h}_k][\bar{v}/\bar{h}] = e'_k[\bar{v}_k[\bar{v}/\bar{h}]/\bar{h}_k]$, so we finally need to prove that

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau\ \hookrightarrow \begin{array}{l}\mathbf{polycase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^y[\bar{v}/\bar{h}]\ \mathbf{and}\ \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B}\ \mathbf{of}\\ \left(L_k\ \rightarrow\ e'_k[\bar{v}_k[\bar{v}/\bar{h}]/\bar{h}_k]\right)_{k\in B}\end{array} : \tau'$$

Now, as $\Delta' \Vdash \bar{v} : \Delta$, by rule (Trans) on premises 2, 5 and 6

$$\begin{array}{l}\Delta' \Vdash v^y[\bar{v}/\bar{h}] : \mathrm{IsPolySum}\ \tau'_e\\ \left(\begin{array}{l}\Delta' \Vdash \bar{v}_k[\bar{v}/\bar{h}] : L_k \in \tau'_e\,?\,\Delta_k\\ \Delta' \Vdash w_k[\bar{v}/\bar{h}] : L_k \in \tau'_e\,?\,\mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'\end{array}\right)_{k\in B}\end{array} \tag{B.34}$$

Also by inductive hypothesis on premise 1

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} e : Y^D\ \hookrightarrow\ e'[\bar{v}/\bar{h}] : \tau'_e \tag{B.35}$$

Finally, by proposition 4.12 on premise 3

$$\Delta' \vdash_{\mathrm{SR}} \tau\ \hookrightarrow\ \tau' \tag{B.36}$$

From B.34, B.35, B.36 and premise 4, we can conclude by (POLYCASE)

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau\ \hookrightarrow \begin{array}{l}\mathbf{polycase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^y[\bar{v}/\bar{h}]\ \mathbf{and}\ \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B}\ \mathbf{of}\\ \left(L_k\ \rightarrow\ e'_k[\bar{v}_k[\bar{v}/\bar{h}]/\bar{h}_k]\right)_{k\in B}\end{array} : \tau'$$

As we wanted to show.

B.12 Proof of proposition 4.23, section 4.4

Proposition 4.23 ⋆ *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau\ \hookrightarrow\ e' : \sigma$
then $S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} e : \tau\ \hookrightarrow\ e' : S\,\sigma$

Proof: By induction on the P derivation.
Extending proofs of propositions by Martínez López [2005, 6.22] and Russo [2004, 3.17].

Case (DCASE-2): We have a derivation of the form

$$\frac{\begin{array}{l}1)\ \Delta \mid \Gamma \vdash_{\mathrm{P}} e : D^{D} \ \hookrightarrow\ e' : \tau'_e \\ 2)\ \Delta \Vdash v^{d} : \mathrm{IsSum}\ \tau'_e \\ 3)\ \Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \tau' \\ \left(\begin{array}{l} 4)\ \bar{h}_j : \Delta_j \mid \Gamma \vdash_{\mathrm{P}} \lambda^{D} x_j.e_j : D^{D}\,(K_j) \to^{D} \tau \ \hookrightarrow\ \lambda x'_j.e'_j : \tau'_j \to \tau''_j \\ 5)\ \Delta \Vdash \bar{v}_j : K_j \in \tau'_e\,?\,\Delta_j \\ 6)\ \Delta \Vdash w_j : K_j \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \\ 7)\ \Delta \Vdash K_j \in \tau'_e\,?\,\tau''_j \sim \tau' \end{array}\right)_{j\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^{D}\, e\ \mathbf{of}\ (K_j^{\,D}\ x_j\ \to\ e_j)_{j\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{protocase}_{v}\ e'\ \mathbf{with}\ v^{d}\ \mathbf{of} \\ \qquad\qquad\quad (K_j^{\,w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : \tau'\end{array}}$$

We want to see that

$$\begin{array}{l}S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^{D}\, e\ \mathbf{of}\ (K_j^{\,D}\ x_j\ \to\ e_j)_{j\in B} : \tau \\ \qquad\qquad\qquad \hookrightarrow \\ \qquad\qquad\qquad \mathbf{protocase}_{v}\ e'\ \mathbf{with}\ v^{d}\ \mathbf{of} \\ \qquad\qquad\qquad\quad (K_j^{\,w_j}\ x'_j\ \to\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : (S\,\tau')\end{array}$$

By inductive hypothesis on premises 1 and 4

$$\begin{array}{l}S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} e : D^{D} \ \hookrightarrow\ e' : S\,\tau'_e \\ \Big(S\,\Delta_j \mid S\,\Gamma \vdash_{\mathrm{P}} \lambda^{D} x_j.e_j : D^{D}\,(K_j) \to^{D} \tau \ \hookrightarrow\ \lambda x'_j.e'_j : S\,(\tau'_j \to \tau''_j)\Big)_{j\in B}\end{array}$$

and since $S\,(\tau'_j \to \tau''_j) = S\,\tau' \to S\,\tau''_j$,

$$\begin{array}{l}S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} e : Y^{D} \ \hookrightarrow\ e' : S\,\tau'_e \\ \Big(S\,\Delta_j \mid S\,\Gamma \vdash_{\mathrm{P}} \lambda^{D} x_j.e_j : D^{D}\,(K_j) \to^{D} \tau \ \hookrightarrow\ \lambda x'_j.e'_j : S\,\tau' \to S\,\tau''_j\Big)_{j\in B}\end{array} \tag{B.37}$$

Also by proposition 4.11 on premise 3

$$S\,\Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ S\,\tau' \tag{B.38}$$

Finally by rule (Close) on premises 2, 5, 6 and 7

$$\begin{array}{l}S\,\Delta' \Vdash v^{d} : \mathrm{IsSum}\ (S\,\tau'_e) \\ \left(\begin{array}{l} S\,\Delta' \Vdash v_j : K_j \in (S\,\tau'_e)\,?\,(S\,\Delta_j) \\ S\,\Delta' \Vdash w_j : K_j \in (S\,\tau'_e)\,?\,\mathrm{HasC}\ (S\,\tau'_e)\ \ K_j\ \ (S\,\tau'_j) \\ S\,\Delta' \Vdash K_j \in (S\,\tau'_e)\,?\,S\,\tau''_j \sim S\,\tau' \end{array}\right)_{j\in B}\end{array} \tag{B.39}$$

From B.37, B.38 and B.39 by rule (DCASE-2)

$$\begin{array}{l}S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^{D}\, e\ \mathbf{of}\ (K_j^{\,D}\ x_j\ \to\ e_j)_{j\in B} : \tau \\ \qquad\qquad\qquad \hookrightarrow \\ \qquad\qquad\qquad \mathbf{protocase}_{v}\ e'\ \mathbf{with}\ v^{d}\ \mathbf{of} \\ \qquad\qquad\qquad\quad (K_j^{\,w_j}\ x'_j\ \to\ e'_j[v_j/h_j])_{j\in B} : (S\,\tau')\end{array}$$

As we wanted to show.

Case (POLYCONSTR): We have a derivation of the form

$$\frac{\begin{array}{l}1)\ \Delta \vdash_{\mathrm{SR}} Y^D \ \hookrightarrow\ \tau'_e \\ 2)\ \Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y(L_j) \ \hookrightarrow\ e' : \tau'_j \\ 3)\ \Delta \Vdash v_j : \mathrm{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j\end{array}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} L_j^D\ e : Y^D \ \hookrightarrow\ L_j^{v_j}\ e' : \tau'_e}$$

We want to prove that

$$S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} L_j^D\ e : Y^D \ \hookrightarrow\ L_j^{v_j}\ e' : S\,\tau'_e$$

By inductive hypothesis on premise 1

$$S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} e : Y(L_j) \ \hookrightarrow\ e' : S\,\tau'_j \tag{B.40}$$

By proposition 4.11 on premise 2

$$S\,\Delta \vdash_{\mathrm{SR}} Y^D \ \hookrightarrow\ S\,\tau'_e \tag{B.41}$$

By rule (Close) on premise 3

$$S\,\Delta \Vdash v_j : S\,(\mathrm{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j)$$

which, by definition of type substitution, is equivalent to

$$S\,\Delta \Vdash v_j : \mathrm{HasPolyC}\ (S\,\tau'_e)\ \ L_j\ \ (S\,\tau'_j) \tag{B.42}$$

From B.40, B.41 and B.42, by (POLYCONSTR)

$$S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} L_j^D\ e : Y^D \ \hookrightarrow\ L_j^{v_j}\ e' : S\,\tau'_e$$

As we wanted to show.

Case (POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l}1)\ \Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y^D \ \hookrightarrow\ e' : \tau'_e \\ 2)\ \Delta \Vdash v^y : \mathrm{IsPolySum}\ \tau'_e \\ 3)\ \Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \tau' \\ \left(\begin{array}{l}4)\ \bar{h}_k : \Delta_k \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow\ e'_k : \sigma_k \\ 5)\ \Delta \Vdash \bar{v}_k : L_k \in \tau'_e\,?\,\Delta_k \\ 6)\ \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'\end{array}\right)_{k\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^D\ \ x_k\ \ \to\ \ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\quad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad\quad \mathbf{of}\ \left(L_k\ \to\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B} : \tau'\end{array}}$$

We want to see that

$$\begin{array}{l} S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B}\ \mathbf{of} \\ \qquad\qquad\quad \big(L_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\big)_{k\in B} : (S\,\tau') \end{array}$$

Now by inductive hypothesis on premises 1 and 4

$$\begin{array}{l} S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} e : Y^{D}\ \hookrightarrow\ e' : S\,\tau'_e \\ \big(\bar{h}_k : S\,\Delta_k \mid S\,\Gamma \vdash_{\mathrm{P}} \lambda^{D} x_k.e_k : Y(L_k) \rightarrow^{D} \tau\ \hookrightarrow\ e'_k : S\,\sigma_k\big)_{k\in B} \end{array} \tag{B.43}$$

By proposition 4.11 on premise 3

$$S\,\Delta \vdash_{\mathrm{SR}} \tau\ \hookrightarrow\ S\,\tau' \tag{B.44}$$

Finally by rule (Close) on premises 2, 5 and 6

$$\begin{array}{l} S\,\Delta \Vdash v^y : \mathrm{IsPolySum}\ (S\,\tau'_e) \\ \left(\begin{array}{l} S\,\Delta \Vdash v_k : L_k \in (S\,\tau'_e)\,?\,(S\,\Delta_k) \\ S\,\Delta \Vdash w_k : L_k \in (S\,\tau'_e)\,?\,\mathrm{HasMGBr}\ (S\,\tau'_e)\ \ L_k\ \ (S\,\sigma_k)\ \ (S\,\tau') \end{array} \right)_{k\in B} \end{array} \tag{B.45}$$

From B.43, B.44 and B.45 by rule (POLYCASE)

$$\begin{array}{l} S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B}\ \mathbf{of} \\ \qquad\qquad\quad \big(L_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\big)_{k\in B} : (S\,\tau') \end{array}$$

as we wanted to show.

B.13 Proof of lemma 4.24, section 4.4

Lemma 4.24 *If* $\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau\ \hookrightarrow\ e' : \sigma$ *then* $EV(e') \subseteq \bar{h}$

Proof: By induction on the P derivation.

Extending proofs of lemmas by Martínez López [2005, 6.23] and Russo [2004, 3.18].

Case (DCASE-2): We have a derivation of the form

$$\dfrac{\begin{array}{l} 1)\ \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : D^{D}\ \hookrightarrow\ e' : \tau'_e \\ 2)\ \bar{h} : \Delta \Vdash v^{d} : \mathrm{IsSum}\ \tau'_e \\ 3)\ \bar{h} : \Delta \vdash_{\mathrm{SR}} \tau\ \hookrightarrow\ \tau' \\ \left(\begin{array}{l} 4)\ \bar{h}_j : \Delta_j \mid \Gamma \vdash_{\mathrm{P}} \lambda^{D} x_j.e_j : D^{D}\,(K_j) \rightarrow^{D} \tau\ \hookrightarrow\ \lambda x'_j.e'_j : \tau'_j \rightarrow \tau''_j \\ 5)\ \bar{h} : \Delta \Vdash \bar{v}_j : K_j \in \tau'_e\,?\,\Delta_j \\ 6)\ \bar{h} : \Delta \Vdash w_j : K_j \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j \\ 7)\ \bar{h} : \Delta \Vdash K_j \in \tau'_e\,?\,\tau''_j \sim \tau' \end{array} \right)_{j\in B} \end{array}}{\begin{array}{l} \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^{D}\, e\ \mathbf{of}\ (K_j^{\ D}\ x_j\ \rightarrow\ e_j)_{j\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{protocase}_v\ e'\ \mathbf{with}\ v^{d}\ \mathbf{of} \\ \qquad\qquad\quad (K_j^{\ w_j}\ x'_j\ \rightarrow\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : \tau' \end{array}}$$

Now by inductive hypothesis on premises 1 and 4

$$EV\left(e'\right) \subseteq \bar{h} \tag{B.46}$$

$$EV\left(e'_j\right) \subseteq \bar{h}_j \ \forall j \in B \tag{B.47}$$

By rule (Evars) on premises 2, 5 and 6

$$EV\left(v^d\right) \subseteq \bar{h} \tag{B.48}$$

$$EV(\bar{v}_j) \subseteq \bar{h} \ \forall j \in B \tag{B.49}$$

$$EV(w_j) \subseteq \bar{h} \ \forall j \in B \tag{B.50}$$

From B.47 we can conclude that $EV\left(e'_j[\bar{v}_j/\bar{h}_j]\right) = EV(\bar{v}_j)$ by substitution. Then, from B.49,

$$EV\left(e'_j[\bar{v}_j/\bar{h}_j]\right) = EV(\bar{v}_j) \subseteq \bar{h} \tag{B.51}$$

Finally, from B.46, B.48, B.50 and B.51 we can conclude

$$EV\left(\begin{array}{l} \textbf{protocase}_v\ e'\ \textbf{with}\ v^d\ \textbf{of} \\ \quad (K_j^{\,w_j}\ x'_j\ \rightarrow\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} \end{array}\right) \quad =$$

$$EV(e') \cup EV\left(v^d\right) \cup \bigcup_{j\in B}\left(EV(w_j) \cup EV\left(e'_j[\bar{v}_j/\bar{h}_j]\right)\right) \ \subseteq \ \bar{h}$$

As we wanted to prove.

Case (POLYCONSTR): We have a derivation of the form

$$\frac{\begin{array}{c}\bar{h} : \Delta \vdash_{\text{SR}} Y^D\ \hookrightarrow\ \tau'_e \\ \bar{h} : \Delta \mid \Gamma \vdash_{\text{P}} e : Y(L_j)\ \hookrightarrow\ e' : \tau'_j \qquad \bar{h} : \Delta \Vdash v_j : \text{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j\end{array}}{\bar{h} : \Delta \mid \Gamma \vdash_{\text{P}} L_j^D\ e : Y^D\ \hookrightarrow\ L_j^{v_j}\ e' : \tau'_e}$$

By inductive hypothesis on the first premise $EV(e') \subseteq \bar{h}$, and by rule (Evars) on the third premise $EV(v_j) \subseteq \bar{h}$. Therefore,

$$EV\left(L_j^{v_j}\ e\right) = EV(v_j) \cup EV\left(e'\right) \cup \bar{h}$$

As we wanted to show.

Case (POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l} 1)\ \bar{h} : \Delta \mid \Gamma \vdash_{\text{P}} e : Y^D\ \hookrightarrow\ e' : \tau'_e \\ 2)\ \bar{h} : \Delta \Vdash v^y : \text{IsPolySum}\ \tau'_e \\ 3)\ \bar{h} : \Delta \vdash_{\text{SR}} \tau\ \hookrightarrow\ \tau' \\ \left(\begin{array}{l} 4)\ \bar{h}_k : \Delta_k \mid \Gamma \vdash_{\text{P}} \lambda^D x_k.e_k : Y(L_k) \rightarrow^D \tau\ \hookrightarrow\ e'_k : \sigma_k \\ 5)\ \bar{h} : \Delta \Vdash \bar{v}_k : L_k \in \tau'_e\,?\,\Delta_k \\ 6)\ \bar{h} : \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\text{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau' \end{array}\right)_{k\in B} \end{array}}{\begin{array}{l} \bar{h} : \Delta \mid \Gamma \vdash_{\text{P}}\ \textbf{case}^D\ e\ \textbf{of}\ (L_k^D\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \textbf{polycase}_v\ e'\ \textbf{with}\ v^y\ \textbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad\qquad\qquad \textbf{of}\ \left(L_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B} : \tau' \end{array}}$$

Now by inductive hypothesis on premises 1 and 4

$$EV\left(e'\right) \subseteq \bar{h} \tag{B.52}$$
$$EV\left(e'_k\right) \subseteq \bar{h}_k \ \forall k \in B \tag{B.53}$$

By rule (Evars) on premises 2, 5 and 6

$$EV(v^y) \subseteq \bar{h} \tag{B.54}$$
$$EV(\bar{v}_k) \subseteq \bar{h} \ \forall k \in B \tag{B.55}$$
$$EV(w_k) \subseteq \bar{h} \ \forall k \in B \tag{B.56}$$

From B.53 we can conclude that $EV\left(e'_k[\bar{v}_k/\bar{h}_k]\right) = EV(\bar{v}_k)$ by substitution. Then, from B.55,

$$EV\left(e'_k[\bar{v}_k/\bar{h}_k]\right) = EV(\bar{v}_k) \subseteq \bar{h} \tag{B.57}$$

Finally, from B.52, B.54, B.56 and B.57 we can conclude

$$EV\left(\begin{array}{c}\textbf{polycase}_v\ e'\ \textbf{with}\ v^y\ \textbf{and}\ (w_k)_{k\in B}\ \textbf{of}\\ \left(L_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B}\end{array}\right) \quad =$$

$$EV(e') \cup EV(v^y) \cup \bigcup_{k\in B}\left(EV(w_k) \cup EV\left(e'_k[\bar{v}_k/\bar{h}_k]\right)\right) \subseteq \bar{h}$$

as we wanted to prove.

B.14 Proof of lemma 4.25, section 4.4

Lemma 4.25 *If* $\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$
then there exist $\bar{\beta}$, Δ_σ *and* τ'' *such that* $\sigma = \forall\bar{\beta}.\Delta_\sigma \Rightarrow \tau''$

Proof: By induction on the P derivation.
Extending proofs of lemmas by Martínez López [2005, 6.24] and Russo [2004, 3.19].

Case (DCASE-2): We have a derivation of the form

$$\frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} e : D^D \hookrightarrow e' : \tau'_e\\ \Delta \Vdash v^d : \mathrm{IsSum}\ \tau'_e\\ \Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow \tau'\\ \left(\begin{array}{l}\bar{h}_j : \Delta_j \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_j.e_j : D^D\,(K_j) \rightarrow^D \tau \hookrightarrow \lambda x'_j.e'_j : \tau'_j \rightarrow \tau''_j\\ \Delta \Vdash \bar{v}_j : K_j \in \tau'_e\,?\,\Delta_j\\ \Delta \Vdash w_j : K_j \in \tau'_e\,?\,\mathrm{HasC}\ \tau'_e\ \ K_j\ \ \tau'_j\\ \Delta \Vdash K_j \in \tau'_e\,?\,\tau''_j \sim \tau'\end{array}\right)_{j\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}}\ \textbf{case}^D\, e\ \textbf{of}\ (K_j^{\ D}\ x_j\ \rightarrow\ e_j)_{j\in B} : \tau\\ \qquad\hookrightarrow\\ \qquad\textbf{protocase}_v\ e'\ \textbf{with}\ v^d\ \textbf{of}\\ \qquad\quad (K_j^{\ w_j}\ x'_j\ \rightarrow\ e'_j[\bar{v}_j/\bar{h}_j])_{j\in B} : \tau'\end{array}}$$

To show $\bar{\beta}$, Δ_σ and τ'' such that

$$\tau' = \forall \bar{\beta}.\Delta_\sigma \Rightarrow \tau''$$

it suffices to take $\bar{\beta} = \emptyset$, $\Delta_\sigma = \emptyset$ and $\tau'' = \tau'$.

Case (POLYCONSTR): We have a derivation of the form

$$\frac{\begin{array}{c}\Delta \vdash_{\mathrm{SR}} Y^D \ \hookrightarrow\ \tau'_e \\ \Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y(L_j) \ \hookrightarrow\ e' : \tau'_j \qquad \Delta \Vdash v_j : \mathrm{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j\end{array}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} L_j^D\ e : Y^D \ \hookrightarrow\ L_j^{v_j}\ e' : \tau'_e}$$

To show $\bar{\beta}$, Δ_σ and τ'' such that

$$\tau'_e = \forall \bar{\beta}.\Delta_\sigma \Rightarrow \tau''$$

it suffices to take $\bar{\beta} = \emptyset$, $\Delta_\sigma = \emptyset$ and $\tau'' = \tau'_e$.

Case (POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y^D \ \hookrightarrow\ e' : \tau'_e \\ \Delta \Vdash v^y : \mathrm{IsPolySum}\ \tau'_e \\ \Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \tau' \\ \left(\begin{array}{l}\bar{h}_k : \Delta_k \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow\ e'_k : \sigma_k \\ \Delta \Vdash \bar{v}_k : L_k \in \tau'_e\,?\,\Delta_k \\ \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'\end{array}\right)_{k \in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^D\ \ x_k\ \ \to\ \ e_k)_{k \in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k \in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ \left(L_k\ \ \to\ \ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k \in B} : \tau'\end{array}}$$

To show $\bar{\beta}$, Δ_σ and τ'' such that

$$\tau' = \forall \bar{\beta}.\Delta_\sigma \Rightarrow \tau''$$

it suffices to take $\bar{\beta} = \emptyset$, $\Delta_\sigma = \emptyset$ and $\tau'' = \tau'$.

B.15 Proof of proposition 5.1, section 5.1

Proposition 5.1 *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow\ e' : \tau'$ *then* $h : S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow\ e' : S\,\tau'$

Proof: By induction on the S derivation.
Extending proofs of propositions by Martínez López [2005, 7.7] and Russo [2004, 4.3].

Case (S-POLYCONSTR): The proof is analogous to that of proposition 4.23 for case (POLYCONSTR).

Case (S-POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{S}} e : Y^{D} \;\hookrightarrow\; e' : \tau'_e \\ \Delta \Vdash v^{y} : \mathrm{IsPolySum}\ \tau'_e \\ \Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; \tau' \\ \left(\begin{array}{l} \bar{h}_k : \Delta_k \mid \Gamma \vdash_{\mathrm{S}} \lambda^{D} x_k.e_k : Y(L_k) \to^{D} \tau \;\hookrightarrow\; e''_k : \tau'_k \\ \sigma_k = \mathrm{Gen}_{\Gamma}(\Delta_k \Rightarrow \tau'_k), \qquad e'_k = \Lambda \bar{h}_k.e''_k \\ \Delta \Vdash w_k : L_k \in \tau'_e \,?\, \mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau' \end{array}\right)_{k \in B}\end{array}}{\begin{array}{r}\Delta \mid \Gamma \vdash_{\mathrm{S}} \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\,D}\ x_k\ \to\ e_k)_{k \in B} : \tau \\ \hookrightarrow \\ \mathbf{polycase}_{v}\ e'\ \mathbf{with}\ v^{y}\ \mathbf{and}\ (w_k)_{k \in B} \\ \mathbf{of}\ (L_k\ \to\ e'_k)_{k \in B} : \tau'\end{array}}$$

We want to see that

$$\begin{array}{r}S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\,D}\ x_k\ \to\ e_k)_{k \in B} : \tau \\ \hookrightarrow \\ \mathbf{polycase}_{v}\ e'\ \mathbf{with}\ v^{y}\ \mathbf{and}\ (w_k)_{k \in B} \\ \mathbf{of}\ (L_k\ \to\ e'_k)_{k \in B} : S\,\tau'\end{array}$$

By inductive hypothesis on the first premise

$$S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : Y^{D} \;\hookrightarrow\; e' : S\,\tau'_e \tag{B.58}$$

By proposition 4.11 on the third premise

$$S\,\Delta \vdash_{\mathrm{SR}} \tau \;\hookrightarrow\; S\,\tau' \tag{B.59}$$

By rule (Close) on the second and last premises

$$\begin{array}{l} S\,\Delta \Vdash v^{y} : \mathrm{IsPolySum}\ (S\,\tau'_e) \\ \big(S\,\Delta \Vdash w_k : L_k \in (S\,\tau'_e)\,?\,\mathrm{HasMGBr}\ (S\,\tau'_e)\ \ L_k\ \ (S\,\sigma_k)\ \ (S\,\tau')\big)_{k \in B}\end{array} \tag{B.60}$$

where $\sigma_k = \mathrm{Gen}_{\Gamma}(\Delta_k \Rightarrow \tau'_k)$

Now for each k, by proposition 2.16-3, there is a substitution T_k such that $T_k\,\Gamma = S\,\Gamma$ and $\mathrm{Gen}_{S\,\Gamma}(T_k\,\Delta_k \Rightarrow T_k\,\tau'_k) = S\,\mathrm{Gen}_{\Gamma}(\Delta_k \Rightarrow \tau'_k)$. So if we call $\sigma'_k = \mathrm{Gen}_{T_k\,\Gamma}(T_k\,\Delta_k \Rightarrow T_k\,\tau'_k)$, we have $\sigma'_k = S\,\sigma_k$ and

$$\big(S\,\Delta \Vdash w_k : L_k \in (S\,\tau'_e)\,?\,\mathrm{HasMGBr}\ (S\,\tau'_e)\ \ L_k\ \ \sigma'_k\ \ (S\,\tau')\big)_{k \in B} \tag{B.61}$$

Finally, by inductive hypothesis on each k, and knowing $T_k\,\Gamma = S\,\Gamma$

$$h_k : T_k\,\Delta_k \mid S\,\Gamma \vdash_{\mathrm{S}} \lambda^{D} x_k.e_k : Y(L_k) \to^{D} \tau \;\hookrightarrow\; e''_k : T_k\,\tau'_k \tag{B.62}$$

By rule (S-POLYCASE) on B.58, B.60, B.59, B.62, and B.61 we can conclude

$$S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} \begin{array}{l} \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\,D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \hookrightarrow \\ \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad \mathbf{of}\ (L_k\ \rightarrow\ e'_k)_{k\in B} : S\,\tau' \end{array}$$

as we wanted to show.

B.16 Proof of proposition 5.2, section 5.1

Proposition 5.2 *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow\ e' : \tau'$ *and* $\Delta' \Vdash v : \Delta$ *then* $\Delta' \mid \Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow\ e'[h/v] : \tau'$

Proof: By induction on the P derivation.
Extending proofs of propositions by Martínez López [2005, 7.8] and Russo [2004, 4.4].

Case (S-POLYCONSTR): The proof is analogous to that of proposition 4.22 for case (POLYCONSTR).

Case (S-POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l} \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{S}} e : Y^{D} \ \hookrightarrow\ e' : \tau'_e \\ \bar{h} : \Delta \Vdash v^y : \mathrm{IsPolySum}\ \tau'_e \\ \bar{h} : \Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \tau' \\ \left(\begin{array}{l} \bar{h}_k : \Delta_k \mid \Gamma \vdash_{\mathrm{S}} \lambda^{D} x_k.e_k : Y(L_k) \rightarrow^{D} \tau \ \hookrightarrow\ e''_k : \tau'_k \\ \sigma_k = \mathrm{Gen}_{\Gamma}(\Delta_k \Rightarrow \tau'_k), \qquad e'_k = \Lambda \bar{h}_k.e''_k \\ \bar{h} : \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau' \end{array}\right)_{k\in B} \end{array}}{\begin{array}{l} \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{S}} \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\,D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \hookrightarrow \\ \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad \mathbf{of}\ (L_k\ \rightarrow\ e'_k)_{k\in B} : \tau' \end{array}}$$

We want to prove that

$$\Delta' \mid \Gamma \vdash_{\mathrm{S}} \begin{array}{l} \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\,D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \hookrightarrow \\ \left(\mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B}\ \mathbf{of}\ (L_k\ \rightarrow\ e'_k)_{k\in B}\right)[\bar{v}/\bar{h}] : \tau' \end{array}$$

which, by evidence substitution, is equivalent to proving

$$\Delta' \mid \Gamma \vdash_{\mathrm{S}} \begin{array}{l} \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\,D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \hookrightarrow \\ \mathbf{polycase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^y[\bar{v}/\bar{h}]\ \mathbf{and}\ \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B} \\ \qquad\qquad \mathbf{of}\ \left(L_k\ \rightarrow\ e'_k[\bar{v}/\bar{h}]\right)_{k\in B} : \tau' \end{array}$$

By alpha conversion we can assume $\bar{h}$ and $\bar{h}_k$ are disjoint. Moreover, by lemma 4.24, $EV(e''_k) \subseteq \bar{h}_k$, so $\bar{h}$ do neither appear free in e''_k nor in e'_k. Therefore $e'_k[\bar{v}/\bar{h}] = e'_k$, and we finally need to prove that

$$\begin{array}{l} \Delta' \mid \Gamma \vdash_{\mathrm{S}} \ \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^y[\bar{v}/\bar{h}]\ \mathbf{and}\ \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ (L_k\ \rightarrow\ e'_k)_{k\in B} : \tau' \end{array}$$

Now, as $\Delta' \Vdash \bar{v} : \Delta$, by rule (Trans) on the second and last premises

$$\begin{array}{l} \Delta' \Vdash v^y[\bar{v}/\bar{h}] : \mathrm{IsPolySum}\ \tau'_e \\ \left(\ \Delta' \Vdash w_k[\bar{v}/\bar{h}] : L_k \in \tau'_e\, ?\, \mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'\ \right)_{k\in B} \end{array} \tag{B.63}$$

Also by inductive hypothesis on the first premise

$$\Delta' \mid \Gamma \vdash_{\mathrm{S}} e : Y^D\ \hookrightarrow\ e'[\bar{v}/\bar{h}] : \tau'_e \tag{B.64}$$

Finally, by proposition 4.12 on the third premise

$$\Delta' \vdash_{\mathrm{SR}} \tau\ \hookrightarrow\ \tau' \tag{B.65}$$

Applying rule (S-POLYCASE), from B.64, B.63, B.65 and the fourth premise we can conclude

$$\begin{array}{l} \Delta' \mid \Gamma \vdash_{\mathrm{S}} \ \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'[\bar{v}/\bar{h}]\ \mathbf{with}\ v^y[\bar{v}/\bar{h}]\ \mathbf{and}\ \left(w_k[\bar{v}/\bar{h}]\right)_{k\in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ (L_k\ \rightarrow\ e'_k)_{k\in B} : \tau' \end{array}$$

as we needed to prove.

B.17 Proof of theorem 5.3, section 5.1

Theorem 5.3 ⋆ *If* $\Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau\ \hookrightarrow\ e' : \tau'$ *then* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau\ \hookrightarrow\ e' : \tau'$

Proof: By induction on the S derivation.
Extending proofs of propositions by Martínez López [2005, 7.9] and Russo [2004, 4.5].

Case (S-POLYCONSTR): We have a derivation of the form

$$\frac{\begin{array}{c} \Delta \vdash_{\mathrm{SR}} Y^D\ \hookrightarrow\ \tau'_e \\ \Delta \mid \Gamma \vdash_{\mathrm{S}} e : Y(L_j)\ \hookrightarrow\ e' : \tau'_j \qquad \Delta \Vdash v_j : \mathrm{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j \end{array}}{\Delta \mid \Gamma \vdash_{\mathrm{S}} L_j^{\ D}\ e : Y^D\ \hookrightarrow\ L_j^{\ v_j}\ e' : \tau'_e}$$

We need to prove that

$$\Delta \mid \Gamma \vdash_{\mathrm{P}} L_j^{\ D}\ e : Y^D \ \hookrightarrow\ L_j^{\ v_j}\ e' : \tau'_e$$

By inductive hypothesis on the third premise

$$\Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y(L_j) \ \hookrightarrow\ e' : \tau'_j \tag{B.66}$$

The result follows from applying rule (POLYCONSTR) on the first and third premises, and B.66.

Case (S-POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{S}} e : Y^D \ \hookrightarrow\ e' : \tau'_e \\ \Delta \Vdash v^y : \mathrm{IsPolySum}\ \tau'_e \\ \Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \tau' \\ \left(\begin{array}{l}\bar{h}_k : \Delta_k \mid \Gamma \vdash_{\mathrm{S}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow\ e''_k : \tau'_k \\ \sigma_k = \mathrm{Gen}_\Gamma(\Delta_k \Rightarrow \tau'_k), \qquad e'_k = \Lambda \bar{h}_k.e''_k \\ \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'\end{array}\right)_{k\in B}\end{array}}{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{S}}\ \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \to\ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ (L_k\ \to\ e'_k)_{k\in B} : \tau'\end{array}}$$

We want to see that

$$\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}}\ \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \to\ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ (L_k\ \to\ e'_k)_{k\in B} : \tau'\end{array}$$

By inductive hypothesis on the first premise

$$\Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y^D \ \hookrightarrow\ e' : \tau'_e \tag{B.67}$$

Now for each branch k, also by inductive hypothesis

$$\bar{h}_k : \Delta_k \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow\ e''_k : \tau'_k$$

Applying rule (QIN) as many times as necessary

$$\emptyset \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow\ \Lambda \bar{h}_k.e''_k : \Delta_k \Rightarrow \tau'_k$$

Applying rule (GEN) as many times as necessary

$$\emptyset \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow\ \Lambda \bar{h}_k.e''_k : \mathrm{Gen}_\Gamma(\Delta_k \Rightarrow \tau'_k)$$

that is

$$\emptyset \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow\ e'_k : \sigma_k \tag{B.68}$$

By B.67, B.68 and the remaining premises, we can apply rule (POLYCASE) to conclude

$$\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^{D}\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ (L_k\ \rightarrow\ e'_k)_{k\in B} : \tau'\end{array}$$

as we wanted to show.

B.18 Proof of theorem 5.4, section 5.1

Theorem 5.4 ⋆ *If* $\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \ \hookrightarrow\ e' : \sigma$
then there exist $\bar{h}'_s$, Δ'_s, e'_s, τ'_s *and* C'_s *such that*
$\bar{h}'_s : \Delta'_s \mid \Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow\ e'_s : \tau'_s$,
$C'_s : \mathrm{Gen}_\Gamma(\Delta'_s \Rightarrow \tau'_s) \geq (\bar{h} : \Delta \mid \sigma)$,
$C'_s[\Lambda h'_s.e'_s] = e'$

Proof: By induction on the P derivation.
Extending proofs of propositions by Martínez López [2005, 7.10] and Russo [2004, 4.6].

Case (POLYCONSTR): We have a derivation of the form

$$\frac{\bar{h} : \Delta \vdash_{\mathrm{SR}} Y^{D}\ \hookrightarrow\ \tau'_e \qquad \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y(L_j)\ \hookrightarrow\ e' : \tau'_j \qquad \bar{h} : \Delta \Vdash v_j : \mathrm{HasPolyC}\ \tau'_e\ L_j\ \tau'_j}{\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} L_j^{\ D}\ e : Y^{D}\ \hookrightarrow\ L_j^{\ v_j}\ e' : \tau'_e}$$

By inductive hypothesis on the second premise, we know there exist $\bar{h}^e_s$, Δ^e_s, e^e_s, τ^e_s and C^e_s such that

$$\bar{h}^e_s : \Delta^e_s \mid \Gamma \vdash_{\mathrm{S}} e : Y(L_j)\ \hookrightarrow\ e^e_s : \tau^e_s \tag{B.69}$$
$$C^e_s : \mathrm{Gen}_\Gamma(\Delta^e_s \Rightarrow \tau^e_s) \geq (\bar{h} : \Delta \mid \tau'_j) \tag{B.70}$$
$$C^e_s[\Lambda \bar{h}^e_s.e^e_s] = e' \tag{B.71}$$

By definition 2.5 on B.70, there must be a substitution S^e (whose variables do not appear free in Γ) and evidence $\bar{v}^e$ such that

$$\tau'_j = S^e\, \tau^e_s \tag{B.72}$$
$$\bar{h} : \Delta \Vdash \bar{v}^e : S^e\, \Delta^e_s \tag{B.73}$$
$$C^e_s = \mathbf{let}_v\ x = []\ \mathbf{in}\ x((\bar{v}^e)) \tag{B.74}$$

Let us take $\bar{h}'_s = \bar{h}$, $\Delta'_s = \Delta$, $e'_s = L_j^{\ v_j}\ e^e_s[\bar{v}_e/\bar{h}^e_s]$, $\tau'_s = \tau'_e$ and $C'_s = [](\bar{h})$. It suffices to verify the following properties

1. $\bar{h}'_s : \Delta'_s \mid \Gamma \vdash_{\mathrm{S}} L_j^{\ D}\ e : Y^{D}\ \hookrightarrow\ e'_s : \tau'_s$
   From B.69, by proposition 5.1, and considering $S^e\, \Gamma = \Gamma$ since the variables in S^e do not appear in Γ

$$\bar{h}^e_s : S^e\, \Delta^e_s \mid \Gamma \vdash_{\mathrm{S}} e : Y(L_j)\ \hookrightarrow\ e^e_s : S^e\, \tau^e_s$$

From B.72 and B.73, by proposition 5.2

$$\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{S}} e : Y(L_j) \ \hookrightarrow \ e_s^e[\bar{v}^e/\bar{h}_s^e] : \tau'_j$$

Taking this last statement and the two remaining premises in the derivation, we can conclude by rule (S-POLYCONSTR)

$$\bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{S}} L_j^{\ D} \ e : Y^D \ \hookrightarrow \ L_j^{\ v_j} \ e_s^e[\bar{v}^e/\bar{h}_s^e] : \tau'_e$$

And by definition of each construct,

$$\bar{h}'_s : \Delta'_s \mid \Gamma \vdash_{\mathrm{S}} L_j^{\ D} \ e : Y^D \ \hookrightarrow \ e'_s : \tau'_s$$

2. $C'_s : \mathrm{Gen}_\Gamma(\Delta'_s \Rightarrow \tau'_s) \geq (\Delta \mid \tau'_e)$
   Follows directly from rule (Id) and proposition 2.16-1, since $C'_s = [](\bar{h}))$, $\Delta'_s = \Delta$ and $\tau'_s = \tau'_e$.
3. $C'_s[\Lambda\bar{h}'_s.e'_s] = L_j^{\ v_j} \ e'$
   From B.71, B.74 and equivalence of residual terms by reduction

$$e_s^e[\bar{v}^e/\bar{h}_s^e] = (\Lambda\bar{h}_s^e.e_s^e)(\bar{v}^e)) = C_s^e[\Lambda\bar{h}_s^e.e_s^e] = e'$$

So

$$C'_s[\Lambda\bar{h}'_s.e'_s] = (\Lambda\bar{h}.L_j^{\ v_j} \ e_s^e[\bar{v}^e/\bar{h}_s^e])(\bar{h})) = L_j^{\ v_j} \ e_s^e[\bar{v}^e/\bar{h}_s^e] = L_j^{\ v_j} \ e' = e'_s$$

This completes the proof.

Case (POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l} 1)\ \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : Y^D \ \hookrightarrow \ e' : \tau'_e \\ 2)\ \bar{h} : \Delta \Vdash v^y : \mathrm{IsPolySum}\ \tau'_e \\ 3)\ \bar{h} : \Delta \vdash_{\mathrm{SR}} \tau \ \hookrightarrow \ \tau' \\ \left(\begin{array}{l} 4)\ \bar{h}_k : \Delta_k \mid \Gamma \vdash_{\mathrm{P}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow \ e'_k : \sigma_k \\ 5)\ \bar{h} : \Delta \Vdash \bar{v}_k : L_k \in \tau'_e \,?\, \Delta_k \\ 6)\ \bar{h} : \Delta \Vdash w_k : L_k \in \tau'_e \,?\, \mathrm{HasMGBr}\ \tau'_e \ L_k \ \sigma_k \ \tau' \end{array}\right)_{k \in B} \end{array}}{\begin{array}{l} \bar{h} : \Delta \mid \Gamma \vdash_{\mathrm{P}} \ \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^{\ D} \ x_k \ \to \ e_k)_{k \in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k \in B} \\ \qquad\qquad\qquad\qquad \mathbf{of}\ \left(L_k \ \to \ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k \in B} : \tau' \end{array}}$$

Step 1 Let us prove that, for all $k \in B$, σ_k is equal to $\forall\bar{\beta}_{j_k}.\Delta_{\sigma_k} \Rightarrow \tau_{\sigma_k}$ such that

$$\begin{array}{l} \bar{h}_k : \Delta_k, \bar{h}_{\sigma_k} : \Delta_{\sigma_k} \mid \Gamma \vdash_{\mathrm{S}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow \ e''_k : \tau_{\sigma_k} \\ \bar{h} : \Delta \Vdash w'_k : L_k \in \tau'_e \,?\, \mathrm{HasMGBr}\ \tau'_e \ L_k \ \sigma'_k \ \tau' \end{array}$$

where
$w'_k = \mathbf{if}_v \ L_k \in v^y \ \mathbf{then}\ w_k \diamond (\mathbf{if}_v \ L_k \in v^y \ \mathbf{then}\ C'_k \ \mathbf{else}\ \bullet) \ \mathbf{else}\ \bullet$
$C'_k = \Lambda\bar{h}_{\sigma_k}.[](\bar{v}_k, \bar{h}_{\sigma_k}))$
$\sigma'_k = \mathrm{Gen}_\Gamma(\Delta_k, \Delta_{\sigma_k} \Rightarrow \tau_{\sigma_k})$
$e'_k = \Lambda\bar{h}_{\sigma_k}.e''_k$

By inductive hypothesis on premise 4, there exist $\bar{h}_k^S$, Δ_k^S, e_k^S, τ_k^S and C_k^S such that

$$\bar{h}_k^S : \Delta_k^S \mid \Gamma \vdash_{\mathrm{S}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow \ e_k^S : \tau_k^S \tag{B.75}$$
$$C_k^S : \mathrm{Gen}_\Gamma(\Delta_k^S \Rightarrow \tau_k^S) \geq (\Delta_k \mid \sigma_k) \tag{B.76}$$
$$C_k^S[\Lambda \bar{h}_k^S.e_k^S] = e'_k \tag{B.77}$$

By lemma 4.25 on premise 4, we know that

$$\sigma_k = \forall \bar{\beta}_{j_k}.\Delta_{\sigma_k} \Rightarrow \tau_{\sigma_k} \tag{B.78}$$

where $\bar{\beta}_{j_k}$ do not appear free in Γ nor in Δ_k.

By definition 2.5 on B.76 and B.78, there exist a substitution S_k (such that $\mathrm{dom}(S_k) \cap FV(\Gamma) = \emptyset$) and evidence $\bar{v}_{\sigma_k}$ such that

$$\tau_{\sigma_k} = S_k\, \tau_k^S \tag{B.79}$$
$$\bar{h}_k : \Delta_k, \bar{h}_{\sigma_k} : \Delta_{\sigma_k} \Vdash \bar{v}_{\sigma_k} : S_k\, \Delta_k^S \tag{B.80}$$
$$C_k^S = \Lambda \bar{h}_{\sigma_k}.[](\!(\bar{v}_{\sigma_k})\!) \tag{B.81}$$

By proposition 5.1 applied to B.75

$$\bar{h}_k^S : S_k\, \Delta_k^S \mid S_k\, \Gamma \vdash_{\mathrm{S}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow \ e_k^S : S_k\, \tau_k^S$$

Now by proposition 5.2 applied to the statement above, using B.79, B.80 and $S_k\, \Gamma = \Gamma$

$$\bar{h}_k : \Delta_k, \bar{h}_{\sigma_k} : \Delta_{\sigma_k} \mid \Gamma \vdash_{\mathrm{S}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \ \hookrightarrow \ e_k^S[\bar{v}_{\sigma_k}/\bar{h}_k^S] : \tau_{\sigma_k} \tag{B.82}$$

Replacing B.81 in B.77

$$e'_k = \Lambda \bar{h}_{\sigma_k}.(\Lambda \bar{h}_k^S.e_k^S)(\!(\bar{v}_{\sigma_k})\!) = \Lambda \bar{h}_{\sigma_k}.e_k^S[\bar{v}_{\sigma_k}/\bar{h}_k^S] \tag{B.83}$$

Let us call $\sigma'_k = \mathrm{Gen}_\Gamma(\Delta_k, \Delta_{\sigma_k} \Rightarrow \tau_{\sigma_k})$. Since $\bar{\beta}_{j_k}$ do not appear free in Δ_k, every free variable in Δ_k must appear free in σ_k (see B.78). By definition 2.5

$$C_k : (\Delta_k \mid \sigma'_k) \geq (\Delta_k \mid \sigma_k)$$

with $C_k = \Lambda \bar{h}_{\sigma_k}.[](\!(\bar{h}_k, \bar{h}_{\sigma_k})\!)$. So by entailment rule (IsMG)

$$\bar{h}_k : \Delta_k \Vdash C_k : \mathrm{IsMG}\ \sigma'_k\ \sigma_k$$

By lemma 4.6 on premises 1) and 5) and the statement above

$$\bar{h} : \Delta \Vdash \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ C_k[\bar{v}_k/\bar{h}_k]\ \mathbf{else}\ \bullet : L_k \in \tau'_e\,?\,\mathrm{IsMG}\ \sigma'_k\ \sigma$$

so calling $C'_k = C_k[\bar{v}_k/\bar{h}_k] = \Lambda \bar{h}_{\sigma_k}.[](\!(\bar{v}_k, \bar{h}_{\sigma_k})\!)$, we have

$$\bar{h} : \Delta \Vdash \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ C'_k\ \mathbf{else}\ \bullet : L_k \in \tau'_e\,?\,\mathrm{IsMG}\ \sigma'_k\ \sigma \tag{B.84}$$

Let us call $\Delta_{aux} = \{\mathrm{IsMG}\ \sigma'_k\ \sigma_k, \mathrm{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'_k\}$. By B.84 and premise 6)

$$\bar{h} : \Delta \Vdash \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ C'_k\ \mathbf{else}\ \bullet, w_k : L_k \in \tau'_e\,?\,\Delta_{aux}$$

Additionally, by rule (Comp-MGBr),

$$h^1_{aux}, h^2_{aux} : \Delta_{aux} \Vdash h^2_{aux} \diamond h^1_{aux} : \text{HasMGBr } \tau'_e \;\; L_k \;\; \sigma'_k \;\; \tau'$$

So by lemma 4.6 on premise 4 and the two statements above

$$\begin{array}{l} \bar{h} : \Delta \Vdash \;\; \textbf{if}_v \; L_k \in v^y \\ \qquad\qquad \textbf{then } w_k \diamond (\textbf{if}_v \; L_k \in v^y \textbf{ then } C'_k \textbf{ else } \bullet) \\ \qquad\qquad \textbf{else } \bullet \\ \qquad\qquad : \;\; L_k \in \tau'_e \,?\, \text{HasMGBr } \tau'_e \;\; L_k \;\; \sigma'_k \;\; \tau' \end{array} \tag{B.85}$$

Calling $e''_k = e^S_k[\bar{v}_{\sigma_k}/\bar{h}^S_k]$, the proof of step 1 follows from B.78, B.82, B.85 and B.83.

Step 2 Let us now complete the main proof. By inductive hypothesis on premise 1), there exist $\bar{h}^S$, Δ^S, e^S, τ^S and C^S such that

$$\bar{h}^S : \Delta^S \mid \Gamma \vdash_{\text{S}} e : Y^D \;\; \hookrightarrow \;\; e^S : \tau^S \tag{B.86}$$

$$C^S : \text{Gen}_\Gamma(\Delta^S \Rightarrow \tau^S) \geq (\Delta \mid \tau'_e) \tag{B.87}$$

$$C^S[\Lambda \bar{h}^S . e^S] = e' \tag{B.88}$$

By definition 2.5 on B.87, there exist a substitution S (such that $\text{dom}(S) \cap FV(\Gamma) = \emptyset$) and evidence $\bar{v}^S$ such that

$$\tau'_e = S\,\tau^S \tag{B.89}$$

$$\bar{h} : \Delta \Vdash \bar{v}^S : S\,\Delta^S \tag{B.90}$$

$$C^S = [](\!(\bar{v}^S)\!) \tag{B.91}$$

By proposition 5.1 applied to B.86

$$\bar{h}^S : S\,\Delta^S \mid S\,\Gamma \vdash_{\text{S}} e : Y^D \;\; \hookrightarrow \;\; e^S : S\,\tau^S$$

Now by proposition 5.2 applied to the statement above, using B.89, B.90 and $S\,\Gamma = \Gamma$

$$\bar{h} : \Delta \mid \Gamma \vdash_{\text{S}} e : Y^D \;\; \hookrightarrow \;\; e^S[\bar{v}^S/\bar{h}^S] : \tau'_e$$

Replacing B.91 in B.88 we have $e' = (\Lambda \bar{h}^S . e^S)(\!(\bar{v}^S)\!) = e^S[\bar{v}^S/\bar{h}^S]$, so replacing in the statement above

$$\bar{h} : \Delta \mid \Gamma \vdash_{\text{S}} e : Y^D \;\; \hookrightarrow \;\; e' : \tau'_e \tag{B.92}$$

Let us take

$$\begin{array}{l} \bar{h}'_s = \bar{h} \\ \Delta'_s = \Delta \\ e'_s = \textbf{polycase}_v \; e' \textbf{ with } v^y \textbf{ and } (w'_k)_{k \in B} \textbf{ of } \left(L_k \;\rightarrow\; \Lambda \bar{h}_k . e'_k\right)_{k \in B} \\ \tau'_s = \tau' \\ C'_s = [](\!(\bar{h})\!) \end{array}$$

It suffices to verify the following properties

1. $\bar{h}'_s : \Delta'_s \mid \Gamma \vdash_{\mathrm{S}} \mathbf{case}^D\, e\ \mathbf{of}\ \left(L_k^D\ x_k\ \rightarrow\ e_k\right)_{k\in B} : \tau\ \hookrightarrow\ e'_s : \tau'_s$
   Follows from rule (S-POLYCASE) applied to B.92, premises 2) and 3) and step 1.
2. $C'_s : \mathrm{Gen}_\Gamma(\Delta'_s \Rightarrow \tau'_s) \geq (\Delta \mid \tau')$
   Follows directly from rule (Id) and proposition 2.16-1.
3. $C'_s[\Lambda\bar{h}'_s.e'_s] = \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B}\ \mathbf{of}\ \left(L_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B}$
   By definition of C'_s, $C'_s[\Lambda\bar{h}'_s.e'_s] = C'_s[\Lambda\bar{h}.e'_s] = (\Lambda\bar{h}.e'_s)((\bar{h})) = e'_s$. So we must prove

$$\begin{array}{c}\mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B}\ \mathbf{of}\ \left(L_k\ \rightarrow\ e'_k[\bar{v}_k/\bar{h}_k]\right)_{k\in B}\\ =\\ \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w'_k)_{k\in B}\ \mathbf{of}\ \left(L_k\ \rightarrow\ \Lambda\bar{h}_k.e'_k\right)_{k\in B}\end{array}$$

   Without loss of generality by definition of equivalence on a $\mathbf{polycase}_v$ expression (see section A.6), we can assume v^y and w_k are not variables. In particular, v^y is of the form $\{L_{k,i}\}_{k\in I, i\in I'_k}$ and w_k is of the form $\left\langle n, (v_i)_{i\in I'_k}\right\rangle$ for all $k \in I$.
   Let us take any $k \in I$, $w_k = \left\langle n, (v_i)_{i\in I'_k}\right\rangle$. By definition of w'_k

$$w'_k = \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ w_k \diamond (\mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ C'_k\ \mathbf{else}\ \bullet)\ \mathbf{else}\ \bullet$$

   and since $L_k \in v^y$ is true,

$$w'_k = w_k \diamond C'_k = \left\langle n, \left(v_i \circ C'_k\right)_{i\in I'_k}\right\rangle$$

   By the reduction rule for $\mathbf{polycase}_v$ expressions, it suffices to see that

$$v_i[e'_k[\bar{v}_k/\bar{h}_k]] = (v_i \circ C'_k)[\Lambda\bar{h}_k.e'_k]$$

   for all $k \in I, i \in I'_k$ ($k \notin I$ are ignored). Indeed,

$$C'_k[\Lambda\bar{h}_k.e'_k] = C'_k[\Lambda\bar{h}_k, \bar{h}_{\sigma_k}.e''_k] = \Lambda\bar{h}_{\sigma_k}.(\Lambda\bar{h}_k, \bar{h}_{\sigma_k}.e''_k)((\bar{v}_k, \bar{h}_{\sigma_k})) =$$

$$\Lambda\bar{h}_{\sigma_k}.e''_k[\bar{v}_k/\bar{h}_k] = \left(\Lambda\bar{h}_{\sigma_k}.e''_k\right)[\bar{v}_k/\bar{h}_k] = e'_k[\bar{v}_k/\bar{h}_k]$$

   so

$$(v_i \circ C'_k)[\Lambda\bar{h}_k.e'_k] = v_i[C'_k[\Lambda\bar{h}_k.e'_k]] = v_i[e'_k[\bar{v}_k/\bar{h}_k]]$$

This completes our proof.

B.19 Proof of proposition 5.5, section 5.2

Proposition 5.5 *If $\sigma \sim^U \sigma'$ then $U\,\sigma = U\,\sigma'$*

Proof: By induction on the unification structure.
Extending proofs of propositions by Martínez López [2005, 7.11] and Russo [2004, 4.7].

The result follows trivially from inductive hypothesis and the definition of substitution (see A.5).

B.20 Proof of proposition 5.6, section 5.2

Proposition 5.6 *If* $S\,\sigma = S\,\sigma'$
then $\sigma \sim^U \sigma'$ *and there exists a substitution* T *such that* $S = TU$

Proof: The proof of this proposition, presented by Martínez López [2005, 7.12], does not depend on the structure of the unification derivation but only on the free variables in σ and σ', so the extensions we have made to the system do not modify it.

B.21 Proof of proposition 5.8, section 5.2

Proposition 5.8 *If* $\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'$ *then* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'$

Proof: By induction on the W-SR derivation.

Extending proofs of propositions by Martínez López [2005, 7.14] and Russo [2004, 4.10].

Case (WSR-POLYDATA): We have a derivation of the form

$$\frac{\left(\begin{array}{l}\Delta_k \vdash_{\text{W-SR}} Y(L_k) \hookrightarrow \tau'_k \\ \sigma_k = \text{Gen}_{\emptyset,\emptyset}(\Delta_k \Rightarrow \tau'_k)\end{array}\right)_{L_k \in Y}}{\begin{array}{l}\text{IsPolySum } t, \\ (L_k \in t\,?\,\text{HasMGC } t\ \ L_k\ \ \sigma_k)_{L_k \in Y}\end{array} \vdash_{\text{W-SR}} Y^D \hookrightarrow t}\ (t \text{ fresh})$$

Let us call $\Delta = \text{IsPolySum } t, (L_k \in t\,?\,\text{HasMGC } t\ \ L_k\ \ \sigma_k)_{L_k \in Y}$. We need to prove that $\Delta \vdash_{\text{SR}} Y^D \hookrightarrow t$.

By definition of Δ, we know that

$$\Delta \Vdash \text{IsPolySum } t \tag{B.93}$$

Now let us take any $L_k \in Y$. By inductive hypothesis,

$$\Delta_k \vdash_{\text{SR}} Y(L_k) \hookrightarrow \tau'_k$$

Applying rule (SR-QIN) as many times as necessary we can conclude

$$\emptyset \vdash_{\text{SR}} Y(L_k) \hookrightarrow \Delta_k \Rightarrow \tau'_k$$

and by (SR-GEN) as many times as necessary,

$$\emptyset \vdash_{\text{SR}} Y(L_k) \hookrightarrow \text{Gen}_{\emptyset,\emptyset}(\Delta_k \Rightarrow \tau'_k)$$

that is

$$\emptyset \vdash_{\text{SR}} Y(L_k) \hookrightarrow \sigma_k \tag{B.94}$$

Trivially by rule (Term),

$$\Delta \Vdash L_k \in \tau'\,?\,\emptyset \tag{B.95}$$

Finally, by definition of Δ,

$$\Delta \Vdash L_k \in \tau'\,?\,\text{HasMGC } \tau'\ \ L_k\ \ \sigma_k \tag{B.96}$$

From B.93, and for all $L_k \in Y$, B.94, B.95 and B.96, applying rule (SR-POLYDATA) we can conclude

$$\Delta \vdash_{\text{SR}} Y^D \hookrightarrow t$$

As we wanted to show.

B.22 Proof of proposition 5.9, section 5.2

Proposition 5.9 *If* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$
then there exist Δ'_w, τ'_w *and* C'_w *such that*
$\Delta'_w \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'_w$ *with all the residual variables fresh and* $C'_w : \text{Gen}_{\emptyset,\emptyset}(\Delta'_w \Rightarrow \tau'_w) \geq (\Delta \mid \sigma)$

Proof: By induction on the SR derivation.
Extending proofs of propositions by Martínez López [2005, 7.15] and Russo [2004, 4.11].

Case (SR-POLYDATA): We have a derivation of the form

$$\frac{\begin{array}{l}\bar{h} : \Delta \Vdash v^y : \text{IsPolySum } \tau' \\ \left(\begin{array}{l}\Delta_k \vdash_{\text{SR}} Y(L_k) \hookrightarrow \sigma_k \\ \bar{h} : \Delta \Vdash \bar{v}_k^{\Delta} : L_k \in \tau' \,?\, \Delta_k \\ \bar{h} : \Delta \Vdash \bar{v}_k^{mgc} : L_k \in \tau' \,?\, \text{HasMGC } \tau' \;\; L_k \;\; \sigma_k\end{array}\right)_{L_k \in Y}\end{array}}{\Delta \vdash_{\text{SR}} Y^D \hookrightarrow \tau'}$$

We need to show Δ'_w, τ'_w with all the type variables fresh such that

$$\Delta'_w \vdash_{\text{W-SR}} Y^D \hookrightarrow \tau'_w \tag{B.97}$$

We also need to construct C'_w such that $C'_w : \text{Gen}_{\emptyset,\emptyset}(\Delta'_w \Rightarrow \tau'_w) \geq (\Delta \mid \sigma)$. So it suffices to construct a substitution S' and evidence $\bar{v}'$ such that:

$$\tau' = S' \, \tau'_w \tag{B.98}$$
$$\bar{h} : \Delta \Vdash \bar{v}' : S' \, \Delta'_w \tag{B.99}$$

and take $C'_w = [](\!(v')\!)$.

By inductive hypothesis on the second premise, for all $L_k \in Y$, we have Δ'_{w_k}, τ'_{w_k} and C'_{w_k} such that

$$\begin{array}{l}\Delta'_{w_k} \vdash_{\text{W-SR}} Y(L_k) \hookrightarrow \tau'_{w_k} \\ C'_{w_k} : \text{Gen}_{\emptyset,\emptyset}(\Delta'_{w_k} \Rightarrow \tau'_{w_k}) \geq (\Delta_k \mid \sigma_k)\end{array} \tag{B.100}$$

Let us call $\sigma'_{w_k} = \text{Gen}_{\emptyset,\emptyset}(\Delta'_{w_k} \Rightarrow \tau'_{w_k})$, take $\tau'_w = t$ a fresh type variable and $\Delta'_w = \text{IsPolySum } t, \big(L_k \in t \,?\, \text{HasMGC } t \;\; L_k \;\; \sigma'_{w_k}\big)_{L_k \in Y}$.

By rule (WSR-POLYDATA), we have

$$\Delta'_w \vdash_{\text{W-SR}} Y^D \hookrightarrow \tau'_w$$

where all the free variables (namely t) are fresh. So B.97 is verified.

Now let us define S' such that $\text{dom}(S') = \{t\}$ and $S' \, t = \tau'$. This verifies B.98 trivially; now only B.99 remains to be proved, where

$$S'\,\Delta'_w = \begin{array}{l}\text{IsPolySum } \tau',\\ \left(L_k \in \tau' \,?\, \text{HasMGC } \tau'\ \ L_k\ \ \sigma'_{w_k}\right)_{L_k \in Y}\end{array}$$

We define $\bar{v}' = v^y, (v'_k)_{L_k \in Y}$, where each v'_k will be constructed conveniently as shown below. We have that $\bar{h} : \Delta \Vdash v^y : \text{IsPolySum } \tau'$ by the first premise of the SR derivation. We are going to show that, for each $L_k \in Y$, evidence v'_k can be constructed to prove

$$\bar{h} : \Delta \Vdash v'_k : L_k \in \tau' \,?\, \text{HasMGC } \tau'\ \ L_k\ \ \sigma'_{w_k}$$

which is all we need to complete our proof.

Firstly, let us recall that, by our premises in the SR derivation,

$$\bar{h} : \Delta \Vdash v^y : \text{IsPolySum } \tau' \tag{B.101}$$

$$\bar{h} : \Delta \Vdash \bar{v}_k^{\Delta} : L_k \in \tau' \,?\, \Delta_k \tag{B.102}$$

$$\bar{h} : \Delta \Vdash v_k^{mgc} : L_k \in \tau' \,?\, \text{HasMGC } \tau'\ \ L_k\ \ \sigma_k \tag{B.103}$$

Now let us prove we can construct evidence for $\Delta \Vdash L_k \in \tau' \,?\, \text{IsMG } \sigma'_{w_k}\ \sigma_k$.

As we stated in B.100, we know that $C'_{w_k} : (\emptyset \mid \sigma'_{w_k}) \geq (\Delta_k \mid \sigma_k)$. Since $\Delta'_k \Vdash \Delta'_k$ (rule (Id)), by properties 2.7.2 and 2.8.3, we can construct C''_{w_k} such that

$$C''_{w_k} : (\Delta_k \mid \sigma'_{w_k}) \geq (\Delta_k \mid \sigma_k)$$

Then by rule (IsMG)

$$\Delta_k \Vdash C''_{w_k} : \text{IsMG } \sigma'_{w_k}\ \sigma_k \tag{B.104}$$

By lemma 4.6 on B.101, B.102 and B.104, evidence can be constructed for

$$\Delta \Vdash L_k \in \tau' \,?\, \text{IsMG } \sigma'_{w_k}\ \sigma_k \tag{B.105}$$

In addition, by rule (Comp-MGC), we know that

$$\left(\begin{array}{l}\text{HasMGC } \tau'\ \ L_k\ \ \sigma_k\\ \text{IsMG } \sigma'_{w_k}\ \sigma_k\end{array}\right) \Vdash \text{HasMGC } \tau'\ \ L_k\ \ \sigma'_{w_k} \tag{B.106}$$

By B.103 and B.105,

$$\Delta \Vdash L_k \in \tau' \,?\, \left(\begin{array}{l}\text{HasMGC } \tau'\ \ L_k\ \ \sigma_k\\ \text{IsMG } \sigma'_{w_k}\ \sigma_k\end{array}\right) \tag{B.107}$$

So again by lemma 4.6 on B.101, B.106 and B.107 we can construct evidence v'_k to finally conclude

$$\bar{h} : \Delta \Vdash v'_k : L_k \in \tau' \,?\, \text{HasMGC } \tau'\ \ L_k\ \ \sigma'_{w_k}$$

Which completes our proof.

B.23 Proof of lemma 5.10, section 5.2

Lemma 5.10 *If* $\bar{h} : \Delta \mid S\,\Gamma \vdash_{\text{W}} e : \tau \ \hookrightarrow\ e' : \tau'$ *then* $EV(e') \subseteq \bar{h}$

Proof: By induction on the W derivation.

Extending proofs of theorems by Martínez López [2005, 7.16] and Russo [2004, 4.12].

It holds trivially.

B.24 Proof of theorem 5.11, section 5.2

Theorem 5.11 ⋆ *If* $\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \hookrightarrow e' : \tau'$
then $\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau'$

Proof: By induction on the W derivation.
Extending proofs of propositions by Martínez López [2005, 7.17] and Russo [2004, 4.13].

Case (W-POLYCONSTR): We have a derivation of the form

$$\frac{\begin{array}{l}\Delta \vdash_{\text{W-SR}} Y^D \hookrightarrow \tau'_e \\ \Delta' \mid S\,\Gamma \vdash_{\mathrm{W}} e : Y(L_j) \hookrightarrow e' : \tau'_j \\ \Delta'' \mid \Delta, \Delta' \Vdash_{\mathrm{W}} v_j : \mathrm{HasPolyC}\ \tau'_e\ L_j\ \tau'_j\end{array}}{\Delta, \Delta', \Delta'' \mid S\,\Gamma \vdash_{\mathrm{W}} L_j^D\ e : Y^D \hookrightarrow L_j^{v_j}\ e' : \tau'_e}$$

We need to prove that

$$\Delta, \Delta', \Delta'' \mid S\,\Gamma \vdash_{\mathrm{S}} L_j^D\ e : Y^D \hookrightarrow L_j^{v_j}\ e' : \tau'_e$$

By proposition 5.8 on the first premise in combination with (Fst) and proposition 4.12

$$\Delta, \Delta', \Delta'' \vdash_{\mathrm{SR}} Y^D \hookrightarrow \tau'_e \tag{B.108}$$

By inductive hypothesis on the second premise and proposition 5.2 (evidence substitution is trivial)

$$\Delta, \Delta', \Delta'' \mid S\,\Gamma \vdash_{\mathrm{S}} e : Y(L_j) \hookrightarrow e' : \tau'_j \tag{B.109}$$

By proposition 5.7 on the third premise

$$\Delta, \Delta', \Delta'' \Vdash \mathrm{HasPolyC}\ \tau'_e\ L_j\ \tau'_j \tag{B.110}$$

The result follows fom rule (S-POLYCONSTR) applied to B.108, B.109 and B.110.

Case (W-POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l}\Delta_e \mid S_e\,\Gamma \vdash_{\mathrm{W}} e : Y^D \hookrightarrow e' : \tau'_e \\ \Delta_p \mid \Delta_e \Vdash_{\mathrm{W}} v^y : \mathrm{IsPolySum}\ \tau'_e \\ \Delta_{SR} \vdash_{\text{W-SR}} \tau \hookrightarrow \tau' \\ \left(h_k : \Delta_k \mid S_k\,S^*_{k-1}\,\Gamma \vdash_{\mathrm{W}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \hookrightarrow e''_k : \tau'_k\right)_{k\in B} \\ \left(\sigma_k = \mathrm{Gen}_{S^*_n\,\Gamma}(S^n_{k+1}\,(\Delta_k \Rightarrow \tau'_k)), \quad e'_k = \Lambda h_k.e''_k\right)_{k\in B} \\ \left(\begin{array}{l}\Delta'_k \mid \Delta'_{k-1}, \ldots, \Delta'_1, \Delta^*_e, \Delta^*_p, \Delta_{SR} \Vdash_{\mathrm{W}} \\ \qquad w_k : L_k \in (S^n_1\,\tau'_e)\,?\,\mathrm{HasMGBr}\ (S^n_1\,\tau'_e)\ L_k\ \sigma_k\ \tau'\end{array}\right)_{k\in B}\end{array}}{\begin{array}{l}\Delta'_n, \ldots, \Delta'_1, \Delta^*_e, \Delta^*_p, \Delta_{SR} \mid S^*_n\,\Gamma \vdash_{\mathrm{W}} \mathbf{case}^D\ e\ \mathbf{of}\ (L_k^D\ x_k\ \to\ e_k)_{k\in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B} \\ \qquad\qquad\qquad \mathbf{of}\ (L_k\ \to\ e'_k)_{k\in B} : \tau'\end{array}}$$

Let us call $\Delta^* = \Delta'_n, \ldots, \Delta'_1, \Delta^*_e, \Delta^*_p, \Delta_{SR}$. We need to prove that

$$\Delta^* \mid S^*_n\,\Gamma \vdash_{\mathrm{S}} \begin{array}{l} \mathbf{case}^D\, e\ \mathbf{of}\ (L_k^{\ D}\ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \\ \hookrightarrow \\ \mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y\ \mathbf{and}\ (w_k)_{k\in B}\ \mathbf{of}\ (L_k\ \rightarrow\ e'_k)_{k\in B} : \tau' \end{array}$$

By inductive hypothesis on the first premise

$$\Delta_e \mid S_e\,\Gamma \vdash_{\mathrm{S}} e : Y^D\ \hookrightarrow\ e' : \tau'_e$$

By proposition 5.1 on the statement above, since $\Delta^*_e = S^n_1\,\Delta_e$ and $S^*_n = S^n_1\,S_e$

$$\Delta^*_e \mid S^*_n\,\Gamma \vdash_{\mathrm{S}} e : Y^D\ \hookrightarrow\ e' : S^n_1\,\tau'_e$$

By proposition 5.2, since $\Delta'_n, \ldots, \Delta'_1, \bar{h}_e : \Delta^*_e, \Delta^*_p, \Delta_{SR} \Vdash \bar{h}_e : \Delta^*_e$

$$\Delta^* \mid S^*_n\,\Gamma \vdash_{\mathrm{S}} e : Y^D\ \hookrightarrow\ e' : S^n_1\,\tau'_e \tag{B.111}$$

By proposition 5.7 on the second premise

$$\Delta_p, \Delta_e \Vdash v^y : \mathrm{IsPolySum}\ \tau'_e$$

By rules (Close) — applying S^n_1 — and (Trans) — since $\Delta^* \Vdash S^n_1\,(\Delta_p, \Delta_e)$ — on the statement above

$$\Delta^* \Vdash v^y : \mathrm{IsPolySum}\ (S^n_1\,\tau'_e) \tag{B.112}$$

By proposition 5.8 on the third premise

$$\Delta_{SR} \vdash_{\mathrm{SR}} \tau\ \hookrightarrow\ \tau'$$

By proposition 4.12 on the statement above, since $\Delta^* \Vdash \Delta_{SR}$

$$\Delta^* \vdash_{\mathrm{SR}} \tau\ \hookrightarrow\ \tau' \tag{B.113}$$

Now let us take k between 1 and n. By inductive hypothesis on the fourth premise,

$$h_k : \Delta_k \mid S_k\,S^*_{k-1}\,\Gamma \vdash_{\mathrm{S}} \lambda^D x_k.e_k : Y(L_k) \rightarrow^D \tau\ \hookrightarrow\ e''_k : \tau'_k$$

By proposition 5.1, applying S^n_{k+1} to the statement above

$$h_k : S^n_{k+1}\,\Delta_k \mid S^*_n\,\Gamma \vdash_{\mathrm{S}} \lambda^D x_k.e_k : Y(L_k) \rightarrow^D \tau\ \hookrightarrow\ e''_k : S^n_{k+1}\,\tau'_k \tag{B.114}$$

By proposition 5.7 on the last premise

$$\Delta'_k, \ldots, \Delta'_1, \Delta^*_e, \Delta^*_p, \Delta_{SR} \Vdash w_k :\ L_k\ \in (S^n_1\,\tau'_e)\,?\,\mathrm{HasMGBr}\ (S^n_1\,\tau'_e)\ \ L_k\ \ \sigma_k\ \ \tau'$$

By rule (Trans), since $\Delta^* \Vdash \Delta'_k, \ldots, \Delta_{SR}$

$$\Delta^* \Vdash w_k :\ L_k\ \in (S^n_1\,\tau'_e)\,?\,\mathrm{HasMGBr}\ (S^n_1\,\tau'_e)\ \ L_k\ \ \sigma_k\ \ \tau' \tag{B.115}$$

The result follows from applying rule (S-POLYCASE) to B.111, B.112, B.113, and for each k, to B.114 and B.115.

B.25 Proof of theorem 5.12, section 5.2

Theorem 5.12 ⋆ *If* $\bar{h} : \Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \;\hookrightarrow\; e' : \tau'$
then $\bar{h}'_w : \Delta'_w \mid T'_w\,\Gamma \vdash_{\mathrm{W}} e : \tau \;\hookrightarrow\; e'_w : \tau'_w$
and there exist a substitution R *and evidence* $\bar{v}'_w$ *such that*
$S \approx R T'_w$
$\tau' = R\,\tau'_w$
$\bar{h} : \Delta \Vdash \bar{v}'_w : R\,\Delta'_w$
$e' = e'_w[\bar{v}'_w/\bar{h}'_w]$

Proof: By induction on the S derivation.
Extending proofs of propositions by Martínez López [2005, 7.18] and Russo [2004, 4.14].

Case (S-POLYCONSTR): We have a derivation of the form

$$\frac{\bar{h} : \Delta \vdash_{\mathrm{SR}} Y^D \;\hookrightarrow\; \tau'_e \qquad \bar{h} : \Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : Y(L_j) \;\hookrightarrow\; e' : \tau'_j \qquad \bar{h} : \Delta \Vdash v_j : \mathrm{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j}{\bar{h} : \Delta \mid S\,\Gamma \vdash_{\mathrm{S}} L_j^D\ e : Y^D \;\hookrightarrow\; L_j^{v_j}\ e' : \tau'_e}$$

By proposition 5.9 on the first premise, there exist $\bar{h}'_{w1} : \Delta'_{w1}$, τ'_{w1} and C'_{w1} such that

$$\bar{h}'_{w1} : \Delta'_{w1} \vdash_{\text{W-SR}} Y^D \;\hookrightarrow\; \tau'_{w1} \tag{B.116}$$

with all the variables fresh and such that $C'_{w1} : \mathrm{Gen}_{\emptyset,\emptyset}(\Delta'_{w1} \Rightarrow \tau'_{w1}) \geq (\Delta \mid \tau'_e)$, that is there is a substitution R_1 and evidence $\bar{v}'_{w1}$ such that

$$\tau'_e = R_1\,\tau'_{w1} \tag{B.117}$$
$$\bar{h} : \Delta \Vdash \bar{v}'_{w1} : R_1\,\Delta'_{w1} \tag{B.118}$$
$$C'_{w1} = [](\!(\bar{v}'_{w1})\!)$$

By inductive hypothesis on the second premise,

$$\bar{h}'_{w2} : \Delta'_{w2} \mid T'_w\,\Gamma \vdash_{\mathrm{W}} e : Y(L_j) \;\hookrightarrow\; e'_{w2} : \tau'_{w2} \tag{B.119}$$

and there exist R_2, $\bar{v}'_{w2}$ such that

$$S \approx R_2\,T'_w \tag{B.120}$$
$$\tau'_j = R_2\,\tau'_{w2} \tag{B.121}$$
$$\bar{h} : \Delta \Vdash \bar{v}'_{w2} : R_2\,\Delta'_{w2} \tag{B.122}$$
$$e' = e'_{w2}[\bar{v}'_{w2}/\bar{h}'_{w2}] \tag{B.123}$$

Let us take $\bar{h}'_w : \Delta'_w = \bar{h}'_{w1} : \Delta'_{w1}, \bar{h}'_{w2} : \Delta'_{w2}, h'_{w3} : \mathrm{HasPolyC}\ \tau'_{w1}\ \ L_j\ \ \tau'_{w2}$. So trivially, from B.116 and B.119 by (W-POLYCONSTR)

$$\bar{h}'_w : \Delta'_w \mid T'_w\,\Gamma \vdash_{\mathrm{W}} L_j^D\ e : Y^D \;\hookrightarrow\; L_j^{h'_{w3}}\ e'_{w2} : \tau'_{w1}$$

Let us define R such that $R\,t = R_1\,t$ if $t \in \mathrm{dom}(R_1)$ and otherwise $R\,t = R_2\,t$. Let us also take $\bar{v}'_w = \bar{v}'_{w1}, \bar{v}'_{w2}, v_j$. It suffices to verify

1. $S \approx R\, T'_w$
   It follows by B.120 and the fact that $R_2 \approx R$ by definition of R.
2. $\tau'_e = R\, \tau'_{w1}$
   It follows from B.117, since by definition of R, $R\, \tau'_{w1} = R_1\, \tau'_{w1}$.
3. $\bar{h} : \Delta \Vdash \bar{v}'_w : R\, \Delta'_w$
   - $\bar{h} : \Delta \Vdash \bar{v}'_{w1} : R\, \Delta'_{w_1}$ follows from B.118, since by definition of R, $R\, \Delta'_{w1} = R_1\, \Delta'_{w1}$.
   - $\bar{h} : \Delta \Vdash \bar{v}'_{w2} : R\, \Delta'_{w_2}$ follows from B.122, since by definition of R, $R\, \Delta'_{w2} = R_2\, \Delta'_{w2}$.
   - $\bar{h} : \Delta \Vdash v_j : R\,(\text{HasPolyC}\ \tau'_{w1}\ \ L_j\ \ \tau'_{w2})$ follows from the third premise, B.117, B.121 and the definition of R.

$$R\,(\text{HasPolyC}\ \tau'_{w1}\ \ L_j\ \ \tau'_{w2}) = \text{HasPolyC}\ (R\, \tau'_{w1})\ \ L_j\ \ (R\, \tau'_{w2}) = \text{HasPolyC}\ (R_1\, \tau'_{w1})\ \ L_j\ \ (R_2\, \tau'_{w2}) = \text{HasPolyC}\ \tau'_e\ \ L_j\ \ \tau'_j$$

4. $L_j^{\ v_j}\ e' = \left(L_j^{\ h'_{w3}}\ e'_{w2}\right)[\bar{v}'_w/\bar{h}'_w]$
   From B.119 by lemma 5.10, we know that $EV(e'_{w2}) \subseteq \bar{h}'_{w2}$, so by B.123,

$$e'_{w2}[\bar{v}'_w/\bar{h}'_w] = e'_{w2}[\bar{v}'_{w2}/\bar{h}'_{w2}] = e'$$

   Similarly, assuming $\bar{h}'_{w1}$, $\bar{h}'_{w2}$ and h'_{w3} are disjoint,

$$h'_{w3}[\bar{v}'_w/\bar{h}'_w] = h'_{w3}[v_j/h'_{w3}] = v_j$$

   So

$$\left(L_j^{\ h'_{w3}}\ e'_{w2}\right)[\bar{v}'_w/\bar{h}'_w] = L_j^{\ h'_{w3}[\bar{v}'_w/\bar{h}'_w]}\ e'_{w2}[\bar{v}'_w/\bar{h}'_w] = L_j^{\ v_j}\ e'$$

This completes the proof.

Case (S-POLYCASE): We have a derivation of the form

$$\frac{\begin{array}{l}\bar{h} : \Delta \mid S\,\Gamma \vdash_{\text{S}} e : Y^D\ \hookrightarrow\ e' : \tau'_e \\ \bar{h} : \Delta \Vdash v^y : \text{IsPolySum}\ \tau'_e \\ \bar{h} : \Delta \vdash_{\text{SR}} \tau\ \hookrightarrow\ \tau' \\ \left(\begin{array}{l}\bar{h}_k : \Delta_k \mid S\,\Gamma \vdash_{\text{S}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau\ \hookrightarrow\ e''_k : \tau'_k \\ \sigma_k = \text{Gen}_{S\,\Gamma}(\Delta_k \Rightarrow \tau'_k), \qquad e'_k = \Lambda \bar{h}_k.e''_k \\ \bar{h} : \Delta \Vdash w_k : L_k \in \tau'_e\,?\,\text{HasMGBr}\ \tau'_e\ \ L_k\ \ \sigma_k\ \ \tau'\end{array}\right)_{k \in B}\end{array}}{\begin{array}{l}\bar{h} : \Delta \mid S\,\Gamma \vdash_{\text{S}}\ \textbf{case}^D\, e\ \textbf{of}\ (L_k^{\ D}\ \ x_k\ \to\ e_k)_{k \in B} : \tau \\ \qquad\qquad \hookrightarrow \\ \qquad\qquad \textbf{polycase}_v\ e'\ \textbf{with}\ v^y\ \textbf{and}\ (w_k)_{k \in B} \\ \qquad\qquad\qquad \textbf{of}\ (L_k\ \to\ e'_k)_{k \in B} : \tau'\end{array}}$$

We will use the notation as defined in figure 5.4.

By inductive hypothesis on the first premise, there are R_e and $\bar{v}'_{w_e}$ such that

$$\bar{h}'_{w_e} : \Delta_{w_e} \mid T_e\,\Gamma \vdash_{\mathrm{W}} e : Y^D \;\hookrightarrow\; e'_w : \tau'_{w_e} \tag{B.124}$$
$$S \approx R_e\,T_e \tag{B.125}$$
$$\tau'_e = R_e\,\tau'_{w_e} \tag{B.126}$$
$$\bar{h} : \Delta \Vdash \bar{v}'_{w_e} : R_e\,\Delta_{w_e} \tag{B.127}$$
$$e' = e'_w[\bar{v}'_{w_e}/\bar{h}'_{w_e}] \tag{B.128}$$

By proposition 5.9 on the third premise

$$\bar{h}_{SR} : \Delta_{SR} \vdash_{\text{W-SR}} \tau \;\hookrightarrow\; \tau'_w \tag{B.129}$$

with all the variables in Δ_{SR} and τ'_w fresh and such that $C'_{SR} : \mathrm{Gen}_{\emptyset,\emptyset}(\Delta_{SR} \Rightarrow \tau'_w) \geq (\Delta \mid \tau')$, that is there exist R_{SR} and $\bar{v}'_{SR}$ such that

$$\tau' = R_{SR}\,\tau'_w \tag{B.130}$$
$$\bar{h} : \Delta \Vdash \bar{v}'_{SR} : R_{SR}\,\Delta_{SR} \tag{B.131}$$
$$C'_{SR} = [](\bar{v}'_{SR})$$

Now let us prove that for all $k = 1, \ldots, n$, there exist $\bar{h}'_{w_k} : \Delta'_{w_k}$, T_k, $e''_{w_k} : \tau'_{w_k}$, R_k and $\bar{v}_{w_k}$ such that

$$\bar{h}'_{w_k} : \Delta'_{w_k} \mid T_k\,\Gamma \vdash_{\mathrm{W}} \lambda^D x_k.e_k : Y(L_k) \to^D \tau \;\hookrightarrow\; e''_{w_k} : \tau'_{w_k} \tag{B.132}$$
$$S \approx R_k\,T^*_k \tag{B.133}$$
$$\tau'_k = R_k\,\tau'_{w_k} \tag{B.134}$$
$$\bar{h}_k : \Delta_k \Vdash \bar{v}'_{w_k} : R_k\,\Delta'_{w_k} \tag{B.135}$$
$$e''_k = e'_{w_k}[\bar{v}'_{w_k}/\bar{h}'_{w_k}] \tag{B.136}$$

We proceed by induction on k. For $k = 1$, from B.125, we can rewrite the fourth premise as

$$\bar{h}_1 : \Delta_1 \mid R_e\,(T_e\,\Gamma) \vdash_{\mathrm{S}} \lambda^D x_1.e_1 : Y(L_1) \to^D \tau \;\hookrightarrow\; e''_1 : \tau'_1$$

So by inductive hypothesis, there exist $\bar{h}'_{w_1} : \Delta'_{w_1}$, T_1, $e''_{w_1} : \tau'_{w_1}$, R_1 and $\bar{v}_{w_1}$ such that

$$\bar{h}'_{w_1} : \Delta'_{w_1} \mid T_1\,(T_e\,\Gamma) \vdash_{\mathrm{W}} \lambda^D x_1.e_1 : Y(L_k) \to^D \tau \;\hookrightarrow\; e''_{w_1} : \tau'_{w_1}$$
$$R_e \approx R_1\,T_1$$
$$\tau'_1 = R_1\,\tau'_{w_1}$$
$$\bar{h}_1 : \Delta_1 \Vdash \bar{v}'_{w_1} : R_1\,\Delta'_{w_1}$$
$$e''_1 = e'_{w_1}[\bar{v}'_{w_1}/\bar{h}'_{w_1}]$$

The fact that $S \approx R_1\,T^*_1$ follows from $S \approx R_e\,T_e \approx R_1\,T_1\,T_e$.

Similarly we can prove the statement above for $k+1$, assuming it holds for k. Since $S \approx R_k\,T^*_k$, we can rewrite the fourth premise as

$$\bar{h}_{k+1} : \Delta_{k+1} \mid R_k\,(T^*_k\,\Gamma) \vdash_{\mathrm{S}} \lambda^D x_{k+1}.e_{k+1} : Y(L_{k+1}) \to^D \tau \;\hookrightarrow\; e''_{k+1} : \tau'_{k+1}$$

So by inductive hypothesis, there exist $\bar{h}'_{w_{k+1}} : \Delta'_{w_{k+1}}$, T_{k+1}, $e''_{w_{k+1}} : \tau'_{w_{k+1}}$, R_{k+1} and $\bar{v}_{w_{k+1}}$ such that

$$\bar{h}'_{w_{k+1}} : \Delta'_{w_{k+1}} \mid T_{k+1}\,(T_k^*\,\Gamma) \vdash_{\mathrm{W}} \lambda^D x_{k+1}.e_{k+1} : Y(L_{k+1}) \to^D \tau \;\hookrightarrow\; e''_{w_{k+1}} : \tau'_{w_{k+1}}$$
$$R_k \approx R_{k+1}\,T_{k+1}$$
$$\tau'_{k+1} = R_{k+1}\,\tau'_{w_{k+1}}$$
$$\bar{h}_{k+1} : \Delta_{k+1} \Vdash \bar{v}'_{w_{k+1}} : R_{k+1}\,\Delta'_{w_{k+1}}$$
$$e''_{k+1} = e'_{w_{k+1}}[\bar{v}'_{w_{k+1}}/\bar{h}'_{w_{k+1}}]$$

The fact that $S \approx R_{k+1}\,T^*_{k+1}$ follows from $S \approx R_k\,T^*_k \approx R_{k+1}\,T_{k+1}\,T^*_k$.

From the proof above it also follows that

$$R_e \approx R_1\,T_1 \approx R_2\,T_2\,T_1 \approx \ldots \approx R_n\,T_n\;\ldots\;T_1 = R_n\,T_1^n \tag{B.137}$$
$$R_k \approx R_{k+1}\,T_{k+1} \approx \ldots \approx R_n\,T_n\;\ldots\;T_{k+1} = R_n\,T_{k+1}^n \tag{B.138}$$

Now by definition 2.5,

$$\Lambda\bar{h}_k.[]((\bar{v}'_{w_k})) : \mathrm{Gen}_{S\,\Gamma}(R_k\,(\Delta'_{w_k} \Rightarrow \tau'_{w_k})) \geq \mathrm{Gen}_{S\,\Gamma}(\Delta'_k \Rightarrow \tau'_k)$$

and since $S \approx R_n\,T^*_n$ (from B.133) and $R_k \approx R_n\,T^n_{k+1}$ (from B.138)

$$\Lambda\bar{h}_k.[]((\bar{v}'_{w_k})) : \mathrm{Gen}_{R_n\,T^*_n\,\Gamma}(R_n\,T^n_{k+1}\,(\Delta'_{w_k} \Rightarrow \tau'_{w_k})) \geq \mathrm{Gen}_{S\,\Gamma}(\Delta'_k \Rightarrow \tau'_k) \tag{B.139}$$

By proposition 2.16-3,

$$[] : R_n\,\mathrm{Gen}_{T^*_n\,\Gamma}(T^n_{k+1}\,(\Delta'_{w_k} \Rightarrow \tau'_{w_k})) \geq \mathrm{Gen}_{R_n\,T^*_n\,\Gamma}(R_n\,T^n_{k+1}\,(\Delta'_{w_k} \Rightarrow \tau'_{w_k})) \tag{B.140}$$

Let us define $\sigma'_{w_k} = \mathrm{Gen}_{T^*_n\,\Gamma}(T^n_{k+1}\,(\Delta'_{w_k} \Rightarrow \tau'_{w_k}))$, and $e'_{w_k} = \Lambda\bar{h}'_{w_k}.e''_{w_k}$. So by proposition 2.7-2 on B.139 and B.140, and rule IsMG

$$\emptyset \Vdash \Lambda\bar{h}_k.[]((\bar{v}'_{w_k})) : \mathrm{IsMG}\;(R_n\,\sigma'_{w_k})\;\sigma_k \tag{B.141}$$

Now let us take

$$\begin{aligned}\bar{h}'_w : \Delta'_w \;=\;& \bar{h}'_{w_e} : T_1^n\,\Delta'_{w_e},\\ & h^y_w : \mathrm{IsPolySum}\;(T_1^n\,\tau'_{w_e}),\\ & \bar{h}'_{SR} : \Delta'_{SR},\\ & \left(h''_k : L_k \in (T_1^n\,\tau'_{w_e})\,?\,\mathrm{HasMGBr}\;(T_1^n\,\tau'_{w_e})\;\;L_k\;\;\sigma'_{w_k}\;\;\tau'_w\right)_{k\in B}\end{aligned}$$

By rule (W-POLYCASE) on B.124, B.129, B.132, and definition of Δ'_w,

$$\bar{h}'_w : \Delta'_w \mid T_n^*\,\Gamma \vdash_{\mathrm{W}} \mathbf{case}^D\,e\;\mathbf{of}\;(L_k^{\,D}\;\;x_k\;\to\;e_k)_{k\in B} : \tau \;\hookrightarrow\; \mathbf{polycase}_v\;e'_w\;\mathbf{with}\;h^y_w\;\mathbf{and}\;(h''_k)_{k\in B}\;\mathbf{of}\;\left(L_k\;\to\;e'_{w_k}\right)_{k\in B} : \tau'$$

From B.129, all the variables in Δ_{SR} and τ'_w are fresh, so we can assume $\mathrm{dom}(R_{SR}) \cap \mathrm{dom}(S') = \emptyset$ for every other substitution S' involved in this proof. We define R such that $R\,t = R_{SR}\,t$ if $t \in \mathrm{dom}(R_{SR})$, and otherwise $R\,t = R_n\,t$. We also define

$$\begin{aligned}v'_w \;&=\; v'_{w_e}, v^y, v'_{SR}, \left(w'_k\right)_{k\in B}\\ w'_k \;&=\; \mathbf{if}_v\;L_k \in v^y\;\mathbf{then}\;w_k \diamond (\mathbf{if}_v\;L_k \in v^y\;\mathbf{then}\;\Lambda\bar{h}_k.[]((\bar{v}'_{w_k}))\;\mathbf{else}\;\bullet)\;\mathbf{else}\;\bullet\end{aligned}$$

It suffices to prove

1. $S \approx R\,T_n^*$
   It follows from B.133 for $k = n$, since $R\,T_n^* = R_n\,T_n^*$.
2. $\tau' = R\,\tau'_w$
   It follows from B.130, since $R\,\tau'_w = R_{SR}\,\tau'_w$.
3. $\bar{h} : \Delta \Vdash \bar{v}'_w : R\,\Delta'_w$
   - $\bar{h} : \Delta \Vdash v'_{w_e} : R\,T_1^n\,\Delta_{w_e}$ holds from B.127 and B.137.
   - $\bar{h} : \Delta \Vdash v^y : \text{IsPolySum}\ (R\,T_1^n\,\tau'_{w_e})$ holds from the second premise, B.126 and B.137.
   - $\bar{h} : \Delta \Vdash v'_{SR} : R\,\Delta'_{SR}$ holds from B.131 and the fact that $R\,\Delta'_{SR} = R_{SR}\,\Delta'_{SR}$.
   - $\bar{h} : \Delta \Vdash w'_k :\ L_k\ \in (R\,T_1^n\,\tau'_{w_e})\,?\,\text{HasMGBr}\ (R\,T_1^n\,\tau'_{w_e})\ L_k\ (R\,\sigma'_{w_k})\ (R\,\tau'_w)$
     From B.126 and B.137 we have $\tau'_e = R_n\,T_1^n\,\tau'_{w_e} = R\,T_1^n\,\tau'_{w_e}$. Also from B.130 we have $\tau' = R_{SR}\,\tau'_w = R\,\tau'_w$. So we must see that

     $$\bar{h} : \Delta \Vdash w_k :\ L_k\ \in \tau'_e\,?\,\text{HasMGBr}\ \tau'_e\ L_k\ (R\,\sigma'_{w_k})\ \tau'$$

     From B.141 and the second premise, by lemma 4.6

     $$\bar{h} : \Delta \Vdash \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ \Lambda\bar{h}_k.[](\!(\bar{v}'_{w_k})\!)\ \mathbf{else}\ \bullet : L_k \in \tau'_e\,?\,\text{IsMG}\ (R\,\sigma'_{w_k})\ \sigma_k$$

     Let us call

     $$v_{aux} = \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ \Lambda\bar{h}_k.[](\!(\bar{v}'_{w_k})\!)\ \mathbf{else}\ \bullet$$
     $$\Delta_{aux} = \text{IsMG}\ (R\,\sigma'_{w_k})\ \sigma_k, \text{HasMGBr}\ \tau'_e\ L_k\ \sigma'_k\ \tau'$$

     By the statement above and the last premise, we have

     $$\bar{h} : \Delta \Vdash v_{aux}, w_k : L_k \in \tau'_e\,?\,\Delta_{aux}$$

     Also by rule (Comp-MGBr)

     $$h_1^{aux}, h_2^{aux} : \Delta_{aux} \Vdash h_2^{aux} \diamond h_1^{aux} : \text{HasMGBr}\ \tau'_e\ L_k\ (R\,\sigma'_{w_k})\ \tau'$$

     so again by lemma 4.6

     $$\bar{h} : \Delta \Vdash \mathbf{if}_v\ L_k \in v^y\ \mathbf{then}\ w_k \diamond v_{aux}\ \mathbf{else}\ \bullet$$
     $$: L_k \in \tau'_e\,?\,\text{HasMGBr}\ \tau'_e\ L_k\ (R\,\sigma'_{w_k})\ \tau'$$

     By definition of v_{aux}, this is what we wanted to show.

4.
$$\begin{array}{l}\mathbf{polycase}_v\ e'\ \mathbf{with}\ v^y \\ \quad \mathbf{and}\ (w_k)_{k\in B} \\ \quad \mathbf{of}\ (L_k\ \rightarrow\ e'_k)_{k\in B}\end{array} = \left(\begin{array}{c}\mathbf{polycase}_v\ e'_w\ \mathbf{with}\ h_w^y \\ \quad \mathbf{and}\ (h''_k)_{k\in B} \\ \quad \mathbf{of}\ \left(L_k\ \rightarrow\ e'_{w_k}\right)_{k\in B}\end{array}\right)[\bar{v}'_w/\bar{h}'_w]$$

By lemma 5.10 on B.124 we have $EV(e'_w) \subseteq \bar{h}'_{w_e}$. Similarly for all k, from B.132 we know $EV\left(e''_{w_k}\right) \subseteq \bar{h}'_{w_k}$, so $e'_{w_k} = \Lambda\bar{h}'_{w_k}.e''_{w_k}$ has no free evidence variables. Assuming by alpha conversion that $\bar{h}'_{w_e}$, h_w^y and h''_k are disjoint, we have

$$\left(\begin{array}{c}\mathbf{polycase}_v\ e'_w\ \mathbf{with}\ h_w^y \\ \quad \mathbf{and}\ (h''_k)_{k\in B} \\ \quad \mathbf{of}\ \left(L_k\ \rightarrow\ e'_{w_k}\right)_{k\in B}\end{array}\right)[\bar{v}'_w/\bar{h}'_w] = \begin{array}{c}\mathbf{polycase}_v\ e'_w[\bar{v}'_{w_e}/\bar{h}'_{w_e}]\ \mathbf{with} \\ h_w^y[v^y/h_w^y]\ \mathbf{and}\ (h''_k[w'_k/h''_k])_{k\in B} \\ \mathbf{of}\ \left(L_k\ \rightarrow\ e'_{w_k}\right)_{k\in B}\end{array}$$

so, by B.128 and definition of substitution, we need to prove that

$$\begin{array}{l}\textbf{polycase}_v\ e'\ \textbf{with}\ v^y \\ \quad \textbf{and}\ (w_k)_{k\in B} \\ \quad \textbf{of}\ (L_k\ \to\ e'_k)_{k\in B}\end{array} \quad = \quad \begin{array}{l}\textbf{polycase}_v\ e'\ \textbf{with}\ v^y \\ \quad \textbf{and}\ (w'_k)_{k\in B} \\ \quad \textbf{of}\ \left(L_k\ \to\ e'_{w_k}\right)_{k\in B}\end{array}$$

Without loss of generality by definition of equivalence on a $\textbf{polycase}_v$ expression (see section A.6), we can assume v^y and w_k are not variables. In particular, v^y is of the form $\{L_{k,i}\}_{k\in I, i\in I'_k}$ and w_k is of the form $\left\langle n, (v_i)_{i\in I'_k}\right\rangle$ for all $k \in I$.

Let us take any $k \in I$, $w_k = \left\langle n, (v_i)_{i\in I'_k}\right\rangle$. By definition of w'_k

$$w'_k = \textbf{if}_v\ L_k \in v^y\ \textbf{then}\ w_k \diamond (\textbf{if}_v\ L_k \in v^y\ \textbf{then}\ \Lambda\bar{h}_k.[](\!(\bar{v}'_{w_k})\!)\ \textbf{else}\ \bullet)\ \textbf{else}\ \bullet$$

and since $L_k \in v^y$ is true,

$$w'_k = w_k \diamond \Lambda\bar{h}_k.[](\!(\bar{v}'_{w_k})\!) = \left\langle n, \left(v_i \circ \Lambda\bar{h}_k.[](\!(\bar{v}'_{w_k})\!)\right)_{i\in I'_k}\right\rangle$$

By the reduction rule for $\textbf{polycase}_v$ expressions, it suffices to see that

$$v_i[e'_k] = (v_i \circ \Lambda\bar{h}_k.[](\!(\bar{v}'_{w_k})\!))[e'_{w_k}]$$

for all $k \in I, i \in I'_k$ ($k \notin I$ are ignored). Indeed, by definition of e'_{w_k}, evidence β-reduction, B.136, and definition of e'_k

$$\begin{array}{r}(v_i \circ \Lambda\bar{h}_k.[](\!(\bar{v}'_{w_k})\!))[e'_{w_k}] = v_i[\Lambda\bar{h}_k.e'_{w_k}(\!(\bar{v}'_{w_k})\!)] = v_i[\Lambda\bar{h}_k.\Lambda\bar{h}'_{w_k}.e''_{w_k}(\!(\bar{v}'_{w_k})\!)] = \\ v_i[\Lambda\bar{h}_k.e''_{w_k}[\bar{v}'_{w_k}/\bar{h}'_{w_k}]] = v_i[\Lambda\bar{h}_k.e''_k] = v_i[e'_k]\end{array}$$

as we needed.

This completes the proof.

Bibliography

[Aiken, 1999] Alexander Aiken. Introduction to set constraint-based program analysis. In *Science of Computer Programming*, volume 35, pages 79–111, 1999.

[Au *et al.*, 1991] Wing-Yee Au, Daniel Weise, and Scott Seligman. Automatic generation of compiled simulations through program specialization. In *Proceedings of the 28th Design Automation Conference*, pages 205–210, San Francisco, California, USA, June 1991. IEEE Computer Society Press.

[Augustsson, 1993] Lennart Augustsson. Implementing Haskell overloading. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 65–73, New York, NY, USA, 1993. ACM Press.

[Augustsson, 1997] Lennart Augustsson. Partial evaluation in aircraft crew planning. In Gallagher [1997], pages 127–136.

[Beshers and Feiner, 1997] Clifford Beshers and Steven Feiner. Generating efficient virtual worlds for visualization using partial evaluation and dynamic compilation. In Gallagher [1997], pages 107–115.

[Clément *et al.*, 1986] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 13–27, Cambridge, Massachussets, USA, August 1986. ACM Press.

[Consel and Danvy, 1993] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of 20th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages (POPL '93)*, pages 493–501, Charleston, South Carolina, USA, January 1993. ACM Press.

[Curry and Feys, 1958] Haskell B. Curry and Robert Feys. *Combinatory Logic*. North Holland, Amsterdam, 1958.

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

[Dussart *et al.*, 1995] Dirk Dussart, Eddy Bevers, and Karel De Vlaminck. Polyvariant constructor specialisation. In *PEPM '95: Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 54–65, New York, NY, USA, 1995. ACM Press.

[Dussart *et al.*, 1997] Dirk Dussart, John Hughes, and Peter Thiemann. Type specialisation for imperative languages. *SIGPLAN Not.*, 32(8):204–216, 1997.

[Futamura, 1971] Yoshihiko Futamura. Partial evaluation of computation process - An approach to a compiler-compiler. *Computer, Systems, Controls*, 2(5):45–50, 1971.

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

[Glück, 2002] Robert Glück. Jones optimality, binding-time improvements, and the strength of program specializers. In *ASIA-PEPM '02: Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, pages 9–19, New York, NY, USA, 2002. ACM Press.

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

[Hannan and Miller, 1992] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[Hindley, 1969] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[Hogg, 1996] Jonathan A. H. Hogg. Dynamic hardware generation mechanism based on partial evaluation. In *Proceedings of 3rd Workshop on Designing Correct Circuits (DCC '96)*, Electronic Workshops in Computing, Båstad, Sweden, September 1996. Springer-Verlag.

[Hughes, 1996a] John Hughes. An introduction to program specialisation by type inference. In *Functional Programming*. Glasgow University, July 1996. Published electronically.

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

[Hughes, 1998] John Hughes. Type specialization. In *ACM Computing Surveys*, volume 30. ACM Press, September 1998. Article 14. Special issue: electronic supplement to the September 1998 issue.

[Jones *et al.*, 1985] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Proceedings of the 1st International Conference on Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science (LNCS)*, pages 124–140, Dijon, France, May 1985. Springer-Verlag.

[Jones *et al.*, 1989] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, February 1989.

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

[Jones, 1988] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. *New Generation Comput.*, 6(2-3):291–302, 1988.

[Jones, 1993] Mark P. Jones. Coherence for qualified types. Technical Report Research Report YALEU/DCS/RR-989, Yale University, September 1993.

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

[Jones, 1995] Mark P. Jones. Simplifying and improving qualified types. In *Functional Programming Languages and Computer Architecture*, pages 160–169, 1995.

[Launchbury, 1991] John Launchbury. *Projection factorisations in partial evaluation.* Cambridge University Press, New York, NY, USA, 1991.

[Lawall, 1998] Julia Lawall. Faster Fourier Transforms via automatic program specialization. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, volume 1706 of *Lecture Notes in Computer Science (LNCS)*, pages 338–355, Copenhagen, Denmark, June 1998. Springer-Verlag.

[Marlet *et al.*, 1999] Renaud Marlet, Scott Thibault, and Charles Consel. Efficient implementations of software architectures via partial evaluation. *Automated Software Engineering: An International Journal*, 6(4):411–440, October 1999.

[Martínez López and Badenes, 2003] Pablo E. Martínez López and Hernán Badenes. Simplifying and solving qualified types for principal type specialisation. In *Proceedings of 7th Workshop Argentino de Informática Teórica (WAIT 2003)*, September 2003.

[Martínez López and Hughes, 2002] Pablo E. Martínez López and John Hughes. Principal type specialisation. In Wei-Ngan Chin, editor, *Proceedings of the 2002 ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105. ACM Press, September 2002.

[Martínez López, 2005] Pablo E. Martínez López. *The Notion of Principality in Type Specialisation.* PhD thesis, University of Buenos Aires, November 2005.

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

[Mogensen, 1993] Torben Æ. Mogensen. Constructor specialization. In *PEPM '93: Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 22–32, New York, NY, USA, 1993. ACM Press.

[Muller *et al.*, 1998] Gilles Muller, Renaud Marlet, Eugen N. Volanschi, Charles Consel, Calton Pu, and Ashvin Goel. Fast, optimized Sun RPC using automatic program specialization.

In *Proceedings of 19th IEEE International Conference on Distributed Computing Systems (ICDCS '98)*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[Peterson and Jones, 1993] John Peterson and Mark Jones. Implementing type classes. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 227–236, New York, NY, USA, 1993. ACM Press.

[Russo, 2004] Alejandro Russo. Principal type specialization of dynamic sum-types. Work for Graduation, University of Rosario, Argentina, August 2004.

[Taha *et al.*, 2001] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Olivier Danvy and Andrzej Filinski, editors, *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science (LNCS)*, pages 257–275, London, UK, May 2001. Springer-Verlag.

[Thibault *et al.*, 1998] Scott Thibault, Charles Consel, and Gilles Muller. Safe and efficient active network programming. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, USA, October 1998.

[Thiemann, 1999] Peter Thiemann. Interpreting specialization in type theory. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, pages 30–43, San Antonio, Texas, USA, January 1999.

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

[Wadler and Blott, 1989] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM Press.

[Wand, 1982] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, July 1982.