Principal Type Specialization of Dynamic Sum-Types

Computer Sciences Department
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Argentina

250 Pellegrini Avenue, Rosario. (2000) Santa Fe. Argentina.
Tel.: +54-341-4802656 int 116
e-mail: fceia@fceia.unr.edu.ar

Principal Type Specialization of Dynamic Sum-Types

Graduate Thesis

to obtain the degree of Licentiate in Computer Sciences at the
National University of Rosario,
August 2004

by

Alejandro C. Russo

Santa Fe, Argentina.

Supervisor: Msc. Pablo E. Martínez López
LIFIA Laboratory
National University of La Plata
50 Street intersection 115 Street, First Floor
(1900) Buenos Aires, Argentina

Computer Sciences Department
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Argentina

e-mail:russo@fceia.unr.edu.ar
http://www.fceia.unr.edu.ar/~russo/

For my grandfather Antonio,
who was an skilled worker that always worked to improve my country.

Contents

Agradecimientos

" — Tus Zapatos de Plata te harán pasar el desierto — replicó Glinda —. Si hubieras conocido su poder, habrías podido volver a casa de tu tía Emma el primer día que llegaste a este país.
— ¡Pero entonces yo no habría tenido mi espléndido cerebro! — dijo el Espantapájaros —. Me habría pasado toda la vida en el maizal del granjero.
— Y yo no habría tenido mi amado corazón — dijo el Leñador de Hojalata —. Me habría quedado oxidado en el bosque hasta el fin del mundo.
— Y yo habría sido para siempre un cobarde — dijo el León — y ningún animal en la selva habría tenido para mí una palabra de simpatía."

Glinda cumple el deseo de Dorotea
El mago de Oz
L. Frank Baum

La culminación de este trabajo lleva consigo el cierre de una etapa importante de mi vida que me ha tomado varios años, tiempo en el que he aprendido tanto infinidad de aspectos técnicos como humanos. He podido recorrer mi camino "amarillo color limón" gracias a muchas personas que me han acompañado a través de él y en donde más de una vez, en tiempos de debilidad, he querido usar mis zapatos de plata. Sin embargo, todo, absolutamente todo, valió la pena. Realmente ha habido un cambio completo en mi persona y mi carácter se ha forjado en base a las dichas y desdichas vividas en este viaje; y es mi profundo deseo que las personas que me han acompañado, por un corto o largo trecho, hayan podido aprovechar mi humilde andar.

Primero que todo quiero agradecer a mi familia, mi padre Jorge, mi madre Leonor, mis hermanos Martín y Barbarina y mi abuela "Lecu", por su incondicional apoyo desde que emprendí este camino lejos de casa. Les agradezco eternamente por haberme brindado la posibilidad de formarme en una Universidad Pública Nacional y contribuir de esta manera a la formación científica de mi país.

Quiero por otra parte, agradecer a René A. Galván por sus consejos sobre que rumbo debía tomar en mi vida cuando era adolescente, siendo uno de los responsables de que haya comenzado esta carrera.

Durante todo mi cursado formé parte de un grupo extraordinario de amigos con los que sorteé diferentes etapas en nuestra meta por graduarnos. Quiero mencionar en especial a Diego A. Bottallo, Juan Manuel Rabasedas, Alejandro Rodríguez, Alejandro Tamalet, Natalia Colussi y Alejandra Usart. Existen también amigos que la vida los ha llevado por otros caminos e igualmente me han ayudado muchísimo: Andrés Gironacci, Adrían Ponce, Fernando Mut y Sebastían Flores; como así también a Canela L. Raigal, quien me apoyo emocionalmente durante mis inicios.

No puedo continuar sin expresar mi inmensa gratitud a los docentes que trabajan en el Departamento de Ciencias de la Computación por haber confiado en mí y por contribuir a mi formación docente. Quisiera mencionar especialmente a Guido Macchi, Federico Guimpel, Raúl Kantor, Graciela Nasini y Mariano Suárez Alvarez.

Me gustaría agradecer en especial a mi director, Pablo E. "Fidel" Martínez López por ser una excelente persona. Además de leer y comentar cada parte de este trabajo, siempre estuvo dispuesto a dedicarme su tiempo y ganas de enseñar. Estoy también gratificado debido a que nuestra relación ha ido más allá de lo profesional, llegando a la amistad. A él le debo la posibilidad que tengo hoy de participar en el ambiente científico de mi país, de conocer diferentes personas a nivel nacional e internacional, y sobre todo, el haberme enseñado a hacer las cosas desinteresadamente y por el bien de todos.

También quisiera mencionar a María Dolores Arzuaga, quien me dio su inagotable paciencia y comprensión a la hora que comencé este trabajo.

Por otro lado, es importante destacar el rol de "mis estudiantes"; estoy seguro que he aprendido más yo de ellos que ellos de mí.

A la hora de encarar cualquier actividad son necesarios los tiempo de distracción y en eso quiero agradecer especialmente a mis amigos de San Nicolás.

Otras personas han colaborado directa o indirectamente con este trabajo y a ellos también les expreso mi gratitud: Mauro Jaskelioff, Emilse Hidalgo, Lucía Pereira, Silvia Reyes y Franco Ferraro.

Agradecimiento especiales a la persona que amo, Natalia Sigalini, sin ella la finalización de este trabajo hubiese resultado imposible.

Finalmente, siempre que se finaliza una etapa se comienza otra, y quiero agradecer a quiénes me han ayudado a poder emprender un nuevo desafío, ellos son Pablo E. Martínez López, Andrei Sabelfeld, Nora Szasz, Ana Bove, Adriana Compagnoni y Pablo Garralda.

"Tía Emma acababa de salir de la casa para regar los repollos cuando miró y vio a Dorotea que corría hacia ella.
— ¡Mi querida! — exclamó, alzando a la niñita en brazos y cubriendo su cara de besos —, ¿de qué parte del mundo vienes?
— De la tierra de Oz — dijo Dorotea seriamente —. Y aquí está Totó también. Y ¡oh, tía Emma, me siento tan contenta de estar de nuevo en casa!"

De Nuevo En Casa
El Mago de Oz
Frank L. Baum

Acknowledgments

*" — Your Silver Shoes will carry you over the desert — replied Glinda —. If you had known their power you could have gone back to your Aunt Em the very first day you came to this country.
— But then I should not have had my wonderful brains! — cried the Scarecrow —. I might have passed my whole life in the farmer's cornfield.
— And I should not have had my lovely heart — said the Tin Woodman —. I might have stood and rusted in the forest till the end of the world.
— And I should have lived a coward forever — declared the Lion — and no beast in all the forest would have had a good word to say to me."*

Glinda Good Witch grants Dorothy's wish
The Wizard of Oz
L. Frank Baum

The culmination of this work is the closure of an important stage of my life, that has taken several years during which I have learned as much in technical aspects as in human ones.

I have been able to travel my "yellow-brick road" thanks to many people that have accompanied me along it, even when in more than one occasion, I almost "knocked my shoes together tree times".

However, everything, absolutely everything was worthwile. It has really been a complete change in my person, and my character has been built and shaped based on the hapiness and misfortunes lived during that trip. It is my deepest desire that, everybody who has accompanied me for a short or long distance, could take advantage of my humble walk.

First of all, I would like to thank my family, my father Jorge, my mother Leonor, my brother Martín, my sister Barbarina and my grandmother "Lecu", for their unconditional support since I took this road, far away from home. I am eternally grateful to them for giving me the possibility of being formed at A National State University, and contribute in this way to the scientific development of my country.

I would also like to thank René A. Galván for his advices on what direction should my life take when I was a teenager, being one of the responsibles of my decision to start this career.

During my studies, I was part of an extraordinary group of friends, with which I overcame different stages in our goal of graduating. I want to specially mention Diego A. Bottallo, Juan Manuel Rabasedas, Alejandro Rodríguez, Alejandro Tamalet, Natalia Colussi, and Alejandra Usart.

There are also friends to which life has taken by other roads, but they have also helped me a lot: Andrés Gironacci, Adrían Ponce, Fernando Mut, and Sebastían Flores; as well as Canela L. Raigal, who emotionally supported me in my beginnings.

I cannot continue without expressing my immense gratitude to the teachers that work at the Department of Computing Sciences, for having trusted me and contribute on my education as a teacher myself. I would like to mention Guido Macchi, Federico Guimpel, Raúl Kantor, Grasiela Nasini, and Mariano Suárez Alvarez.

I want to thank Pablo E. "Fidel" Martínez López for being an excellent person. Apart from reading and commenting each part of this work, he was always willing to give me his time and his teacher skills. I am very gratified because our relationship has gone beyond the professional aspect and has become a great friendship. To him I owe the possibility that I have today of participating in the scientific atmosphere of my country, of knowing different people of national and international level, and mainly, having learned how to act unselfishly and for the good of all.

I would also like to mention María Dolores Arzuaga, who gave me her endless patience and understanding when I started with this work.

On the other hand, it is important to highlight the role of my students. I am sure that I have learned more of them than they have learned from me.

When facing any activity, times for distraction are aldo needed For that reason I specially want to thank my friends from San Nicolás.

Other people that have collaborated directly or indirectly, and to whom I also want to express my gratitude are: Mauro Jaskelioff, Emilse Hidalgo, Lucía Pereira, Silvia Reyes, and Franco Ferraro.

Especial thanks to whom I love, Natalia Sigalini, it would have been impossible to end this work without her.

Finally, whenever a stage concludes it begins another one, and I want to thank to those who have helped me to undertake a new challenge. They are: Pablo E. Martínez López, Andrei Sabelfeld, Nora Szasz, Ana Bove, Adriana Compagnoni, and Pablo Garralda.

> *"Aunt Em had just come out of the house to water the cabbages when she looked up and saw Dorothy running toward her.*
> *— My darling child! — she cried, folding the little girl in her arms and covering her face with kisses. Where in the world did you come from?*
> *— From the Land of Oz — said Dorothy gravely. "And here is Toto, too. And oh, Aunt Em! I'm so glad to be at home again!"*

Home Again
The Wizard of Oz
Frank L. Baum

Chapter 1

Introduction

> *" — Mire: dentro del movimiento de la democracias la carta más peligrosa que se juega es precisamente la de la revolución. Una revolución se sabe siempre dónde comienza, pero nunca se puede saber dónde irá a terminar "*
>
> Lo que me dijo el general Uriburu
> J.M. Espigares Moreno
> Buenos Aires, 1933

In this chapter we briefly present the concepts behind *program specialization* and how it can be carried out by *type specialization*. In addition, we also explain the contribution of this work to the field.

1.1 Program Specialization

"There is a trade-off between efficiency and generality" was always the phrase said by teachers in several courses when I tried to write a program that solved many instances of a given problem and was efficient at the same time.

Programers usually want to write the minimum lines of code while possible. One way to obtain this is to write general programs, that is, a program that solves many similar problems. General programs are often clearer, more understandable and easier to implement than specific ones and we can assume that it is better to write general programs. However there exists a very important entity that does not combine neatly with general programs, in the sense of time of computing them: the computer. General programs are less efficient than specific ones, so we want to write general programs but at the same time we want efficient ones. The idea of *program specialization* is to provide an *automatic* form to go from a general and non-efficient program to a specific and efficient one. This is done by a program, here called *specializer*, whose input is a program and whose output is one or more particular versions of it. The program used as input is called *source program*, and those produced as output are called *residual programs*. The classic example is the recursive `power` function calculating x^n

```
power n x = if n == 1
              then x
              else x * power (n-1) x
```

whose computation involves several comparisons and recursive calls, but when the input parameter `n` is known — for example let us say it is `3` — it can be specialized to a non-recursive residual version which can only computes powers of that particular `n` — the function

```
power3 x = x * (x * x)
```

in our example. It is clear that the residual version is much more efficient than the source version when computing cubes. Program specialization has been studied from several different approaches; among them, *Partial Evaluation* [Jones *et al.*, 1993; Consel and Danvy, 1993] is by far the most popular and well-known.

Partial evaluation is a technique that 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 those computations that cannot be performed statically. 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 — called *dynamic data* because they are not known until run-time — 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 program [Futamura, 1971; Jones *et al.*, 1985; Jones *et al.*, 1989; Wand, 1982; Hannan and Miller, 1992]. In this case, the interpreter is used as the source program, the object program is used as the static data, and then the residual program is the compiled version of the object program; so, the specialization of the interpreter yields compilation. 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 specialization of the interpreter mentioned above: 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.

An important notion in the program specialization approach is that of *inherited limit* [Mogensen, 1996; Mogensen, 1998]. An inherited limit is some limitation in the residual program imposed by the structure of the source program and the specialization method; that is, the *form* of obtainable generated programs is limited by the form of the program to be specialized. For example, the number of functions (supposed that each function in the source language be specialized in a unique way), the number of variables, the number of types, etc. Mogensen has argued that historical developments in program specialization gradually remove inherited limits, and suggest how this principle can be used as a guideline for further development [Mogensen, 1996].

One good way to detect the presence or absence of inherited limits is to specialize a self-interpreter and compare the residual programs with the source one: if they are essentially the same, then we can be confident that no inherited limits exist. We then say that the specialization was *optimal* (or *Jones-optimal*, after Neil Jones [Jones, 1988]). Partial evaluation, in the case of self-interpreters written in untyped languages, can obtain optimality; but for typed interpreters things are different. As partial evaluation works by reduction, the type of the residual program is restricted by that of the source one; thus, the residual code will contain type information coming from the representation of programs in the interpreter: optimality cannot be achieved, and the inherited limit of types is exposed. This problem was stated by Neil Jones in 1987 as one of the open problems in partial evaluation [Jones, 1988].

1.2 Principal Type Specialization

Type Specialization is a different form of program specialization introduced by John Hughes in 1996 as a solution for optimal specialization of typed interpreters. It has also proved to be a rich approach to program specialization. For example, it is possible to use the same interpreter for a given object language L to obtain automatically a compiler for both untyped and typed version of L with just a change on static or dynamic information [Hughes, 1998]. This cannot be obtained by traditional techniques of partial evaluation.

Types in programming languages capture different properties about expressions and this is the key to this approach: the information marked static in an expression will be moved into the residual type, expressing more detailed facts about expressions than source types; we need a more powerful residual type system for expressing that more detailed information. To illustrate this fact, we can think of an expression with type *Int*; the facts we know about it is that, if the computation of the expression finishes, the result will be an integer. But if we know more about the expression, for example that it is a constant 28, a better type associated to it is one capturing that information — let us call this type $\hat{28}$, and allow residual types to be extended with that kind of types. Having all the information in the type, there is no need to execute the program anymore, and thus, we can replace the integer constant by a dummy constant having type $\hat{28}$ — that is, the source expression $28 : \mathit{Int}$ can be specialized to $\bullet : \hat{28}$, where $\bullet$ is the dummy constant.

In the original formulation of *type specialization*, presented like a generalized form of type inference [Hughes, 1996], both the source and residual type systems are monomorphic, imposing an inhereted limit: the residual programs cannot be polymorphic, they cannot have more polymorphism than the source program. Other drawback is the lack of principality because of the monomorphic and non-syntax directed nature of the rules, which has important undesirable consequences. In [Martínez López and Hughes, 2004] and [Martínez López, 2004] these problems were fixed for a subset of the language presented by John Hughes. The inherited limit of polymorphism was removed and was proved that his *syntax directed* system has a notion of principality, called *principal specialization*. The specialization process was divided in two independent phases: constraint generation and constraint solving. The first phase tries to flow information as much as it can, and when there is some absent information which must flow from the code to the type, constraints play a crucial rôle. In the second phase, when this information is present, the right residual program can be calculated using heuristics [Badenes and Martínez López, 2002].

1.3 Contribution of this Work

The main contribution of this work is to extend the language described in [Martínez López and Hughes, 2004] and [Martínez López, 2004] to be able to manipulate dynamic sum-types. According to the definition of sum-types given by [Jones *et al.*, 1993], they are basically data types without names and recursion. However, we consider sum-types with names but without recursion — recursion is out of the scope of this work.

All formal system rules and proofs presented in [Martínez López and Hughes, 2004] and [Martínez López, 2004] are extended in order to incorporate dynamic sum-types to the language, and all proofs are completed to show that the notion of principality is preserved by the extension. With respect to the constraint solving phase, we leave formalization of rules involving dynamic sum-types as future work but provide an implementation instead.

Dynamic data types are needed to write interpreters to untyped languages under the *type specialization* and *principal type specialization* approaches. Martínez López' work has not yet been extended to obtain *principal type specialization* of dynamic data types and thus, we can consider this work as a small step towards the achievement of this goal.

1.4 Overview

This work is divided in five chapters.

In Chapter 2 we describe briefly the ideas behind specialization and explain the theory of qualified types developed by Mark Jones [Jones, 1994], which is the technical fundation used in [Martínez López and Hughes, 2004] and [Martínez López, 2004]. Additionally, we present Martínez López' reformulation of type specialization to obtain principality. In Chapter 3 we present our extension to his approach to deal with dynamic sum-types. After that, in Chapter 4 we develop the extension of the algorithm that obtains principal specializations. Then, in Chapter 5 we show some details of the extension that were made to the existing prototype implementing principal type specialization in the functional language Haskell [Peyton Jones and Hughes (editors), 1999]. Finally, in Chapter 6 we talk about future work and conclusions.

An appendix with formal proofs is given as an addition in order to allow a more smooth reading of this work.

Chapter 2

Specialization

"Tú pensabas de niño, que es mago aquel que puede hacer cualquier cosa. Eso pensé yo, alguna vez. Y todos nosotros. Y la verdad es que a medida que un hombre adquiere más poder y sabiduría, se le estrecha el camino, hasta que al fin no elige, y hace pura y simplemente lo que tiene que hacer"

La sombra en libertad
Un mago de Terramar
Úrsula K. Le Guin

In this chapter we sumarize the concepts behind *type specialization*, the theory of *qualified types* and *principal type specialization*.

2.1 Type Specialization

Type Specialization is an approach to program specialization introduced by John Hughes in 1996 [Hughes, 1996]. The main idea of type specialization is to specialize *both* the source program and its type to a residual program and residual type. In order to do this, instead of a generalized form of evaluation, type specialisation uses a generalized form of type inference.

2.1.1 Source Language

The source language we consider is a λ-calculus enriched with local definitions and arithmetic constants and operations. Furthermore, there are two kind of annotations for constructs: $\_^S$ or $\_^D$. We also have **lift**, **poly** and **spec**, which are another kind of annotation whose purpose will be explained later. So, the definition of the source language is the following.

Definition 2.1. Let x denote a *source term variable* from a countable 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:

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

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

It is expected that static terms be removed from the source program by computing and moving them into their residual types, while dynamic terms be kept in the residual code. Annotations are provided by the programer and are part of the input to the specializer — they cannot be calculated as in partial evaluation, more details can be found in [Hughes, 1996].

The construction **lift** is responsible for changing the information that has been moved to residual types back to residual code — see Example 2.6 — **poly** introduces *polyvariant* expressions, and **spec** — see example Example 2.9 — is applied to *polyvariant* terms in order to produce different specializations of the same source term.

Source types will reflect the static or dynamic nature of expressions — the type of constants, functions and operators will be consistent with the types of arguments. The source types are defined as follows.

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

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

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

This language is a small subset of the language of the type specializer from [Hughes, 1996], but contains enough constructs to illustrate the basic notions.

2.1.2 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 constructs. Residual terms are defined as follows.

Definition 2.3. Let x' denote a *residual term variable* from a countable infinite set of variables. A *residual term*, denoted by e', is an element of the language defined by the following grammar:

$$\begin{array}{rllll} e' ::= & x' & \mid n & \mid e' + e' & \mid \bullet \\ \mid & \lambda x'.e' & \mid e'@e' & \mid \textbf{let}\ x' = e'\ \textbf{in}\ e' & \\ \mid & (e'_1, \ldots, e'_n) & & \mid \pi_{n,n}\, e' & \end{array}$$

As in the source language, $(e'_1, \ldots, e'_n)$ is a finite tuple of e's for every possible arity n, and $\pi_{1,2}\, e'$ and $\pi_{2,2}\, e'$ may be abbreviated **fst** e' and **snd** e' respectively.

The expression $\bullet$ corresponds to the residual of static constants, the numbers n to the residual of dynamic numbers, lambda abstraction and application and let constructs are the residual of corresponding dynamic ones, and finally, tuples and projections corresponds to both the residual of tuples and the residual of polyvariant expressions and their specializations. It is important to mention that [Hughes, 1996] makes no distinction between static and dynamic tuples, so both the residual of dynamic tuples and the tuples introduced by polyvariance will be eliminated in a postprocessing phase called *arity raising*.

Residual types reflect the definition of source types.

DEFINITION 2.4. A *residual type*, denoted by τ', is an element of the language defined by the grammar:

$$\tau' ::= \mathit{Int} \mid \hat{n} \mid \tau' \to \tau' \mid (\tau', \ldots, \tau')$$

where $(\tau'_1, \ldots, \tau'_n)$ is a finite tuple of τ's for every possible arity n.

The novel feature of this language is the use of an infinite number of one-point types — being the one-point type $\hat{n}$ the residual type corresponding to some static integer whose value is known to be n.

2.1.3 Specializing

In order to express the result of the specialization procedure, [Hughes, 1996] introduced a new kind of judgment, and a system of rules to infer valid judgments. These judgments, similarly to typing judgments in the source language, make use of assignments to determine the specialization of free variables.

DEFINITION 2.5. A *specialization assignment*, denoted by Γ, is a (finite) list of *specialization statements* of the form $x : \tau \hookrightarrow e' : \tau'$, where no source variable appears more than once.

The specialization of a given program is expressed by type inference with a judgement of the form

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

which denotes that the program e with source type τ can be specialized to a residual program e' with residual type τ', under the hypothesis Γ (containing assumptions about the specialization of free variables).

Instead of showing the rules that specify the specialization process, which can be found in [Hughes, 1996], we explain through examples its capability and limits that motivate the need for principality.

2.1.4 Examples

EXAMPLE 2.6. Observe how every expression annotated as dynamic appears in the residual term (in fact, we have that a fully dynamic expression, that is one in which every annotation is $\_^{D}$, specializes to a copy of itself with the annotations removed).

1. $\vdash 42^{D} : \mathit{Int}^{D} \hookrightarrow 42 : \mathit{Int}$
2. $\vdash 42^{S} : \mathit{Int}^{S} \hookrightarrow \bullet : \hat{42}$
3. $\vdash (2^{D} +^{D} 1^{D}) +^{D} 1^{D} : \mathit{Int}^{D} \hookrightarrow (2+1)+1 : \mathit{Int}$
4. $\vdash (2^{S} +^{S} 1^{S}) +^{S} 1^{S} : \mathit{Int}^{S} \hookrightarrow \bullet : \hat{4}$
5. $\vdash \textbf{lift}\ (2^{S} +^{S} 1^{S}) +^{D} 1^{D} : \mathit{Int}^{D} \hookrightarrow 3+1 : \mathit{Int}$

Also observe in 5 how the use of **lift** allows us to cast a static integer into a dynamic one, thus inserting the result of the static computation back into the residual term.

Example 2.7. Assignments provide the information for the specialization of free variables, which allows the specialization of functions.

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

In 3 there is information that was moved from the context of the function to the function's body, where **lift** places it.

Example 2.8. One feature of type specialization is that there exist correctly annotated terms that cannot be specialized; consider

$$\begin{array}{l}\textbf{let}^D\ f = \lambda^D x.\textbf{lift}\ x +^D 1^D \\ \ \textbf{in}\ (f \mathbin{@^D} 42^S, f \mathbin{@^D} 17^S)^D\ : (Int^D, Int^D)^D.\end{array}$$

As we have seen in Example 2.7-3, the body of the function is specialized according to the parameter, but f has two *different* parameters!. In order to allow f to be specialized in more than one way, we must use the annotation **poly**.

Example 2.9. Observe the use of **poly** in the definition of f (and how that annotation produces a tuple for the definition of f' in the residual codes of both specializations), and the use of **spec** in every application of f to an argument (and how that produces the corresponding projections).

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

2. $$\begin{array}{l}\vdash \textbf{let}^D\ f = \textbf{poly}\ (\lambda^D x.\textbf{lift}\ x +^D 1^D) \\ \quad \textbf{in}\ (\textbf{spec}\ f \mathbin{@^D} 42^S, \textbf{spec}\ f \mathbin{@^D} 17^S)^D\ : (Int^D, Int^D)^D \hookrightarrow \\ \quad \textbf{let}\ f' = (\lambda x'.17 + 1, \lambda x'.55 + 1, \lambda x'.42 + 1) \\ \quad \textbf{in}\ (\pi_{3,3}\ f'@\bullet, \pi_{1,3}\ f'@\bullet) : (Int, Int)\end{array}$$

The size and order of the residual tuple is arbitrary, provided that it has at least two elements ($\lambda x'.42 + 1$ and $\lambda x'.17 + 1$), and that the projections select the appropriate element, as can be seen when contrasting specialization 2 against specialization 1. In this way, we can obtain a potentially infinite number of specializations from a given source program.

The following example show the problems that were solved in [Martínez López and Hughes, 2004].

Example 2.10. Observe that in all cases there is some static information missing.

1. $\lambda^D x.x +^S 1^S\ : Int^S \to^D Int^S$

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

All have many different *unrelated* specializations! For example, the function in Example 2.10-1 has one specialization for each possible value for x — in particular, $\lambda x'.\bullet : \hat{n} \rightarrow \hat{n}'$, for every value of n and n' such that $n' = n + 1$. If this function appears in one module, but is applied in another one, then the specialization should wait until the value n of the argument is known in order to decide its residual type. The same problem appears in the case of polyvariance where the generation of the tuple or the selection of the right projection should be deferred until all the information is available. This problem is called *lack of principality* and fixing it requires a big change in the residual language.

2.2 Qualified Types

The theory of qualified types [Jones, 1994] is a framework that allows the development of constrained type systems in an intermediate level between monomorphic and polymorphic type disciplines.

Martínez López used this concept deeply when he reformulated *type specialization* to obtain principality [Martínez López and Hughes, 2004].

Qualified types can be seen in two ways: either as a restricted form of polymorphism, or as an extension of the use of monotypes (commonly described as *overloading*, in which a function may have different interpretations according to the types of its arguments). Predicates are used to restrict the use of type variables, which are allowed as a type.

The theory explains how to enrich types with predicates, how to perform type inference using the enriched types, and which are the minimal properties that predicates must satisfy in order for the resulting type system to have similar properties as the Hindley-Milner one [Milner, 1978]. In particular, it has been shown that any well typed program has a *principal type* that can be calculated by an extended version of Milner's algorithm.

2.2.1 Predicates and Entailment

Polymorphism is the ability to treat some terms as having many different types. We can express a polymorphic type by means of a *type scheme* [Damas and Milner, 1982], using universal quantification to abstract those parts of a type that may vary. That is, if $f(t)$ is a type for every possible value of type variable t, then giving the type scheme $\forall t.f(t)$ to a term means that the term can receive any of the types in the set

$$\{f(\tau) \text{ s.t. } \tau \text{ is a type}\}$$

But sometimes that is not enough: not all the types can replace t and still express a possible type for the term. For those cases, a form of restricted quantification can be used. If $P(t)$ is a predicate on types, we use the type scheme $\forall t.P(t) \Rightarrow f(t)$ to represent the set of types

$$\{f(\tau) \text{ s.t. } \tau \text{ is a type and } P(\tau) \text{ holds}\}$$

and accurately reflect the desired types for a given term.

The key feature in the theory is the use of a language of *predicates* to describe sets of types (or, more generally, relations between types). The exact set of predicates may vary from one application to another — predicates that we use are described in Section 2.3.1.

In [Jones, 1994], Mark Jones uses several notations of conventions, in particular regarding lists of elements, lists of pairs, the usual operations on lists, and their use on type expressions.

NOTATION 2.11. Let L and L' be any kind of (finite) lists or sets, and l be an element. We write L, L' for the result of the *union* of sets L and L' or the *append* of lists L and L'. We write l, L for the result of the inclusion of element l to set L or the *cons* of l to list L. Finally, we write $\emptyset$ for the empty set or list, and assume that $\emptyset, L = L, \emptyset = L$.

As a consequence of all these conventions, a use of l could represent an element or a singleton list, depending on the context.

Another point where conventions are convenient is when working with qualified types; there are two possible ways to add predicates to the basic syntax: one by one, or all toghether in a set. For example, this amounts to define either that $\rho ::= \delta \Rightarrow \rho \mid \tau$ or $\rho ::= \Delta \Rightarrow \tau$, being Δ the list of predicates $\delta_1, \ldots, \delta_n$. We will use the first form as the definition and the second as an abbreviation (although the other way is also possible).

NOTATION 2.12. Assuming that a list of predicates $\Delta = \delta_1, \cdots, \delta_m$, a list of type variables $\alpha = \alpha_1, \cdots, \alpha_m$, a list of evidence variables $h = h_1, \cdots, h_m$, and a list of evidence expressions $v = v_1, \cdots, v_m$, we use the abbreviations:

Object	Expression	Abbreviation(s)
Qualified type	$\delta_1 \Rightarrow \cdots \delta_m \Rightarrow \tau'$	$\Delta \Rightarrow \tau'$
Type scheme	$\forall\alpha_1. \cdots \forall\alpha_m.\rho$	$\forall\alpha.\rho$
Evidence abstr.	$\Lambda h_1. \cdots \Lambda h_m.e'$	$\Lambda h.e'$
Evidence app.	$((e'((v_1))) \cdots)((v_m))$	$e'((v))$

In the special case when $m = 0$, all the sequences are empty, and then the abbreviations stand for the enclosed element (e.g. $e'((v))$ represents e'). This implies, for example, that a type τ can be understood as a qualified type ($\emptyset \Rightarrow \tau$) or a type scheme ($\forall\emptyset.\emptyset \Rightarrow \tau$) depending on the context of use.

Another convention is concerned with lists of pairs.

NOTATION 2.13. Lists of pairs may be abbreviated by a pair of lists in the following way. If $h = h_1, \ldots, h_n$ and $\Delta = \delta_1, \ldots, \delta_n$, the list $h_1 : \delta_1, \ldots, h_n : \delta_n$ may be abbreviated as $h : \Delta$ or as Δ depending on the context. The latter is also used for a list of predicates — no explicit remotion of the variables (first components of pairs) will be used.

The union (concatenation) of two sets (lists) of pairs $h : \Delta$ and $h' : \Delta'$ will be denoted $h : \Delta, h' : \Delta'$ (as an alternative to $h, h' : \Delta, \Delta'$, which may also be used).

Despite the possible source of confusion that these conventions may be for a casual reader, they can be easily mastered with very little practice.

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

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

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

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

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

Figure 2.1: Structural laws satisfied by entailment.

The minimun required properties of predicates are captured by using an entailment relation ($\Vdash$) between (finite) sets of predicates satisfying a few simple laws. The judgement $\Delta_1 \Vdash \Delta_2$ means that predicates belonging to Δ_1 can be used to construct evidence for all the predicates in Δ_2.

The basic properties that entailment must satisfy are:

Monotonicity: $\Delta \Vdash \Delta'$ whenever $\Delta \supseteq \Delta'$

Transitivity: if $\Delta \Vdash \Delta'$ and $\Delta' \Vdash \Delta''$, then $\Delta \Vdash \Delta''$

Closure property: if $\Delta \Vdash \Delta'$, then $S\Delta \Vdash S\Delta'$.

The last condition is needed to ensure that the system of predicates is compatible with the use of parametric polymorphism. These properties can be expressed by a system containing the rules in Figure 2.1. Observe that if $\delta \in \Delta$, then $\Delta \Vdash \delta$ by monotonicity of $\Vdash$ — $\Delta \Vdash \{\delta\}$ is also written as $\Delta \Vdash \delta$ by virtue of Notation 2.11.

2.2.2 Type inference with Qualified Types

In the theory of qualified types, the language of types and type schemes is stratified in a similar way as in the Hindley-Milner system, where the most important restriction is that qualified or polymorphic types cannot be argument of functions; that is, types (denoted by τ) are defined by a grammar with at least these productions $\tau \ ::= \ t \mid \tau \to \tau$. On top of types are constructed qualified types of the form $\Delta \Rightarrow \tau$ (denoted by ρ), and then type schemes of the form $\forall\{\alpha_i\}.\rho$ (denoted by σ). We use freely the conventions defined in Notation 2.12. Using that notation, any type scheme can be written in the form $\forall\alpha_i.\Delta \Rightarrow \tau$, representing the set of qualified types

$$\{\Delta[\alpha_i/\tau_i] \Rightarrow \tau[\alpha_i/\tau_i] \text{ s.t. } \tau_i \text{ is a type}\}$$

The language of terms — denoted by e — is based on the untyped λ-calculus (it has, at least, variables, applications, abstractions, and the **let** construct); it is called OML,

$$\text{(QIN)}\quad \frac{\Delta, \delta \mid \Gamma \vdash e : \rho}{\Delta \mid \Gamma \vdash e : \delta \Rightarrow \rho}$$

$$\text{(QOUT)}\quad \frac{\Delta \mid \Gamma \vdash e : \delta \Rightarrow \rho \qquad \Delta \Vdash \delta}{\Delta \mid \Gamma \vdash e : \rho}$$

$$\text{(GEN)}\quad \frac{\Delta \mid \Gamma \vdash e : \sigma}{\Delta \mid \Gamma \vdash e : \forall\alpha.\sigma}\ (\alpha \notin FV(\Delta) \cup FV(\Gamma))$$

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

Figure 2.2: Related variables and predicates typing rules for OML.

abbreviating 'Overloaded ML'. Type inference uses judgements extended with a context of predicates

$$\Delta \mid \Gamma \vdash e : \sigma$$

representing the fact that when the predicates in Δ are satisfied, and the types of the free variables of e are as specified by Γ, then the term e has type σ. The type system is in essence similar to a Hindley-Milner one, with the rules showed in Figure 2.2 added. The rules (GEN) and (INST) are used to generalize and instantiate type variables in type schemes, and (QIN) and (QOUT) to manage predicates.

DEFINITION 2.14. A *constrained type scheme* is an expression of the form $(\Delta \mid \sigma)$ where Δ is a set of predicates and σ is a type scheme.

In order to find all the ways in which a particular e can be used within a given Γ, the theory have to deal with sets of the form

$$\{(\Delta \mid \sigma) \text{ s.t. } \Delta \mid \Gamma \vdash e : \sigma\}$$

The main tool used to deal with these sets is a preorder $\geq$ — pronounced *more general* — defined on pairs of constrained type schemes, and whose intended meaning is that if $(\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ then it is possible to use an object which can be treated as having 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'. To define it formally, the notion of *generic instance* is needed.

DEFINITION 2.15. A *qualified type* $\Delta_\tau \Rightarrow \tau$ is a *generic instance* of the constrained type scheme $(\Delta \mid \forall\alpha_i.\Delta' \Rightarrow \tau')$ if there are types τ_i such that

$$\Delta_\tau \Vdash \Delta, \Delta'[\alpha_i/\tau_i] \text{ and } \tau = \tau'[\alpha_i/\tau_i]$$

In particular, a qualified type $\Delta \Rightarrow \tau$ is instance of another qualified type $\Delta' \Rightarrow \tau'$ if and only if $\Delta \Vdash \Delta'$ and $\tau = \tau'$. Now we are in position to define the "more general" ordering ($\geq$) on constrained type schemes.

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

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

Figure 2.3: Translation rules from OML to OP that involves predicates.

Definition 2.16. The constrained type scheme $(\Delta \mid \sigma)$ is said to be *more general* than the constrained type scheme $(\Delta' \mid \sigma')$, written $(\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, if every generic instance of $(\Delta' \mid \sigma')$ is a generic instance of $(\Delta \mid \sigma)$.

2.2.3 Coherence and Evidence

In order to give semantics to the terms in the system, [Jones, 1994] introduces the notion of *evidence*, and provide a translation from the original language of terms, OML, to one manipulating evidence explicitely — called OP, for 'Overloaded Polymorphic λ-calculus'. The essential idea is that an object of type $\Delta \Rightarrow \tau$ can only be used if it is supplied with suitable evidence that predicates in Δ do indeed hold. The treatment of evidence can be ignored in the basic typing algorithm, but is essential to provide *coherence*, that is, the meaning of a term does not depend on the way it is typechecked [Breazu-Tannen *et al.*, 1991]. The properties of predicate entailment must be extended to deal with predicate assignments and evidence expressions (in particular, the rules given in Figure 2.1 already contained this extension, where h denote an evidence variable, and v denotes an evidence expression). Observe that we are using the conventions introduced in Notation 2.13, so predicate assignments are written as $h : \Delta$ meaning $h_1 : \delta_1, \ldots, h_n : \delta_n$, and similarly for $v : \Delta$.

Unfortunately, there exist OML terms for which the translation gives more than one non-equivalent term, showing that the meaning of those OML terms depend in the way they are typed. In order to characterize terms with a unique meaning when possible, OP typings have to be studied; thus, reduction and equality of OP terms are defined, and then, the central notion of conversion is provided. A *conversion* from σ to σ' is a collection of OP terms that allow the transformation of any OP term of type σ into an OP term of type σ' by manipulating evidence; this is an extension of the notion of $\geq$ defined before. The motivation for using this notion is that an important property of the ordering relation $\geq$ used to compare types in OML breaks down in OP, due to the presence of evidence: a term with a general type can be used as having an instance of that type only after adjusting the evidence it uses.

The definition of conversions extends the definition of $\geq$ (Definition 2.16) with the treatment of evidence.

Definition 2.17. 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 the β_i appears free in σ, Δ, or Δ'. A closed OP term C of type $(\Delta \mid \sigma) \rightarrow (\Delta' \mid \sigma')$, such that erasing all evidence from it returns the identity function, is called a *conversion* from $(\Delta \mid \sigma)$ to $(\Delta' \mid \sigma')$, written $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, if there are types τ_i, evidence variables h' and h'_τ, and evidence expressions v and v' such that:

- $\tau' = \tau[\alpha_i/\tau_i]$
- $h' : \Delta', h'_\tau : \Delta'_\tau \Vdash v : \Delta, v' : \Delta_\tau[\alpha_i/\tau_i]$, and
- $C = (\lambda x.\Lambda h', h'_\tau.x((v))((v')))$

Conversions are only used in the theory of qualified types to relate different translations for the same term.

Finally, Jones define the notions of *simplification* and *improvement* on predicate sets. These notions will also appear in principal type specialization.

2.3 Principal Type Specialization

As we saw in Section 2.1, Hughes' formulation of type specialization forces an algorithm to wait until all the context is known before making any attempt to specialize a given expression [Hughes, 1996]. The problem, called by us *lack of principality*, is very similar to the problem appearing in simply typed λ-calculus when typing an expression like $\lambda x.x$, where the type of x is determined by the context of use — different typings for this expression have no relation between them expressible in the system. The solution to the latter is to extend the type language in order to allow polymorphism — by introducing type variables — and defining a notion of instantiation for types.

Martínez López' work finds *principal type specializations* for each term in the source language such that every other valid specialization of this term can be obtained by instantiation of it [Martínez López and Hughes, 2004]. Thus, specialization can be done in isolation, without any context. A first step in this direction is to use residual type variables to defer the specialization of expressions depending on the context. However, this is not enough, as subtle dependencies between types (as the relation between n and n' in the specialization of Example 2.10-1), cannot be expressed. The *theory of qualified types*, briefly described in Section 2.2, presents a type framework that allows expressing conditions relating universally quantified variables [Jones, 1994].

2.3.1 Residual Language

Extending the residual type language with predicates implies that the residual term language must also be extended to manipulate evidence. The extensions have two parts: the "structural" components taken from the theory of qualified types, and the particular constructs needed to express specialization features.

Following the theory of qualified types, the residual type language is extended with type variables (t), and the syntactic categories of qualified types (ρ) and type schemes (σ); also particular predicates (δ) are defined. The most important innovations with respect to the theory of qualified types are the new type construct **poly** σ, and the use of scheme variables (s), both used to express polyvariance.

Definition 2.18. Let t denote a *type variable* from an countable infinite set of variables, and s a *type scheme variable* from another countable infinite set of variables, both

$$\begin{array}{rl}
(\beta_v) & (\Lambda h.e'_1)((v)) \rhd e'_1[h/v] \\[1em]
(\eta_v) & \Lambda h.e'_1((h)) \rhd e'_1 \quad (h \notin EV(e'_1)) \\[1em]
(\mathbf{let}_v) & \mathbf{let}_v\ x = e'_1\ \mathbf{in}\ e'_2 \rhd e'_2[x/e'_1] \\[1em]
(\circ_v) & (v_1 \circ v_2)[e'] \rhd v_1[v_2[e']] \\
(\mathbf{if}_v\text{-}\mathit{True}) & \mathbf{if}_v\ \mathit{True}\ \mathbf{then}\ e'_1\ \mathbf{else}\ e'_2 \rhd e'_1 \\
(\mathbf{if}_v\text{-}\mathit{False}) & \mathbf{if}_v\ \mathit{False}\ \mathbf{then}\ e'_1\ \mathbf{else}\ e'_2 \rhd e'_2
\end{array}$$

Figure 2.4: Reduction for residual terms.

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

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

The residual term language is extended with evidence (v), including evidence variables (h), evidence abstractions $(\Lambda h.e')$, and evidence applications $(e'((v)))$. Evidence is very important in this formulation of type specialization because it allows us to abstract differences among different residual terms of a given source term, and is one of the cornerstones for the principality result. 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. Observe that conversions, denoted by C, are defined separately from other elements in the language, and that they are contexts — instead of (families of) terms, as in [Jones, 1994].

DEFINITION 2.19. A *residual term*, denoted by e', is an element of the language defined by the following grammar:

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

We will be working under a equivalence $=$ on residual terms, defined as the minimal relation that contains α-conversions for both λ and Λ-abstractions, and rules in Figure 2.4. Equivalence is also extended to conversions, defining that $C = C'$ if for all expressions e', $C[e'] = C'[e']$. The meaning of $\Vdash$, whose structural propierties are given in Figure 2.1, is completed with the rules that show, in Figure 2.5, how the evidence for particular predicates is constructed. The predicate IsInt is provable when the type is a one-point

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

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

$$\text{(IsOpIsInt)}\quad \Delta, h : \tau' := \tau_1' + \tau_2', \Delta' \Vdash h : \text{IsInt}\ \tau'$$

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

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

Figure 2.5: Entailment for evidence construction.

type representing a number and the evidence is the value of that number. Similarly, the predicate $\_ := \_ + \_$ is provable when 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 that addition. The predicate IsMG internalizes the ordering $\geq$ and the evidence is the corresponding conversion. The composition of evidence used in this rule was defined in Figure 2.4.

2.3.2 Residual Types

As we discussed in Section 2.2.3, the relation between different types and scheme types is expressed by $\geq$. We define conversions as a special kind of contexts, rather than as terms in the residual language. Additionally, we use conversions as part of the evidence language, which prove some kind of predicates with a similar semantics to relation $\geq$. This evidence will be applied to terms, so we need to slightly modify the definition of conversion as follows:

Definition 2.20. 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 the β_i appears free in σ, $h : \Delta$, or $h' : \Delta'$. 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 h_τ and h_τ', and evidence expressions v and v' such that:

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

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 top level of e'.

The following assertions hold when $\sigma, \sigma', \sigma''$ are 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 $C' \circ C : (\Delta \mid \sigma) \geq (\Delta'' \mid \sigma'')$

Example 2.21. Conversions are used to adjust the evidence demanded by different type schemes. For all Δ it holds that

1. $[]((42)) : (\Delta \mid \forall t.\text{IsInt } t \Rightarrow t \rightarrow \mathit{Int}) \geq (\Delta \mid \hat{42} \rightarrow \mathit{Int})$

2. $C : (\Delta \mid \forall t_1, t_2.\text{IsInt } t_1, \text{IsInt } t_2 \Rightarrow t_1 \rightarrow t_2) \geq (\Delta \mid \forall t.\text{IsInt } t \Rightarrow t \rightarrow t)$
   where $C = \Lambda h.[]((h))((h))$

3. $\Lambda h.[] : (\Delta \mid \hat{42} \rightarrow \mathit{Int}) \geq (\Delta \mid \forall t.\text{IsInt } t \Rightarrow \hat{42} \rightarrow \mathit{Int})$

In [Martínez López and Hughes, 2004], a type system to infer the type of a residual expresion is also presented, proving later that specialization is well behaved with respect to it. This system of rules is not able to infer the type of any residual expresion — only to check it. The expressions that will be verified are those that come from the specialization process: the programmer does not write any piece of residual code; instead, the specialization obtains the residual code together with its residual type. This last remark justifies why it is reasonable to provide the form of higher-order polymorphism, controlled by annotations **poly** and **spec**.

2.3.3 Specifying Principal Type Specialization

The system specifying type specialisation is composed by two sets of rules.

The first one relates source types with residual types, expressing which residual types can be obtained by specialising a given source one. This system, which is called SR — see Figure 2.6 — is important because it is needed to restrict the possible choices of residuals for bound variables when specializing lambda-abstractions and specializations of polyvariant expressions; without these restrictions we can obtain more specializations than expected — for example the source term $\lambda^D x.x : \mathit{Int}^S \rightarrow^D \mathit{Int}^S$ can be specialized to $\lambda x'.x' : \mathit{Bool} \rightarrow \mathit{Bool}$.

The second one is the specialization process itself and appears in Figures 2.7 and 2.8. Judgements have the structure $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$, expanding notions explained in Section 2.1 for the use of qualified types.

Rules (QIN) and (QOUT) incorporate the notion of evidence introduced in Section 2.2.3 and allow us to move information from the context (as predicates) into the residual terms (adding predicates in the type and abstracting evidence) and back. Observe that these are dual rules, so it is possible to eliminate the effect produced by one of them by using the other one.

We revisit here some examples of Section 2.1, but now using the rules that appear in Figures 2.7 and 2.8.

Example 2.22. The source term in Example 2.10-1 can now be specialized as follows

$$\vdash_{\text{P}} \lambda^D x.x +^S 1^S : \mathit{Int}^S \rightarrow^D \mathit{Int}^S \hookrightarrow \Lambda h_t, h_{t'}.\lambda x'.h_{t'} : \forall t, t'.\text{IsInt } t, t' := t + \hat{1} \Rightarrow t \rightarrow t'$$

Observe the use of evidence abstractions to wait for the residual of static information. This is one of the keys allowing principal specialisation. The evidence will be the numbers corresponding to the static values of x and resulting operations.

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

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

$$\text{(SR-DFUN)} \quad \frac{\Delta \vdash_{\text{SR}} \tau_1 \hookrightarrow \tau'_1 \qquad \Delta \vdash_{\text{SR}} \tau_2 \hookrightarrow \tau'_2}{\Delta \vdash_{\text{SR}} \tau_2 \rightarrow^D \tau_1 \hookrightarrow \tau'_2 \rightarrow \tau'_1}$$

$$\text{(SR-TUPLE)} \quad \frac{(\Delta \vdash_{\text{SR}} \tau_i \hookrightarrow \tau'_i)_{i=1,..,n}}{\Delta \vdash_{\text{SR}} (\tau_1, \ldots, \tau_n)^D \hookrightarrow (\tau'_1, \ldots, \tau'_n)}$$

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

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

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

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

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

Figure 2.6: Rules defining the source-residual relationship.

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

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

$$\text{(D+)}\quad \frac{(\Delta \mid \Gamma \vdash_{\text{P}} e_i : \mathit{Int}^{D} \;\hookrightarrow\; e'_i : \mathit{Int})_{i=1,2}}{\Delta \mid \Gamma \vdash_{\text{P}} e_1 +^{D} e_2 : \mathit{Int}^{D} \;\hookrightarrow\; e'_1 + e'_2 : \mathit{Int}}$$

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

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

$$\text{(S+)}\quad \frac{(\Delta \mid \Gamma \vdash_{\text{P}} e_i : \mathit{Int}^{S} \;\hookrightarrow\; e'_i : \tau'_i)_{i=1,2} \qquad \Delta \Vdash v : \tau' := \tau'_1 + \tau'_2}{\Delta \mid \Gamma \vdash_{\text{P}} e_1 +^{S} e_2 : \mathit{Int}^{S} \;\hookrightarrow\; \bullet : \tau'}$$

$$\text{(DTUPLE)}\quad \frac{(\Delta \mid \Gamma \vdash_{\text{P}} e_i : \tau_i \;\hookrightarrow\; e'_i : \tau'_i)_{i=1,..,n}}{\Delta \mid \Gamma \vdash_{\text{P}} (e_1, \ldots, e_n)^{D} : (\tau_1, \ldots, \tau_n)^{D} \;\hookrightarrow\; (e'_1, \ldots, e'_n) : (\tau'_1, \ldots, \tau'_n)}$$

$$\text{(DPRJ)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e : (\tau_1, \ldots, \tau_n)^{D} \;\hookrightarrow\; e' : (\tau'_1, \ldots, \tau'_n)}{\Delta \mid \Gamma \vdash_{\text{P}} \pi^{D}_{i,n}\, e : \tau_i \;\hookrightarrow\; \pi_{i,n}\, e' : \tau'_i}$$

$$\text{(DLAM)}\quad \frac{\Delta \mid \Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2 \vdash_{\text{P}} e : \tau_1 \;\hookrightarrow\; e' : \tau'_1 \qquad \Delta \vdash_{\text{SR}} \tau_2 \;\hookrightarrow\; \tau'_2}{\Delta \mid \Gamma \vdash_{\text{P}} \lambda^{D} x.e : \tau_2 \rightarrow^{D} \tau_1 \;\hookrightarrow\; \lambda x'.e' : \tau'_2 \rightarrow \tau'_1}\ (x' \textit{ fresh})$$

$$\text{(DAPP)}\quad \frac{\Delta \mid \Gamma \vdash_{\text{P}} e_1 : \tau_2 \rightarrow^{D} \tau_1 \;\hookrightarrow\; e'_1 : \tau'_2 \rightarrow \tau'_1 \qquad \Delta \mid \Gamma \vdash_{\text{P}} e_2 : \tau_2 \;\hookrightarrow\; e'_2 : \tau'_2}{\Delta \mid \Gamma \vdash_{\text{P}} e_1 \mathbin{@^{D}} e_2 : \tau_1 \;\hookrightarrow\; e'_1 @ e'_2 : \tau'_1}$$

$$\text{(DLET)}\quad \frac{\begin{array}{c}\Delta \mid \Gamma \vdash_{\text{P}} e_2 : \tau_2 \;\hookrightarrow\; e'_2 : \tau'_2 \\ \Delta \mid \Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2 \vdash_{\text{P}} e_1 : \tau_1 \;\hookrightarrow\; e'_1 : \tau'_1\end{array}}{\Delta \mid \Gamma \vdash_{\text{P}} \textbf{let}^{D}\ x = e_2\ \textbf{in}\ e_1 : \tau_1 \;\hookrightarrow\; \textbf{let}\ x' = e'_2\ \textbf{in}\ e'_1 : \tau'_1}\ (x' \textit{ fresh})$$

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

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

Figure 2.7: Specialisation rules (first part)

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

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

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

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

Figure 2.8: Specialisation rules (second part).

Example 2.23. The expression in Example 2.10-2 is a polyvariant term that cannot be specialized by Hughes' formulation [Hughes, 1996], but in this other approach its principal specialization is:

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

In this example, the polyvariant function is abtracted by conversion h, which abstracts the evidence of the type $(\forall t.\mathrm{IsInt}\ t \Rightarrow t \rightarrow \mathit{Int})$, being more general than any possible instance for s. At the same time, the real value of x in each possible expression obtained from this polivariant function is abstracted using the evidence variable h_x.

Example 2.24. In this example, the same polyvariant function that appears in the previous example is instantiated twice, receiving different static information each time.

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

Observe the interaction between annotations **poly** and **spec**, which introduces abstraction and application of evidence corresponding to the values that variable x assumes.

Example 2.25. Finally, we show how Example 2.10-3 is specialized, obtaining:

$$\begin{array}{l} \vdash_{\mathrm{P}} \lambda^{D} f . \mathbf{spec}\ f \ @^{D}\, 13^{S} : \mathbf{poly}\ (\mathit{Int}^{S} \rightarrow^{D} \mathit{Int}^{D}) \rightarrow^{D} \mathit{Int}^{D} \\ \quad \hookrightarrow \Lambda h_u, h_l . \lambda f' . h_l[f']@\bullet : \forall s . \mathrm{IsMG}\ (\forall t . \mathrm{IsInt}\ t \Rightarrow t \rightarrow \mathit{Int})\ s, \\ \qquad\qquad\qquad\qquad\qquad \mathrm{IsMG}\ s\ (\hat{13} \rightarrow \mathit{Int}), \Rightarrow \mathbf{poly}\ s \rightarrow \mathit{Int} \end{array}$$

This example shows a higher-order function that recieves a polyvariant function as its argument to apply it to a specific static value. The argument function must be an instance of $\forall t.\text{IsInt } t \Rightarrow t \rightarrow \mathit{Int}$, but at least as general as $\hat{13} \rightarrow \mathit{Int}$. These conditions are represented with evidence variables, which manipulate the residual code according to evidence that prove predicates when we determine the value of s.

The system $\vdash_{\text{P}}$ is stable under substitutions, an essential property for principality.

Proposition 2.26. *If* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$ *then* $S\,\Delta \mid S\,\Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : S\,\sigma$.

Specialization also respects the $\Vdash$ relation.

Proposition 2.27. *If* $h : \Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \tau'$ *and* $h' : \Delta' \Vdash v : \Delta$*, then* $h' : \Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e'[v/h] : \tau'$

2.3.4 Algorithm

Additionally to the specification rules given in Figures 2.7 and 2.8, Martínez López presents an algorithm for principal specialization, previously defining a *syntax directed* system similar to that used in [Jones, 1994]. This algorithm is based on the Milner's W algorithm [Milner, 1978], and the rules can be interpreted as an attribute grammar [Rèmy, 1989]. The system of rules is showed in Figures 2.9 and 2.10. This algorithm uses a number of auxiliaries subsystems which can be summarized as follows:

Unification: The unification algorithm is based on Robinson's algorithm, with modifications to deal with substitution under quantification (that is, inside polyvariant residual types). We use a kind of "*skolemisation*" of quantified variables to avoid substituting them — in order to do this, we extend residual type schemes with *skolem* constants, ranging over c, and belonging to a countable infinite set with no intersection with other variables. In order to specify the unification algorithm, we use a system of rules to derive judgments of the form $\sigma_c \sim^U \sigma_c$, with U ranging over substitutions. The rules are presented in Figure 2.11.

Entailment: The idea of an algorithm for entailment is to calculate a set of predicates that should be added to the current predicate assignment Δ in order to be able to entail a given predicate δ. The input is the current predicate assignment and the predicate δ to entail, and the output is the set of predicates to add and the evidence proving δ. The result can be easily achieved by adding δ to Δ with a new variable h. So, the only rule that is necessary for this algorithm is

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

that is, generate a new fresh variable h and add $h : \delta$ to the current predicate assignment.

More refined algorithms can be designed to handle ground predicates (such as $\text{IsInt } \hat{n}$) or predicates already appearing in Δ, but all these cases can be handled by simplification and constraint solving phases [Badenes and Martínez López, 2002].

$$\text{(W-VAR)}\quad \frac{x : \tau \hookrightarrow e' : \tau' \in \Gamma}{\emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} x : \tau \;\hookrightarrow\; e' : \tau'}$$

$$\text{(W-DINT)}\quad \emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} n^{D} : \mathit{Int}^{D} \;\hookrightarrow\; n : \mathit{Int}$$

$$\text{(W-D+)}\quad \frac{\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \mathit{Int}^{D} \;\hookrightarrow\; e_1' : \mathit{Int} \qquad \Delta_2 \mid S_2\,(S_1\,\Gamma) \vdash_{\mathrm{W}} e_2 : \mathit{Int}^{D} \;\hookrightarrow\; e_2' : \mathit{Int}}{S_2\,\Delta_1, \Delta_2 \mid S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 +^{D} e_2 : \mathit{Int}^{D} \;\hookrightarrow\; e_1' + e_2' : \mathit{Int}}$$

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

$$\text{(W-SINT)}\quad \emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} n^{S} : \mathit{Int}^{S} \;\hookrightarrow\; \bullet : \hat{n}$$

$$\text{(W-S+)}\quad \frac{\begin{array}{c}\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \mathit{Int}^{S} \;\hookrightarrow\; e_1' : \tau_1' \\ \Delta_2 \mid S_2\,(S_1\,\Gamma) \vdash_{\mathrm{W}} e_2 : \mathit{Int}^{S} \;\hookrightarrow\; e_2' : \tau_2' \\ \Delta \mid S_2\,\Delta_1, \Delta_2 \Vdash_{\mathrm{W}} v : t := S_2\,\tau_1' + \tau_2'\end{array}}{\Delta, S_2\,\Delta_1, \Delta_2 \mid S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 +^{S} e_2 : \mathit{Int}^{S} \;\hookrightarrow\; \bullet : t}\ (t\ \mathit{fresh})$$

$$\text{(W-DLAM)}\quad \frac{\Delta \vdash_{\text{W-SR}} \tau_2 \;\hookrightarrow\; \tau_2' \qquad \Delta' \mid S\,(\Gamma, x : \tau_2 \hookrightarrow x' : \tau_2') \vdash_{\mathrm{W}} e : \tau_1 \;\hookrightarrow\; e' : \tau_1'}{\Delta', S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} \lambda^{D} x.e : \tau_2 \rightarrow^{D} \tau_1 \;\hookrightarrow\; \lambda x'.e' : S\,\tau_2' \rightarrow \tau_1'}\ (x'\ \mathit{fresh})$$

$$\text{(W-DAPP)}\quad \frac{\begin{array}{c}\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \tau_2 \rightarrow^{D} \tau_1 \;\hookrightarrow\; e_1' : \tau_1' \\ \Delta_2 \mid S_2\,(S_1\,\Gamma) \vdash_{\mathrm{W}} e_2 : \tau_2 \;\hookrightarrow\; e_2' : \tau_2' \qquad S_2\,\tau_1' \sim^{U} \tau_2' \rightarrow t\end{array}}{U S_2\,\Delta_1, U\,\Delta_2 \mid U S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 @^{D} e_2 : \tau_1 \;\hookrightarrow\; e_1' @ e_2' : U\,t}\ (t\ \mathit{fresh})$$

$$\text{(W-POLY)}\quad \frac{\begin{array}{c}h : \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \;\hookrightarrow\; e' : \tau' \\ \Delta' \mid \emptyset \Vdash_{\mathrm{W}} v : \mathrm{IsMG}\ (\mathrm{Gen}_{S\,\Gamma}(\Delta \Rightarrow \tau'))\ s\end{array}}{\Delta' \mid S\,\Gamma \vdash_{\mathrm{W}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \;\hookrightarrow\; v[\Lambda h.e'] : \mathbf{poly}\ s}\ (s\ \mathit{fresh})$$

$$\text{(W-SPEC)}\quad \frac{\begin{array}{c}\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \mathbf{poly}\ \tau \;\hookrightarrow\; e' : \tau_\sigma' \qquad \tau_\sigma' \sim^{U} \mathbf{poly}\ s \\ \Delta' \vdash_{\text{W-SR}} \tau \;\hookrightarrow\; \tau' \qquad \Delta'' \mid U\,\Delta, \Delta' \Vdash_{\mathrm{W}} v : \mathrm{IsMG}\ (U\,s)\ \tau'\end{array}}{\Delta'', U\,\Delta, \Delta' \mid U S\,\Gamma \vdash_{\mathrm{W}} \mathbf{spec}\ e : \tau \;\hookrightarrow\; v[e'] : \tau'}\ (s\ \mathit{fresh})$$

Figure 2.9: Type Specialisation Algorithm (first part).

$$\text{(W-DTUPLE)}\quad \frac{\begin{array}{c}\Delta_1 \mid S_1\,\Gamma \vdash_{\text{W}} e_1 : \tau_1 \;\hookrightarrow\; e'_1 : \tau'_1 \\ \ldots \qquad \Delta_n \mid S_n\,S_{n-1}\ldots S_1\,\Gamma \vdash_{\text{W}} e_n : \tau_n \;\hookrightarrow\; e'_n : \tau'_n\end{array}}{\begin{array}{l}S_n \ldots S_2\,\Delta_1, \ldots, \Delta_n \mid S_n \ldots S_1\,\Gamma \\ \quad \vdash_{\text{W}} (e_1, \ldots, e_n)^D : (\tau_1, \ldots, \tau_n)^D \\ \qquad \hookrightarrow\; (e'_1, \ldots, e'_n) : (S_n \ldots S_2\,\tau'_1, S_n \ldots S_3\,\tau'_2, \ldots, \tau'_n)\end{array}}$$

$$\text{(W-DPRJ)}\quad \frac{\Delta \mid S\,\Gamma \vdash_{\text{W}} e : (\tau_1, \ldots, \tau_n)^D \;\hookrightarrow\; e' : \tau' \qquad \tau' \sim^U (t_1, \ldots, t_n)}{U\,\Delta \mid U S\,\Gamma \vdash_{\text{W}} \pi^D_{i,n}\,e : \tau_i \;\hookrightarrow\; \pi_{i,n}\,e' : U\,t_i}\ (t_1,\ldots,t_n \textit{ fresh})$$

$$\text{(W-DLET)}\quad \frac{\begin{array}{c}\Delta_2 \mid S_2\,\Gamma \vdash_{\text{W}} e_2 : \tau_2 \;\hookrightarrow\; e'_2 : \tau'_2 \\ \Delta_1 \mid S_1\,(S_2\,\Gamma, x : \tau_2 \hookrightarrow x' : \tau'_2) \vdash_{\text{W}} e_1 : \tau_1 \;\hookrightarrow\; e'_1 : \tau'_1\end{array}}{\begin{array}{l}S_1\,\Delta_2, \Delta_1 \mid S_1 S_2\,\Gamma \vdash_{\text{W}} \mathbf{let}^D\ x = e_2\ \mathbf{in}\ e_1 : \tau_1 \\ \qquad\qquad\qquad \hookrightarrow\; \mathbf{let}\ x' = e'_2\ \mathbf{in}\ e'_1 : \tau'_1\end{array}}\ (x' \textit{ fresh})$$

Figure 2.10: Type Specialisation Algorithm (second part).

Source-Residual Relationship: The relationship between source and residual types is calculated by providing the algorithm with the source type as its input, so it produces as output the residual type and a predicate assignment expressing the restrictions on type variables. It can be interpreted as an attribute grammar with judgments of the form $\Delta \vdash_{\text{W-SR}} \tau \;\hookrightarrow\; \tau'$, where τ is an inherited attribute (i.e. input to the algorithm),and Δ and τ' are synthesized ones (i.e. output). The rules are given in Figure 2.12.

Finally, we present the most important property which is the reason for these systems — principality:

Theorem 2.28. *If we have $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \;\hookrightarrow\; e' : \sigma$, then there exist e'_p and σ_p satisfying $\Gamma \vdash_{\text{P}} e : \tau \;\hookrightarrow\; e'_p : \sigma_p$ such that for all Δ'', e'', σ'' with $\Delta'' \mid \Gamma \vdash_{\text{P}} e : \tau \;\hookrightarrow\; e'' : \sigma''$ there exists a conversion C and a substitution R satisfying $C : (\ \mid R\,\sigma_p) \geq (\Delta'' \mid \sigma'')$ and $C[e'_p] = e''$.*

The meaning of this theorem is that every residual term and type obtained by the system $\vdash_{\text{P}}$ can be expressed as a particular case of the residual term and type produced by the algorithm. We can found technical details of this proof in [Martínez López and Hughes, 2004].

Principality allows us to specialize programs in a modular way, specializing each piece of code independently from the context where it will be used.

2.3.5 Extensions

The language considered in Section 2.3.1 is a small subset of a real programming language. In order to consider examples of some interest, such as the interpreter for lambda-calculus, Martínez López extends the language with new constructs [Martínez López, 2004], considering how to obtain principal specialization for them. Some simple extensions that we can mention are:

$$c \sim^{\mathrm{Id}} c$$

$$\hat{n} \sim^{\mathrm{Id}} \hat{n}$$

$$Int \sim^{\mathrm{Id}} Int$$

$$\alpha \sim^{\mathrm{Id}} \alpha$$

$$\frac{\alpha \notin FV(\sigma)}{\alpha \sim^{[\alpha/\sigma]} \sigma}$$

$$\frac{\alpha \notin FV(\sigma)}{\sigma \sim^{[\alpha/\sigma]} \alpha}$$

$$\frac{\tau_1' \sim^{T} \tau_2' \qquad T\,\tau_1'' \sim^{U} T\,\tau_2''}{\tau_1' \to \tau_2' \sim^{UT} \tau_1'' \to \tau_2''}$$

$$\frac{\tau_{11}' \sim^{T_1} \tau_{21}' \qquad T_1\,\tau_{12}' \sim^{T_2} T_1\,\tau_{22}' \qquad \ldots \qquad T_{n-1}\ldots T_1\,\tau_{1n}' \sim^{T_n} T_{n-1}\ldots T_1\,\tau_{2n}'}{(\tau_{11}', \ldots, \tau_{1n}') \sim^{T_n \ldots T_1} (\tau_{21}', \ldots, \tau_{2n}')}$$

$$\frac{\sigma \sim^{U} \sigma'}{\mathbf{poly}\ \sigma \sim^{U} \mathbf{poly}\ \sigma'}$$

$$\frac{\sigma[\alpha/c] \sim^{U} \sigma'[\alpha'/c]}{\forall\alpha.\sigma \sim^{U} \forall\alpha'.\sigma'}\ (c\ \mathit{fresh})$$

$$\frac{\delta \sim^{U} \delta' \qquad \rho \sim^{U} \rho'}{\delta \Rightarrow \rho \sim^{U} \delta' \Rightarrow \rho'}$$

$$\frac{\tau \sim^{U} \tau'}{\mathrm{IsInt}\ \tau \sim^{U} \mathrm{IsInt}\ \tau'}$$

$$\frac{\tau \sim^{T} \tau' \qquad T\,\tau_1 \sim^{U} T\,\tau_1' \qquad UT\,\tau_2 \sim^{V} UT\,\tau_2'}{\tau := \tau_1 + \tau_2 \sim^{VUT} \tau' := \tau_1' + \tau_2'}$$

$$\frac{\sigma_1 \sim^{T} \sigma_2 \qquad T\,\sigma_1' \sim^{U} T\,\sigma_2'}{\mathrm{IsMG}\ \sigma_1\ \sigma_1' \sim^{UT} \mathrm{IsMG}\ \sigma_2\ \sigma_2'}$$

Figure 2.11: Rules for unification.

$$\text{IsInt } t \vdash_{\text{W-SR}} Int^S \hookrightarrow t \quad (t \textit{ fresh})$$

$$\emptyset \vdash_{\text{W-SR}} Int^D \hookrightarrow Int$$

$$\frac{\Delta_1 \vdash_{\text{W-SR}} \tau_1 \hookrightarrow \tau_1' \qquad \Delta_2 \vdash_{\text{W-SR}} \tau_2 \hookrightarrow \tau_2'}{\Delta_1, \Delta_2 \vdash_{\text{W-SR}} \tau_2 \rightarrow^D \tau_1 \hookrightarrow \tau_2' \rightarrow \tau_1'}$$

$$\frac{(\Delta_1 \vdash_{\text{W-SR}} \tau_1 \hookrightarrow \tau_1')_{i=1,\ldots,n}}{\Delta_1, \ldots, \Delta_n \vdash_{\text{W-SR}} (\tau_1, \ldots, \tau_n)^D \hookrightarrow (\tau_1', \ldots, \tau_n')}$$

$$\frac{\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'}{\text{IsMG } \sigma\ s \vdash_{\text{W-SR}} \textbf{poly } \tau \hookrightarrow \textbf{poly } s} \ (\sigma{=}\text{Gen}_{\emptyset}(\Delta{\Rightarrow}\tau') \textit{ and } s \textit{ fresh})$$

Figure 2.12: Rules calculating principal source-residual relationship.

- Booleans as a base type (together with its primitives).
- Static sum-like types, with contructors and pattern-matching as part of a **case**.

There are extensions that are not straightforward to obtain, for example, we have:

- Recursion (dynamic and static one)
- Static functions
- Polymorphism

We have summarized the principal ideas, definitions and theorems behind *principal type specialization*. We proceed in the next chapter with our work, that is, extending *principal type specialization* to handle dynamic sum-types.

Chapter 3

Principal Specialization of Dynamic Sum-Types

" Mas no transformarás una sola cosa, un guijarro, un grano de arena hasta que no sepas cuál será el bien y el mal que resultará. El mundo se mantiene en Equilibrio. El poder de Transformación y de Invocación de un mago puede romper ese equilibrio. Tiene que ser guiado por el conocimiento, y servir a la necesidad."

La escuela de hechicería
Un mago de Terramar
Úrsula K. Le Guin

In this chapter we present our proposal to deal with dynamic sum-type in the framework developed in [Martínez López and Hughes, 2004].

First, in Sections 3.1 and 3.2, we extend the grammars describing source and residual terms as well as source and residual types. Then, in Section 3.3, we give two rules to type the extensions made in the residual code. Finally, in Section 3.4, we present the rules to specicialize terms related to dynamic sum-types. We also extend all the proofs of lemmas, propositions and theorems that appear in Sections 3.3 and 3.4 to prove that they hold after our additions.

3.1 Extending Source and Residual Language

In order to introduce dynamic sum-types we first have to extend the source language described in [Martínez López and Hughes, 2004]. We use the same notation and conventions for dynamic sum-types used in [Hughes, 1996], where constructors are distinguished lexically and take only one argument. However, instead of anonymous sum-types, we will use named ones. Moreover, our dynamic sum-types have arity zero, i.e. with no parameters. We extend the source language as follows.

Definition 3.1. Let D denote a sum-type name and C a constructor name. A *source term*, denoted by e, is an element of the language defined by the following abstract grammar:

$$
\begin{array}{lcl}
e & ::= & [\ ddcl\ ]^{*}\ e_p \\
ddcl & ::= & \mathbf{data}\ D^{D}\ =\ cs \\
cs & ::= & C_1^{D}\ \tau\ ||\ \ldots\ ||\ C_n^{D}\ \tau \\
e_p & ::= & \ldots \\
 & & \mid C^{D} \\
 & & \mid \mathbf{case}^{D}\, e_p\ \mathbf{of}\ [\ br\ ]^{*} \\
br & ::= & C^{D}\ x\ \rightarrow\ e_p
\end{array}
$$

where e_p is the grammar describing source terms in [Martínez López and Hughes, 2004] but with two extra constructs. A sequence of none, one or more e items is denoted by $[\, e\,]^*$. The non-terminal symbol cs denotes just an enumeration of contructors (the symbol $||$ is used to avoid confusion with the symbol $|$ used for choice).

Declarations of dynamic sum-types are only allowed at the begining of the program. The reason for this is only simplicity and it is not hard to construct a new grammar that enable us to declare dynamic sum-types in any other part of the program.

Because we are dealing with dynamic constructs, we have to add them to the residual language too.

Definition 3.2. Let D denote a sum-type name and C a constructor name. A *residual term*, denoted by e', is an element of the language defined by the following grammar:

$$\begin{array}{lll}
e' & ::= & [\, ddcl' \,]^* \; e'_p \\
ddcl' & ::= & \mathbf{data}\; D \;=\; cs' \\
cs' & ::= & C^1_{v'_p}\; \tau \;||\; \ldots \;||\; C^n_{v'_p}\; \tau \\
e'_p & ::= & \ldots \\
 & & |\; C \\
 & & |\; \mathbf{case}\; e'_p\; \mathbf{of}\; [\, br' \,]^* \\
 & & |\; \mathbf{protocase}_v\; e'_p\; \mathbf{with}\; v'_p\; \mathbf{of}\; [\, br' \,]^* \\
br' & ::= & C\, x \;\rightarrow\; e'_p \\
v'_p & ::= & \ldots \;|\; \{C^k\}_{k \in I}
\end{array}$$

where e'_p is the grammar describing residual terms in [Martínez López and Hughes, 2004] but with three extra constructs. On the other hand, v'_p is the grammar defined in [Martínez López and Hughes, 2004] for evidence but with a new kind of evidence: a set of constructors names. The non-terminal symbol cs' denotes just an enumeration of contructors. The purpose of the construct $\mathbf{protocase}_v$ will be explain later.

The function of upper and lower indexes of residual constructors needs further explanation. If there is a dynamic constructor in our source program, said C^D_k, the specialization process put it in the residual program as C^k_v, without the dynamic tag and with the upper index being the same as the lower index in the source program. On the other hand, the lower index in the residual program, denoted by evidence v, indicates which of the several specializations of the sum-type we are refering — see next example.

Example 3.3. The goal of the lower index in a residual dynamic constructor is important because it determines which specialization of D^D is considered. For example, if we have the following sum-type declaration

$$data\; D^D = C^D_1\;\; Int^S$$

in the source code

$$\begin{array}{l}
\mathbf{let}^D\;\; d_1 = C^D_1\;\; 29^S \\
\;\mathbf{in}\;\mathbf{let}^D\;\; d_2 = C^D_1\;\; 71^S \\
\qquad \mathbf{in}\; 4^D
\end{array}$$

The residuals of C_1^D are applied to two arguments with different residual types. Then it is necessary to consider two different specialization of D^D, where C^1 can be applied to arguments with types $\hat{29}$ and $\hat{71}$ respectively. After specialization and constraint solving, the residual type is *Int*, and thus there are no possibilities for new restrictions affecting the residual code produced. So, constraint solving can completely and safely solve all the predicates [Badenes and Martínez López, 2002], producing the residual program

$$\begin{array}{l} \mathit{data}\ D_1 = C_1^1\ \hat{29} \\ \mathit{data}\ D_2 = C_2^1\ \hat{71} \\ \\ \textbf{let}\ d_1 = C_1^1\ \bullet \\ \ \textbf{in let}\ d_2 = C_2^1\ \bullet \\ \quad\ \ \textbf{in}\ 4 \end{array}$$

Observe how C_2^1 belongs to D_2.

3.2 Extending Source And Residual Types

It is also necessary to extend the source and residual types. We do this as follows.

DEFINITION 3.4. A *source type*, denoted by τ, is an element of the language defined by the following grammar:

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

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

DEFINITION 3.5. Let t denote a *type variable* from a countable infinite set of variables and s a *type scheme variable* from another countable 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

$$\begin{array}{lcl} \tau' & ::= & t \mid \mathit{Int} \mid \hat{n} \mid \tau' \to \tau' \mid (\tau', \ldots, \tau') \mid \textbf{poly}\ \sigma \mid D_n \\ \rho & ::= & \delta \Rightarrow \rho \mid \tau' \\ \sigma & ::= & s \mid \forall s.\sigma \mid \forall t.\sigma \mid \rho \\ \delta & ::= & \mathrm{IsInt}\ \tau' \mid \tau' := \tau' + \tau' \mid \mathrm{IsMG}\ \sigma\ \sigma \mid \delta_d \end{array}$$

We are free of choosing any form of a residual sum-type name (except those that already exist, like *Int*, etc.) because residual programs are generated automatically, not by hand. We choose that a residual sum-type name is composed of two parts: a string, denoted by D, and a number, denoted by n. A residual constructor will have its lower index identical to the lower index of its residual sum-type name.

New predicates are introduced in order to establish relations between residual types, helping us to determine which is the residual type of the argument of each residual constructor.

$$\text{(RT-DCONSTR)}\ \frac{\begin{array}{l}\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : \tau'_j \\ (\Delta \Vdash v_k : C^k \in \tau'_e?\Delta_k, v_{m_k} : C^k \in \tau'_e?\text{HasC } \tau'_e \;\; C^k \;\; \tau'_k)_{k \in D \wedge k \neq j} \\ \Delta \Vdash v_{m_j} : \text{HasC } \tau'_e \;\; C^j \;\; \tau'_j, v_{cs} : \text{ConstrsOf } \tau'_e\end{array}}{\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} C^j_{v_{m_j}} \; e' : \tau'_e}$$

$$\text{(RT-DCASE)}\ \frac{\begin{array}{l}\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : \tau'_e \\ (h_k : \Delta_k \mid \Gamma_{\text{R}} \vdash_{\text{RT}} \lambda x'_k.e'_k : \tau'_k \rightarrow \tau')_{k \in B} \\ (\Delta \Vdash v_{m_k} : C^k \in \tau'_e?\text{HasC } \tau'_e \;\; C^k \;\; \tau'_k, v_k : C^k \in \tau'_e?\Delta_k)_{k \in B} \\ \Delta \Vdash v_{cs} : \text{ConstrsOf } \tau'_e\end{array}}{\begin{array}{l}\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} \textbf{protocase}_v \; e' \; \textbf{with} \; v_{cs} \; \textbf{of} \\ \qquad\qquad (C^k_{v_{m_k}} \; x'_k \;\rightarrow\; e'_k[v_k/h_k])_{k \in B} \\ \qquad : \tau'\end{array}}$$

Figure 3.1: Residual type rules of case and constructor application

Definition 3.6. The new predicates, denoted in Definition 3.5 as δ_d, are defined as follows

$$\begin{array}{rl}\delta_d ::= & \text{ConstrsOf } \tau' \\ & \mid \text{HasC } \tau' \;\; C^k \;\; \tau' \\ & \mid C \in \tau'?\delta\end{array}$$

Residual sum-types will be determined during constraint solving, so we need to put information into predicates in order to use it when this phase is performed. The purpose of each predicate is explained later in Section 3.4.

3.3 Extending RT Relation

The typing of residual terms is defined by a separate system, called RT, as we explained in Section 2.3.2.

There is no need of type inference for residual terms because the specialization process is well behaved with respect to system RT. In Figure 3.1 we define the rules for typing expressions that involve sum-types constructs.

We extend the proof of the following properties. They show that contexts can be weakened in residual judgments, and that conversions indeed relate types σ and σ' in their contexts.

Proposition 3.7. *If* $h : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : \sigma$, *and* $\Delta' \Vdash v : \Delta$, *then* $\Delta' \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e'[v/h] : \sigma$.

Theorem 3.8. *If* $h : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : \sigma$, *and* $C : (h : \Delta \mid \sigma) \geq (h' : \Delta' \mid \sigma')$, *then* $h' : \Delta' \mid \Gamma_{\text{R}} \vdash_{\text{RT}} C[e'] : \sigma'$.

3.4 Specializations Rules for Dynamic Sum-Types

The system specifying type specialization is composed by two system of rules, called SR and P. In order to specialize sum-types we have to extend both systems and the entailment relationship ($\Vdash$).

$$\text{(HASC)}\ \frac{\Delta \Vdash D_n(C^k\ ) \sim \tau'}{\Delta \Vdash n : \text{HasC}\ D_n\ \ C^k\ \ \tau'}$$

$$\text{(CONSTRS-OF)}\ \frac{D_n = \{C^k\ \}_{k\in I}}{\Delta \Vdash \{C^k\ \}_{k\in I} : \text{ConstrsOf}\ D_n}$$

$$\text{(GUARD-TRUE)}\ \frac{C^k \in D_n \quad \Delta \Vdash v : \Delta'}{\Delta \Vdash v : C^k \in D_n?\Delta'}$$

$$\text{(GUARD-FALSE)}\ \frac{C^k \notin D_n}{\Delta \Vdash \bullet : C^k \in D_n?\Delta'}$$

$$\text{(UNIFY-HASC)}\ \frac{\Delta, h : \text{HasC}\ \tau'\ \ C^k\ \ \tau_1'', \Delta' \Vdash \tau_1'' \sim \tau_2''}{\Delta, h : \text{HasC}\ \tau'\ \ C^k\ \ \tau_1'', \Delta' \Vdash h : \text{HasC}\ \tau'\ \ C^k\ \ \tau_2''}$$

$$\text{(HASC-GUARD)}\ \frac{\Delta, \text{HasC}\ \tau'\ \ C^k\ \ \tau'', \Delta' \Vdash v : \Delta''}{\Delta, \text{HasC}\ \tau'\ \ C^k\ \ \tau'', \Delta' \Vdash v : C^k \in \tau'?\Delta''}$$

$$\text{(ENTL-GUARD)}\ \frac{h : \Delta \Vdash v' : \Delta' \qquad \Delta'' \Vdash v_{cs} : \text{ConstrsOf}\ \tau'}{\Delta'', h : C^k \in \tau'?\Delta \Vdash \mathbf{if}_v\ C^k \in v_{cs}\ \mathbf{then}\ v'\ \mathbf{else}\ \bullet : C^k \in \tau'?\Delta'}$$

Figure 3.2: Entailment for evidence construction

3.4.1 Entailment Relationship

Evidence is used in residual code for capturing differences between all possible specializations of a given source program.

We add several entailment rules to specify how to construct evidence for predicates involving sum-types — see Figure 3.2. The evidence of (HASC) is just the number of the specialized instance obtained from D^D. The rule (CONSTRS-OF) has as its evidence the names of constructors that are in a residual sum-type definition. On the other hand, rules (GUARD-TRUE) and (GUARD-FALSE) say that if a constructor belongs to a residual sum-type definition, each of the predicates under the guard has to be proved accordingly, but if this is not the case, they can be proved trivially using evidence $\bullet$.

Additionally, (UNIFY-HASC) is expressing that if we have two predicates HasC which are relating a constructor of the same sum-type with two different residual types, these types have to unify, being consistent with the idea explained in Example 3.3. Observe that a predicate HasC is preserved in the hypothesis of that rule, because it will help constraint solving to construct residual sum-types definitions. The rule (HASC-GUARD) is used for constructing the evidence of a conditional predicate when we are sure that a constructor belongs to a sum-type. Finally, (ENTL-GUARD) shows how to construct evidence of a guard that we do not know in advance if it is either true or false.

We will need the following lemma for our formal proofs in the next chapter.

LEMMA 3.9. *If* $h_1 : \Delta_1 \Vdash v_2 : \Delta_2$ *and* $h : \Delta \Vdash v_1 : C^k \in \tau'?\Delta_1, v_{cs} : \text{ConstrsOf } \tau'$ *then* $h : \Delta \Vdash \textbf{if}_v\ C^k \in v_{cs}\ \textbf{then}\ v_2[v_1/h_1]\ \textbf{else}\ \bullet : C^k \in \tau'?\Delta_2$

3.4.2 Source-Residual Relationship

The source-residual relationship, expressed by the judgment $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau'$, ensures a relation between a source type and a residual one in order to achive principality together with system P. We have previously introduced new source types on Definition 3.4, so through this relation we have to express which residual types are related to them — see Figure 3.3.

The symbol D usually represents a dynamic sum-type but sometimes, depending on the context, it also represents a set of indices of the constructors of the corresponding dynamic sum-type used — remember that constructors are enumerated sequentially starting from one. Additionally, $D(C^k)$ is the source type of C_k^{D}'s argument in the definition of the sum-type D^{D}.

The rule (SR-DDT), presented in Figure 3.3, establishes the conditions that have to hold in order to D^{D} be related to a residual sum-type τ'. It establishes that for each residual constructor C^k belonging to the definition of τ', its argument has the residual type τ'_k — captured by $C^k \in \tau'?\text{HasC } \tau'\ \ C^k\ \ \tau'_k$. Observe that we also need predicates restricting τ'_k. We are capturing the posibility of a constructor not being needed, and the residual sum-type will not include it — the corresponding guarded predicates will be trivially proved with evidence $\bullet$ in constraint solving.

EXAMPLE 3.10. In this example we show predicates that are generated when applying rule (SR-DDT). We have the sum-type declaration

$$\mathit{data}\ D^{D} = \mathit{Left}^{D}\ \ \mathit{Int}^{S}\ \mid \mathit{Right}^{D}\ \ \mathit{Bool}^{S}$$

and the judgement

$$\begin{aligned}
&\vdash_{\text{P}} \lambda^{D} e.e : D^{D} \to^{D} D^{D} \\
&\quad\hookrightarrow \Lambda h_1, h_2, h_3, h_4, h_5.\lambda e'.e' : \ \forall t_1, t_2, t_3.h_1 : \mathit{Right} \in t_1?\text{HasC } t_1\ \mathit{Right}\ t_2, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_2 : \mathit{Right} \in t_1?\text{IsBool } t_2, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_3 : \mathit{Left} \in t_1?\text{HasC } t_1\ \mathit{Left}\ t_3, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_4 : \mathit{Left} \in t_1?\text{IsInt } t_3, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_5 : \text{ConstrsOf } t_1 \Rightarrow t_1 \to t_1
\end{aligned}$$

If we apply this term to $(\mathit{Left}^{D}\ \ 10^{S})$, $\hat{10}$ will be the residual type of *Left*'s argument. This last fact can be appreciated in the following judgement

$$\begin{aligned}
&\vdash_{\text{P}} (\lambda^{D} e.e) \mathbin{@^{D}} (\mathit{Left}^{D}\ \ 10^{S}) : D^{D} \to^{D} D^{D} \\
&\quad\hookrightarrow \Lambda h_1, h_2, h_3, h_4, h_5, h_6.(\lambda e'.e')@(\mathit{Left}_{h_6}\ \bullet) \\
&\quad :\ \forall t_1, t_2, t_3.h_1 : \mathit{Right} \in t_1?\text{HasC } t_1\ \mathit{Right}\ t_2, \\
&\qquad\qquad\qquad\quad h_2 : \mathit{Right} \in t_1?\text{IsBool } t_2, \\
&\qquad\qquad\qquad\quad h_3 : \mathit{Right} \in t_1?\text{HasC } t_1\ \mathit{Right}\ t_3, \\
&\qquad\qquad\qquad\quad h_4 : \mathit{Right} \in t_1?\text{IsBool } t_3, \\
&\qquad\qquad\qquad\quad h_5 : \text{ConstrsOf } t_1, \\
&\qquad\qquad\qquad\quad h_6 : \text{HasC } t_1\ \mathit{Left}\ \hat{10} \Rightarrow t_1
\end{aligned}$$

$$\text{(SR-DDT)}\ \frac{\begin{array}{l}(\Delta_k \vdash_{\text{SR}} D(C^k) \ \hookrightarrow\ \tau'_k)_{k\in D} \\ (\Delta \Vdash C^k \in \tau' ? \Delta_k, C^k \in \tau' ? \text{HasC}\ \tau' \ \ C^k \ \ \tau'_k)_{k\in D} \\ \Delta \Vdash \text{ConstrsOf}\ \tau'\end{array}}{\Delta \vdash_{\text{SR}} D^D \ \hookrightarrow\ \tau'}$$

Figure 3.3: Rule defining the source-residual relationship for dynamic sum-types

where a predicate *HasC* restricting *Left* has been added to the context. Entailment rules are responsible for eliminating guarded predicates. Note how evidence h_6 indicates, in $Left_{h_6}$, which one of the several possible specializations of the given source sum-type we are refering to.

The following properties of the system SR are useful. We show that they are preserved after our addition.

Proposition 3.11. *If* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$ *then* $S\,\Delta \vdash_{\text{SR}} \tau \ \hookrightarrow\ S\,\sigma$.

Proposition 3.12. *If* $\Delta \vdash_{\text{SR}} \tau \ \hookrightarrow\ \sigma$ *and* $\Delta' \Vdash \Delta$*, then* $\Delta' \vdash_{\text{SR}} \tau \ \hookrightarrow\ \sigma$.

Theorem 3.13. *If* $\Delta \vdash_{\text{SR}} \tau \ \hookrightarrow\ \sigma$ *and* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *then* $\Delta' \vdash_{\text{SR}} \tau \ \hookrightarrow\ \sigma'$.

This last theorem shows that if a residual type can be obtained from a source one, any instance of it can be obtained too.

3.4.3 Rules for Constructors and Case

The rules to specify specialization of constructors and **case**'s appear in Figure 3.4. Observe how (dconstr) specializes the constructor's argument e to e' and introduces residual types τ'_k for the other arguments that belong to the same sum-type. If we have a constructor's argument with different residual types, they have to be unified as stated the rule (unify-hasc) — see Examples 3.22 and 3.23.

Observe that the constructor appearing in rule (dconstr) is always applied to an argument. This is enough because non-applied constructors are considered η-expanded, obtaining functions whose bodies always have applied constructors. After all specialization and post-processing phases have been carried out, η-reductions can be performed to get a more elegant residual code. This last remark is important in order to understand how some source codes are specialized — see Examples 3.24 and 3.25.

The dual rule of (dconstr) is (dcase), the most complex one. The source term e is specialized to e' with residual type τ'_e, which could have information about the residual types of the constructors' arguments — see Examples 3.26 and 3.27. In addition to the specialization of e, every branch is specialized assuming that each constructor has an argument whose residual type is τ'_k. However, we do not know a priori if a constructor is ever applied to an argument; but if there is someone, it has to have the residual type τ'_k. We obtain this effect by means of guards, (unify-hasc) and (hasc-guard) — see Example 3.28.

It could happen that all the evidence that correspond to a branch are just $\bullet$ — see entailment rule (guard-false) — leaving a meaningless residual branch. Nevertheless,

$$\text{(DCONSTR)}\quad \frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\text{P}} e : D(C^j) \hookrightarrow e' : \tau'_j \\ (\Delta_k \vdash_{\text{SR}} D(C^k) \hookrightarrow \tau'_k)_{k \in D \wedge k \neq j} \\ (\Delta \Vdash v_k : C^k \in \tau'_e ? \Delta_k, v_{m_k} : C^k \in \tau'_e ? \text{HasC } \tau'_e \;\; C^k \;\; \tau'_k)_{k \in D \wedge k \neq j} \\ \Delta \Vdash v_{m_j} : \text{HasC } \tau'_e \;\; C^j \;\; \tau'_j, v_{cs} : \text{ConstrsOf } \tau'_e\end{array}}{\Delta \mid \Gamma \vdash_{\text{P}} C^D_j \; e : D^D \hookrightarrow C^j_{v_{m_j}} \; e' : \tau'_e}$$

$$\text{(DCASE)}\quad \frac{\begin{array}{l}\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau' \\ \Delta \mid \Gamma \vdash_{\text{P}} e : D^D \hookrightarrow e' : \tau'_e \\ (h_k : \Delta_k \mid \Gamma \vdash_{\text{P}} \lambda^D x_k . e_k : D(C^k) \rightarrow^D \tau \hookrightarrow \lambda x'_k . e'_k : \tau'_k \rightarrow \tau')_{k \in B} \\ (\Delta \Vdash v_{m_k} : C^k \in \tau'_e ? \text{HasC } \tau'_e \;\; C^k \;\; \tau'_k, v_k : C^k \in \tau'_e ? \Delta_k)_{k \in B} \\ \Delta \Vdash v_{cs} : \text{ConstrsOf } \tau'_e\end{array}}{\begin{array}{ll}\Delta \mid \Gamma \vdash_{\text{P}} & \textbf{case } e \textbf{ of} \\ & \quad (C^D_k \;\; x_k \rightarrow e_k)_{k \in B} \\ & : \tau \\ & \hookrightarrow \\ & \textbf{protocase}_v \; e' \textbf{ with } v_{cs} \textbf{ of} \\ & \quad (C^k_{v_{m_k}} \;\; x'_k \rightarrow e'_k[v_k/h_k])_{k \in B} \\ & : \tau'\end{array}}$$

Figure 3.4: Specialization rules for case and constructor application

$$\begin{array}{lll}\textbf{protocase}_v \; e' \textbf{ with } \{C^j\}_{j \in J \wedge J \neq \emptyset} \textbf{ of} & \triangleright & \textbf{case } e' \textbf{ of} \\ \quad (C^k_{v_{m_k}} \;\; x'_k \rightarrow e'_k)_{k \in B} & & \quad (C^j_{v_{m_j}} \;\; x'_j \rightarrow e'_j)_{j \in B \cap J} \\ \textbf{protocase}_v \; e' \textbf{ with } \{\} \textbf{ of} & \triangleright & \textbf{error} \text{ “There are no branches.”} \\ \quad (C^k_{v_{m_k}} \;\; x'_k \rightarrow e'_k)_{k \in B} & & \end{array}$$

Figure 3.5: Reduction rule for a $\textbf{protocase}_v$

the $\textbf{protocase}_v$ has all the necessary information to be transformed into a **case** where every branch is meaningful. In Figure 3.5 we have the reduction rules that obtain this effect; observe how the only branches preseved in the transformation are those whose constructors belong to the definiton of a given residual sum-type — see Example 3.28. We use the construct **error**, defined in [Martínez López, 2004], to produce controled errors in residual code. On the other hand, we also need to extend the definition of equality between residual terms in order to establish that two $\textbf{protocase}_v$ are equal if and only if they can produce the same residual **case**s — see Figures 3.6 and 3.7.

Now, we need to extend the proof of the proposition that system P is well behaved with respect to systems SR and RT.

THEOREM 3.14. *If* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$*, and for all* $x : \tau_x \hookrightarrow x' : \tau'_x \in \Gamma$*,* $\Delta \vdash_{\text{SR}} \tau_x \hookrightarrow \tau'_x$*, then* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$*.*

Given a specialisation assignment, $\Gamma = [x_i : \tau_i \hookrightarrow x'_i : \sigma_i \mid i = 1..n]$, we define the projection of Γ to the residual language to be $\Gamma_{(\text{RT})} = [x'_i : \sigma_i \mid i = 1..n]$.

THEOREM 3.15. *If* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$*, then* $\Delta \mid \Gamma_{(\text{RT})} \vdash_{\text{RT}} e' : \sigma$*.*

$$
\begin{array}{rcl}
\textbf{protocase}_v\ e' \ \textbf{with}\ \{C^k\ \}_{k\in I}\ \textbf{of} & = & \textbf{protocase}_v\ e'' \ \textbf{with}\ \{C^k\ \}_{k\in I}\ \textbf{of} \\
(C^k_{v_{m_k}}\ x'_k\ \rightarrow\ e'_k)_{k\in B} & & (C^k_{v'_{m_k}}\ x''_k\ \rightarrow\ e''_k)_{k\in B} \\
& \text{iff} & \\
e' & = & e'' \\
(v_{m_k} & = & v'_{m_k})_{k\in B\cap I} \\
(\lambda x'_k.e'_k & = & \lambda x''_k.e''_k)_{k\in B\cap I}
\end{array}
$$

Figure 3.6: Equality Rule I for $\textbf{protocase}_v$

$$
\begin{array}{rcl}
\textbf{protocase}_v\ e' \ \textbf{with}\ h\ \textbf{of} & = & \textbf{protocase}_v\ e'' \ \textbf{with}\ h\ \textbf{of} \\
(C^k_{v_{m_k}}\ x'_k\ \rightarrow\ e'_k)_{k\in B} & & (C^k_{v'_{m_k}}\ x''_k\ \rightarrow\ e''_k)_{k\in B} \\
\multicolumn{3}{c}{\text{iff for all } D_n \text{ with the form } \{C^k\ \}_{k\in I}} \\
e'[D_n/h] & = & e''[D_n/h] \\
(v_{m_k}[D_n/h] & = & v'_{m_k}[D_n/h])_{k\in B\cap I} \\
(\lambda x'_k.e'_k[D_n/h] & = & \lambda x''_k.e''_k[D_n/h])_{k\in B\cap I}
\end{array}
$$

Figure 3.7: Equality Rule II for $\textbf{protocase}_v$

Additionally, we also extend the following properties

PROPOSITION 3.16. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \tau'$ *and* $h' : \Delta' \Vdash v : \Delta$, *then* $h' : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'[h/v] : \tau'$

PROPOSITION 3.17. *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 3.18. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ *then* $EV(e') \subseteq h$

LEMMA 3.19. *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ *then there exist* β_j, Δ_σ, *and* τ' *such that* $\sigma = \forall\beta_j.\Delta_\sigma \Rightarrow \tau'$.

3.5 Examples

In this section we show examples of specializations using the rules described, providing also useful observations which will help to better understand ideas behind specialization of sum-types. In this section we start with basic examples. Then, in next chapter, we will show more interesting examples, with interactions between sum-types and polyvariant expressions, static functions, static recursion, etc. We will specialize ground terms.

EXAMPLE 3.20. Observe in this example how it is possible to produce two instances of a given dynamic sum-type. We have two predicates HasC, one of them on t and $\hat{46}$, and the other on t' and $\hat{99}$.

We have the sum-type declaration

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

in the source code

$$\begin{array}{l}\mathbf{let}^D\ d_1 = \mathbf{if}^D\, True^D\ \mathbf{then}\ Left^D\ 46^S\ \mathbf{else}\ Right^D\ 40^D\\ \ \mathbf{in}\ \mathbf{let}^D\ d_2 = \mathbf{if}^D\, False^D\ \mathbf{then}\ Right^D\ 20^D\ \mathbf{else}\ Left^D\ 99^S\\ \quad\quad \mathbf{in}\ 4^D\\ :Int^D\end{array}$$

and one of its specializations is

$$\begin{array}{l}\Lambda h_1,h_2,h_3,h_4,h_5,h_6.\mathbf{let}\ d_1 = \mathbf{if}\ True\ \mathbf{then}\ Left_{h_1}\ \bullet\ \mathbf{else}\ Right_{h_2}\ 40\\ \quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ \mathbf{let}\ d_2 = \mathbf{if}\ False\ \mathbf{then}\ Right_{h_4}\ 20\ \mathbf{else}\ Left_{h_5}\ \bullet\\ \quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ 4\end{array}$$

with residual type

$$\begin{array}{rl}\forall t,t'.h_1 & : \text{HasC}\ t\ Left\ \hat{46},\\ h_2 & : \text{HasC}\ t\ Right\ Int,\\ h_3 & : \text{ConstrsOf}\ t,\\ h_4 & : \text{HasC}\ t'\ Right\ Int,\\ h_5 & : \text{HasC}\ t'\ Left\ \hat{99},\\ h_6 & : \text{ConstrsOf}\ t' \Rightarrow Int\end{array}$$

Constraint solving will detect that between t and t' there is no interaction, so two different residual sum-types can be defined. Additionally, we are able to detect where each residual sum-type is used by looking at the lower index of each constructor. The constraint solving phase will produce the following residual code

$$\begin{array}{l}\mathbf{data}\ D_1 = Left_1\ \hat{46}\ |\ Right_1\ Int\\ \mathbf{data}\ D_2 = Left_2\ \hat{99}\ |\ Right_2\ Int\\ \\ \mathbf{let}\ d_1 = \mathbf{if}\ True\ \mathbf{then}\ Left_1\ \bullet\ \mathbf{else}\ Right_1\ 40\\ \ \mathbf{in}\ \mathbf{let}\ d_2 = \mathbf{if}\ False\ \mathbf{then}\ Right_2\ 20\ \mathbf{else}\ Left_2\ \bullet\\ \quad\quad \mathbf{in}\ 4\end{array}$$

Example 3.21. Here, we force information to flow from a contructor whose argument has residual type $\hat{75}$ to another whose argument is the argument of the enclosing function, fixing in this way which argument has to be taken by the function.

We have the sum-type declaration

$$data\ D^D = Left^D\ Int^S\ |\ Right^D\ Int^D$$

in the source code

$$\begin{array}{l}\lambda^D b.\mathbf{let}^D\ d_1 = Left^D\ 75^S\\ \quad\quad \mathbf{in}\ \mathbf{let}^D\ d_2 = Left^D\ 11^S\\ \quad\quad\quad\quad \mathbf{in}\ \mathbf{let}^D\ d_3 = Left^D\ b\\ \quad\quad\quad\quad\quad\quad \mathbf{in}\ \mathbf{let}^D\ u = \mathbf{if}^D\, True^D\ \mathbf{then}\ d_3\ \mathbf{else}\ d_1\\ \quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ 4^D\\ :Int^S \rightarrow^D Int^D\end{array}$$

and one of its specializations is

$$
\begin{array}{l}
\Lambda h_1, h_2, h_3, h_4, h_5, h_6.\lambda b.\mathbf{let}\ d_1 = \mathit{Left}_{h_4}\ \bullet \\
\qquad\qquad\qquad\qquad\qquad \mathbf{in\ let}\ d_2 = \mathit{Left}_{h_1}\ \bullet \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in\ let}\ d_3 = \mathit{Left}_{h_4}\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in\ let}\ u = \mathbf{if}\ \mathit{True}\ \mathbf{then}\ d_3\ \mathbf{else}\ d_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ 4 \\
: \ \forall t, t'.h_1 : \mathrm{HasC}\ t\ \mathit{Left}\ \hat{11}, \\
\qquad h_2 : \mathit{Right} \in t?\mathrm{HasC}\ t\ \mathit{Right}\ \mathit{Int}, \\
\qquad h_3 : \mathrm{ConstrsOf}\ t, \\
\qquad h_4 : \mathrm{HasC}\ t'\ \mathit{Left}\ \hat{75}, \\
\qquad h_5 : \mathit{Right} \in t'?\mathrm{HasC}\ t'\ \mathit{Right}\ \mathit{Int}, \\
\qquad h_6 : \mathrm{ConstrsOf}\ t' \Rightarrow \hat{75} \rightarrow \mathit{Int}
\end{array}
$$

Note that the guarded predicates are needed because there is no information concerning *Right*. Constraint solving will eliminate them, producing

$$
\begin{array}{l}
\mathbf{data}\ D_1 = \mathit{Left}_1\ \hat{75} \\
\mathbf{data}\ D_2 = \mathit{Left}_2\ \hat{11} \\
\\
\lambda b.\mathbf{let}\ d_1 = \mathit{Left}_1\ \bullet \\
\quad \mathbf{in\ let}\ d_2 = \mathit{Left}_2\ \bullet \\
\qquad \mathbf{in\ let}\ d_3 = \mathit{Left}_1\ b \\
\qquad\quad \mathbf{in\ let}\ u = \mathbf{if}\ \mathit{True}\ \mathbf{then}\ d_3\ \mathbf{else}\ d_1 \\
\qquad\qquad \mathbf{in}\ 4 \\
: \hat{75} \rightarrow \mathit{Int}
\end{array}
$$

EXAMPLE 3.22. It is possible that some constructors do not appear in the source code, and in consequence there are no information about the residual type of their arguments.

We have the sum-type declaration

$$\mathit{data}\ D^D = \mathit{Left}^D\ \ \mathit{Int}^S\ \mid \mathit{Right}^D\ \ \mathit{Int}^S$$

and the judgement

$$
\begin{array}{l}
\vdash_{\mathrm{P}} \mathit{Left}^D\ \ 57^S\ : D^D \\
\quad \hookrightarrow \Lambda h_1, h_2, h_3, h_4.\mathit{Left}_{h_1}\ \bullet :\ \ \forall t, t'.h_1 : \mathrm{HasC}\ t\ \mathit{Left}\ \hat{57}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad h_2 : \mathit{Right} \in t?\mathrm{HasC}\ t\ \mathit{Right}\ t', \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad h_3 : \mathit{Right} \in t?\mathrm{IsInt}\ t', \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad h_4 : \mathrm{ConstrsOf}\ t \Rightarrow t
\end{array}
$$

If the construct *Right* is eventually applied to an argument, its type should unify to t' by application of the entailment rules (HASC-GUARD) and (UNIFY-HASC).

EXAMPLE 3.23. We present an example that involves dynamic constructors where specialization is possible, but where predicates cannot be satisfied during constraint solving because two different residual types are assigned to the same constructor's argument.

We have the sum-type declaration

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

and the judgement

$$\begin{array}{l}\vdash_{\mathrm{P}} \mathbf{if}^D\ True\ \mathbf{then}\ Only^D\ \ 7^S\ \ \mathbf{else}\ Only^D\ \ 89^S\ : D^D \\ \quad \hookrightarrow \Lambda h_1, h_2, h_3.\mathbf{if}\ \ True\ \mathbf{then}\ Only_{h_1}\ \bullet\ \mathbf{else}\ Only_{h_2}\ \bullet : \ \ \forall t.h_1 : \mathrm{HasC}\ t\ Only\ \hat{7}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad h_2 : \mathrm{HasC}\ t\ Only\ \hat{89}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad h_3 : \mathrm{ConstrsOf}\ t \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow t\end{array}$$

The entailment rule (UNIFY-HASC) says that we need to prove $\hat{7} \sim \hat{89}$ to satisfy successfully the predicates in context; but that is impossible, and so constraint solving will fail.

EXAMPLE 3.24. In this example we can see how any information that proceed from the context of the function (by means of t') must unify with $\hat{101}$ to successfully solve predicates.

We have the sum-type declaration

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

and the judgement

$$\begin{array}{l}\vdash_{\mathrm{P}} \lambda^D x.\mathbf{let}^D\ \ f = Only^D \\ \qquad\quad \mathbf{in}\ (f\ @^D\ 101^S, f\ @^D\ x)^D \\ \quad : Int^S \rightarrow^D (D^D, D^D)^D \\ \quad \hookrightarrow \\ \quad \Lambda h_1, h_2, h_3, h_4.\lambda x.\mathbf{let}\ f = Only_{h_2} \\ \qquad\qquad\qquad\qquad\quad \mathbf{in}\ (f@\bullet, f@\bullet) \\ \quad :\ \forall t, t'.h_1 : \mathrm{IsInt}\ t', \\ \qquad\qquad h_2 : \mathrm{HasC}\ t\ Only\ \hat{101}, \\ \qquad\qquad h_3 : \mathrm{HasC}\ t\ Only\ t', \\ \qquad\qquad h_4 : \mathrm{ConstrsOf}\ t \Rightarrow t' \rightarrow (t, t)\end{array}$$

EXAMPLE 3.25. All constructors are η-expanded before the specialization and η-reduced after all post-processing phases. We just analyze and show source programs without η-expansions, but it is important to be aware of this to understand how specialization of a source constructor is carried out.

We have the sum-type declaration

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

and the judgement

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

EXAMPLE 3.26. Specialization and entailment rules were designed to enable constraint solving to detect when a **case** branch never receives information related to it, erasing them to obtain less dead code. Here, we show an example where this happens.

We have the sum-type

$$data\ D^{D} = Left^{D}\ Int^{S}\ \mid Right^{D}\ Bool^{S}$$

in the source code

$$\begin{array}{l}\mathbf{let}^{D}\ c = \mathbf{case}^{D}\,(Left^{D}\ 33^{S}\,)\ \mathbf{of} \\ \qquad\qquad Left^{D}\ n\ \rightarrow\ \mathbf{lift}\ n \\ \qquad\qquad Right^{D}\ b\ \rightarrow\ 7^{D} \\ \ \mathbf{in}\ 4^{D} \\ :Int\end{array}$$

and one of its specializations is

$$\begin{array}{l}\Lambda h_1, h_2, h_3, h_4. \\ \mathbf{let}\ c = \mathbf{protocase}_{v}\ (Left_{h_1}\ \bullet)\ \mathbf{with}\ h_4\ \mathbf{of} \\ \qquad\quad Left_{h_1}\ n\ \rightarrow\ 33 \\ \qquad\quad Right_{h_2}\ n\ \rightarrow\ 7 \\ \ \mathbf{in}\ 4\end{array}$$

with residual type

$$\begin{array}{l}\forall t, t'.h_1 : \mathrm{HasC}\ t\ Left\ \hat{33}, \\ \qquad\ h_2 : Right \in t?\mathrm{HasC}\ t\ Right\ t', \\ \qquad\ h_3 : Right \in t?\mathrm{IsBool}\ t', \\ \qquad\ h_4 : \mathrm{ConstrsOf}\ t \Rightarrow Int\end{array}$$

Constraint solving will produce

$$\mathbf{data}\ D_1 = Left_1\ \hat{33}$$

$$\begin{array}{l}\mathbf{let}\ c = \mathbf{case}\ (Left_1\ \bullet)\ \mathbf{of} \\ \qquad\qquad Left_1\ n\ \rightarrow\ 33 \\ \ \mathbf{in}\ 4\end{array}$$

where the branch concerning *Right* was eliminated.

EXAMPLE 3.27. In this example we show a **case** construct with no information to spread through any of its branches. We can observe that all predicates HasC are guarded — they were generated by application of (SR-DDT) and (DCASE) rules.

We have the sum-type declaration

$$data\ D^{D} = Left^{D}\ Int^{S}\ \mid Right^{D}\ Bool^{S}$$

and the judgement

$$
\begin{array}{l}
\vdash_{\mathrm{P}} \lambda^{D} e.\mathbf{case}^{D}\, e \;\mathbf{of} \\
\qquad\quad \mathit{Left}^{D}\; n \;\rightarrow\; \mathbf{lift}\; n \\
\quad : D^{D} \rightarrow^{D} \mathit{Int}^{D} \\
\quad \hookrightarrow \\
\quad \Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7. \\
\quad \lambda e.\mathbf{protocase}_{v}\; e \;\mathbf{with}\; h_7 \;\mathbf{of} \\
\qquad\quad \mathit{Left}_{h_5}\; n \;\rightarrow\; h_6 \\
\quad :\; \forall t_1, t_2, t_3, t_4. h_1 : \mathit{Left} \in t_1?\mathrm{HasC}\; t_1\; \mathit{Left}\; t_2, \\
\qquad\qquad\qquad\quad h_2 : \mathit{Left} \in t_1?\mathrm{IsInt}\; t_2, \\
\qquad\qquad\qquad\quad h_3 : \mathit{Right} \in t_1?\mathrm{HasC}\; t_1\; \mathit{Right}\; t_3, \\
\qquad\qquad\qquad\quad h_4 : \mathit{Right} \in t_1?\mathrm{IsBool}\; t_3, \\
\qquad\qquad\qquad\quad h_5 : \mathit{Left} \in t_1?\mathrm{HasC}\; t_1\; \mathit{Left}\; t_4, \\
\qquad\qquad\qquad\quad h_6 : \mathit{Left} \in t_1?\mathrm{IsInt}\; t_4, \\
\qquad\qquad\qquad\quad h_7 : \mathrm{ConstrsOf}\; t_1 \Rightarrow t_1 \rightarrow \mathit{Int}
\end{array}
$$

The constructor *Left* can have an argument of residual type $\hat{2}$, for instance, by just applying the previous source term to $(\mathit{Left}^{D}\;\; 2^{S})$. In this case we can conclude by (HASC-GUARD) and (UNIFY-HASC) that t_2 and t_4 have to be unified to $\hat{2}$. The entailment rules (HASC-GUARD) and (UNIFY-HASC) have an important rôle during constraint solving — they are responsible for spreading information between constructors that have equal names and belong to the same residual sum-type.

It is also possible that we never receive information concerning *Left*'s argument, which is easily achived, for example, applying the previous source abstraction to the source term $(\mathit{Right}^{D}\;\; \mathit{True}^{S})$.

EXAMPLE 3.28. Branches are specialized assuming that constructor C^k has an argument of residual type τ'_k. First, we present an example where a dynamic sum-type is abstracted and then we show what happen when we apply it to two different source terms.

We have the sum-type declaration

$$\mathit{data}\; D^{D} = \mathit{Left}^{D}\;\; \mathit{Int}^{S} \mid \mathit{Right}^{D}\;\; \mathit{Bool}^{S}$$

in the source code

$$
\begin{array}{l}
\lambda^{D} e.\mathbf{case}^{D}\, e \;\mathbf{of} \\
\qquad \mathit{Left}^{D}\; n \;\rightarrow\; \mathbf{let}^{D}\;\; \mathit{id} = \lambda^{D} x.x \\
\qquad\qquad\qquad\qquad \mathbf{in}\;\mathbf{let}^{D}\;\; \mathit{force} = (\mathit{id}\;@^{D}\, n, \mathit{id}\;@^{D}\, 44^{S})^{D} \\
\qquad\qquad\qquad\qquad\quad\;\; \mathbf{in}\;\mathbf{lift}\; n \\
\qquad \mathit{Right}^{D}\; b \;\rightarrow\; \mathbf{if}^{S}\, b \;\mathbf{then}\; 10^{D}\;\; \mathbf{else}\; 70^{D} \\
: D^{D} \rightarrow^{D} \mathit{Int}^{D}
\end{array}
$$

and one of its specializations is

$$
\begin{array}{l}
\Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9. \\
\lambda e.\mathbf{protocase}_{v}\; e \;\mathbf{with}\; h_{12} \;\mathbf{of} \\
\qquad \mathit{Left}_{h_5}\; n \;\rightarrow\; \mathbf{let}\; \mathit{force} = (\mathit{id}@n, \mathit{id}@\bullet) \\
\qquad\qquad\qquad\qquad \mathbf{in}\; h_6 \\
\qquad \mathit{Right}_{h_7}\; b \;\rightarrow\; \mathbf{if}_{v}\; h_8 \;\mathbf{then}\; 10 \;\mathbf{else}\; 70
\end{array}
$$

with residual type

$$
\begin{array}{l}
\forall t_1, t_2, t_3, t_4. \\
\quad h_1 : \mathit{Left} \in t_1?\mathrm{HasC}\ t_1\ \mathit{Left}\ t_2, \\
\quad h_2 : \mathit{Left} \in t_1?\mathrm{IsInt}\ t_2, \\
\quad h_3 : \mathit{Right} \in t_1?\mathrm{HasC}\ t_1\ \mathit{Right}\ t_3, \\
\quad h_4 : \mathit{Right} \in t_1?\mathrm{IsBool}\ t_3, \\
\quad h_5 : \mathit{Left} \in t_1?\mathrm{HasC}\ t_1\ \mathit{Left}\ \hat{44}, \\
\quad h_6 : \mathit{Left} \in t_1?\mathrm{IsInt}\ \hat{44}, \\
\quad h_7 : \mathit{Right} \in t_1?\mathrm{HasC}\ t_1\ \mathit{Right}\ t_4, \\
\quad h_8 : \mathit{Right} \in t_1?\mathrm{IsBool}\ t_4, \\
\quad h_9 : \mathrm{ConstrsOf}\ t_1 \Rightarrow t_1 \rightarrow \mathit{Int}
\end{array}
$$

Observe that there are two guarded HasC for each constructor, one of them generated by (SR-DDT) and the other one by (DCASE).

If we apply the previous function to the code

$$
\begin{array}{l}
\vdash_{\mathrm{P}} \mathbf{if}^{D}\, \mathit{False}^{D}\ \mathbf{then}\ \mathit{Right}^{D}\ \mathit{True}^{S}\ \mathbf{else}\ \mathit{Left}^{D}\ 44^{S} : D^{D} \\
\quad \hookrightarrow \\
\quad \Lambda h_1, h_2, h_3.\mathbf{if}\ \mathit{False}\ \mathbf{then}\ \mathit{Right}_{h_1}\ \bullet\ \mathbf{else}\ \mathit{Left}_{h_2}\ \bullet : \\
\quad \forall t.h_1 : \mathrm{HasC}\ t\ \mathit{Right}\ \hat{\mathit{True}}, h_2 : \mathrm{HasC}\ t\ \mathit{Left}\ \hat{44}, h_3 : \mathrm{ConstrsOf}\ t \Rightarrow t
\end{array}
$$

specialization and constraint solving will be carried out without problems. However, if the code we apply it to is

$$
\begin{array}{l}
\vdash_{\mathrm{P}} \mathit{Left}^{D}\ 6^{S} : D^{D} \\
\quad \hookrightarrow \\
\quad \Lambda h_1, h_2, h_3, h_4.\mathit{Left}_{h_1}\ \bullet \\
\quad : \ \forall t, t'.h_1 : \mathrm{HasC}\ t\ \mathit{Left}\ \hat{6}, h_2 : \mathit{Right} \in t?\mathrm{HasC}\ t\ \mathit{Right}\ t', h_3 : \mathit{Right} \in t?\mathrm{IsBool}\ t', \\
\qquad\qquad h_4 : \mathrm{ConstrsOf}\ t \Rightarrow t
\end{array}
$$

specialization can proceed, but constraint solving will fail because it is impossible to unify $\hat{44}$ with $\hat{6}$ — which is forced by the entailment rules (UNIFY-HASC) and (HASC-GUARD).

EXAMPLE 3.29. In this example, we have two **case**'s. The first one receives information which is useful for the branch involving *Left*, while the second does not receive any information for its branch.

We have the following sum-types declarations

$$
\begin{array}{l}
\mathit{data}\ {D_1}^{D} = \mathit{Left}^{D}\ \mathit{Int}^{S} \mid \mathit{Right}^{D}\ \mathit{Bool}^{S} \\
\mathit{data}\ {D_2}^{D} = \mathit{Only}^{D}\ \mathit{Int}^{S}
\end{array}
$$

used in the source code

$$
\begin{array}{l}
\mathbf{let}^{D}\ f_1 = \lambda^{D} e.\mathbf{case}^{D}\, e\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{Left}^{D}\ n\ \rightarrow\ \mathbf{lift}\ n +^{D} 1^{D} \\
\qquad\qquad\qquad \mathit{Right}^{D}\ n\ \rightarrow\ 2^{D} \\
\ \mathbf{in}\ \mathbf{let}^{D}\ f_2 = \lambda^{D} e.\mathbf{case}^{D}\, e\ \mathbf{of} \\
\qquad\qquad\qquad\quad \mathit{Only}^{D}\ n\ \rightarrow\ \mathbf{lift}\ n \\
\quad \mathbf{in}\ \mathbf{let}^{D}\ g = f_1\ @^{D}\, (\mathit{Left}^{D}\ 10^{S}) \\
\qquad \mathbf{in}\ 4^{D} \\
: \mathit{Int}^{D}
\end{array}
$$

and one of its specializations is

$$
\begin{array}{l}
\Lambda\ h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9, h_{10}, h_{11}, h_{12}, h_{13}. \\
\quad \mathbf{let}\ f_1 = \lambda e.\mathbf{protocase}_v\ e\ \mathbf{with}\ h_5\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{Left}_{h_{11}}\ n\ \rightarrow\ 10+1 \\
\qquad\qquad\qquad \mathit{Right}_{h_3}\ n\ \rightarrow\ 2 \\
\quad\ \mathbf{in\ let}\ f_2 = \lambda e.\mathbf{protocase}_v\ e\ \mathbf{with}\ h_{10}\ \mathbf{of} \\
\qquad\qquad\qquad\quad \mathit{Only}_{h_8}\ n\ \rightarrow\ h_9 \\
\qquad \mathbf{in\ let}\ g = f_1@(\mathit{Left}_{h_{11}}\ \bullet) \\
\qquad\quad \mathbf{in}\ 4
\end{array}
$$

with residual type

$$
\begin{array}{l}
\forall t_1, t_2, t_3, t_4, t_5, t_6, t_7. \\
\ \ h_1 : \mathit{Right} \in t_1?\mathrm{HasC}\ t_1\ \mathit{Right}\ t_2, h_2 : \mathit{Right} \in t_1?\mathrm{IsBool}\ t_2, \\
\ \ h_3 : \mathit{Right} \in t_1?\mathrm{HasC}\ t_1\ \mathit{Right}\ t_3, h_4 : \mathit{Right} \in t_1?\mathrm{IsBool}\ t_3, h_5 : \mathrm{ConstrsOf}\ t_1, \\
\ \ h_6 : \mathit{Only} \in t_4?\mathrm{HasC}\ t_4\ \mathit{Only}\ t_5, h_7 : \mathit{Only} \in t_4?\mathrm{IsInt}\ t_5, \\
\ \ h_8 : \mathit{Only} \in t_4?\mathrm{HasC}\ t_4\ \mathit{Only}\ t_6, h_9 : \mathit{Only} \in t_4?\mathrm{IsInt}\ t_6, h_{10} : \mathrm{ConstrsOf}\ t_4, \\
\ \ h_{11} : \mathrm{HasC}\ t_1\ \mathit{Left}\ \hat{10}, h_{12} : \mathit{Right} \in t_1?\mathrm{HasC}\ t_1\ \mathit{Right}\ t_7, \\
\ \ h_{13} : \mathit{Right} \in t_1?\mathrm{IsBool}\ t_7 \Rightarrow \mathit{Int}
\end{array}
$$

Constraint solving will detect these facts and will proceed deleting the *Right* and *Only* branches, giving the following residual code

$$
\begin{array}{l}
\mathbf{data}\ D_1 = \mathit{Left}_1\ \hat{10} \\
\\
\mathbf{let}\ f_1 = \lambda e.\mathbf{case}\ e\ \mathbf{of} \\
\qquad\qquad\quad \mathit{Left}_1\ n\ \rightarrow\ 10+1 \\
\ \mathbf{in\ let}\ f_2 = \lambda e.\mathbf{error}\ \text{“There are no branches.”} \\
\qquad \mathbf{in\ let}\ g = f_1@(\mathit{Left}_1\ \bullet) \\
\qquad\quad \mathbf{in}\ 4
\end{array}
$$

We have eliminated unneeded branches from **case**s, not including *Right* in definition of D_1 because it will not be used, thus obtaining more concise code.

Chapter 4

The Algorithm and the Proof, Extended

“ — La primera lección en Roke, y la última, es ≪Haz lo que sea necesario≫. ¡Y no más!
— Las lecciones intermedias han de consistir, entonces, en aprender qué es lo necesario. ”

Los hijos de la mar abierta
La costa mas lejana
Úrsula K. Le Guin

In this chapter we extend the proof for *principality* of system P (Theorem 2.28) to include dynamic sum-types.

Basically, we have to make the following extensions: first, in Section 4.1, we extend $\vdash_{\rm S}$, a syntax directed version of $\vdash_{\rm P}$, to take account of dynamic sum-types, and we prove that both systems are still equivalent in the same way as before (Theorems 4.5 and 4.6). Then, in Section 4.2, we extend the algorithm $\vdash_{\rm W}$ and the proof that $\vdash_{\rm S}$ is still equivalent to $\vdash_{\rm W}$ (Theorems 4.13 and 4.14).

The main result is obtained as a corollary, combining the four theorems described above, so we do not need to extend its proof separately.

4.1 Extension of The Syntax Directed System, S

The syntax directed versions of rules (DCONSTR) and (DCASE) respect the equivalence between systems $\vdash_{\rm P}$ and $\vdash_{\rm S}$, in such a way that results on one of the systems can be translated into results of the other. Rules that belong to a syntax directed system are always better suited for an algorithm.

First, we must review some definitions and properties of system P, which were previously stated in [Martínez López and Hughes, 2004]. They are needed to introduce a way to generalize as much type variables as possible under a certain assignment.

Definition 4.1. Let $A = (FV(\Delta) \cup FV(\tau'))/(FV(\Gamma) \cup FV(\Delta'))$. We define

$$\mathrm{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau') = \forall A.\Delta \Rightarrow \tau'$$

When $\Delta' = \emptyset$, we may choose to write $\mathrm{Gen}_{\Gamma}(\Delta \Rightarrow \tau')$.

The correspondence of this notion of generalization with several applications of the rule (GEN) can be stated as the following property.

Proposition 4.2. *If* $\Delta' \mid \Gamma \vdash_{\rm P} e : \tau \hookrightarrow e' : \Delta \Rightarrow \tau'$*, then* $\Delta' \mid \Gamma \vdash_{\rm P} e : \tau \hookrightarrow e' : \mathrm{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau')$*, and both derivations only differ in the application of rule* (GEN)*.*

Additionally, type schemes obtained by generalization with Gen can be related by the ordering $\geq$, as stated in the following assertions that hold for all Γ and τ':

1. if $h' : \Delta' \Vdash v : \Delta$ and $C = [](\!(v)\!)$ then $C : \mathrm{Gen}_\Gamma(\Delta \Rightarrow \tau') \geq (h' : \Delta' \mid \tau')$
2. if $h' : \Delta' \Vdash v : \Delta$ and $C = \Lambda h'.[](\!(v)\!)$ then $C : \mathrm{Gen}_\Gamma(\Delta \Rightarrow \tau') \geq \mathrm{Gen}_\Gamma(\Delta' \Rightarrow \tau')$
3. for all substitutions R and all contexts Δ,
   $[] : R\,\mathrm{Gen}_\Gamma(\Delta \Rightarrow \tau') \geq \mathrm{Gen}_{R\Gamma}(R\,\Delta \Rightarrow R\,\tau')$

The system S does not produce type schemes but residual types only. The rules in Figure 3.4 do not contain either qualified types or type schemes, thus our extensions to system S are trivial — see Figure 4.1.

$$\text{(S-DCONSTR)}\ \frac{\begin{array}{l}\Delta_s \mid \Gamma \vdash_{\mathrm{S}} e : D(C^j) \ \hookrightarrow\ e'_s : \tau'_{j^s} \\ (\Delta^s_k \vdash_{\mathrm{SR}} D(C^k) \ \hookrightarrow\ \tau'_{k^s})_{k \in D \wedge k \neq j} \\ (\Delta_s \Vdash v^s_k : C^k \in \tau'_{e_s} ?\Delta^s_k, v^s_{m_k} : C^k \in \tau'_{e_s} ?\mathrm{HasC}\ \tau'_{e_s}\ \ C^k\ \ \tau'_{k^s})_{k \in D \wedge k \neq j} \\ \Delta_s \Vdash v^s_{m_j} : \mathrm{HasC}\ \tau'_{e_s}\ \ C^j\ \ \tau'_{j^s}, v^s_{cs} : \mathrm{ConstrsOf}\ \tau'_{e_s}\end{array}}{\Delta_s \mid \Gamma \vdash_{\mathrm{S}} C^{D}_j\ \ e : D^{D} \ \hookrightarrow\ C^j_{v^s_{m_j}}\ e'_s : \tau'_{j^s}}$$

$$\text{(S-DCASE)}\ \frac{\begin{array}{l}\Delta_s \vdash_{\mathrm{SR}} \tau \ \hookrightarrow\ \tau'_s \\ \Delta_s \mid \Gamma \vdash_{\mathrm{S}} e : D^{D} \ \hookrightarrow\ e'_s : \tau'_{e_s} \\ (h^s_k : \Delta^s_k \mid \Gamma \vdash_{\mathrm{S}} \lambda^{D} x_k.e_k : D(C^k) \rightarrow^{D} \tau \ \hookrightarrow\ \lambda x'_{k^s}.e'_{k^s} : \tau'_{k^s} \rightarrow \tau'_s)_{k \in B} \\ (\Delta_s \Vdash v^s_{m_k} : C^k \in \tau'_{e_s} ?\mathrm{HasC}\ \tau'_{e_s}\ \ C^k\ \ \tau'_{k^s}, v^s_k : C^k \in \tau'_{e_s} ?\Delta^s_k)_{k \in B} \\ \Delta_s \Vdash v^s_{cs} : \mathrm{ConstrsOf}\ \tau'_{e_s}\end{array}}{\begin{array}{ll}\Delta_s \mid \Gamma \vdash_{\mathrm{S}} & \mathbf{case}\ e\ \mathbf{of} \\ & \quad (C^{D}_k\ \ x_k\ \rightarrow\ e_k)_{k \in B} \\ & : \tau \\ & \hookrightarrow \\ & \mathbf{protocase}_v\ e'_s\ \mathbf{with}\ v^s_{cs}\ \mathbf{of} \\ & \quad (C^k_{v^s_{m_k}}\ \ x'_{k^s}\ \rightarrow\ e'_{k^s}[v^s_k/h^s_k])_{k \in B} \\ & : \tau'_s\end{array}}$$

Figure 4.1: Syntax Directed Specialisation Rules.

Now, we have to prove that our extensions to system S do not modify the following properties.

PROPOSITION 4.3. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow\ e' : \tau'$ *then* $h : S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow\ e' : S\,\tau'$

PROPOSITION 4.4. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow\ e' : \tau'$ *and* $\Delta' \Vdash v : \Delta$, *then*

$$\Delta' \mid \Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow\ e'[h/v] : \tau'$$

which show that system S is well behaved with respect to entailment and substitutions. Below, we are able to establish the equivalence between systems S and P.

THEOREM 4.5. *If* $\Delta \mid \Gamma \vdash_{\mathrm{S}} e : \tau \ \hookrightarrow\ e' : \tau'$ *then* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \ \hookrightarrow\ e' : \tau'$.

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

$$\frac{\tau_1 \sim^{U} \tau_1' \qquad U\,\tau_2 \sim^{V} U\,\tau_2'}{\text{HasC } \tau_1 \;\; C^k \;\; \tau_2 \sim^{VU} \text{HasC } \tau_1' \;\; C^k \;\; \tau_2'}$$

$$\frac{\tau \sim^{U} \tau' \qquad U\,\delta \sim^{V} U\,\delta'}{C^k \in \tau?\delta \sim^{VU} C^k \in \tau'?\delta'}$$

Figure 4.2: Rules for unification.

THEOREM 4.6. *If $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e' : \sigma$, then there exist $h'_s, \Delta'_s, e'_s, \tau'_s$, and C'_s such that*

a) $h'_s : \Delta'_s \mid \Gamma \vdash_{\mathrm{S}} e : \tau \;\hookrightarrow\; e'_s : \tau'_s$

b) $C'_s : \mathrm{Gen}_\Gamma(\Delta'_s \Rightarrow \tau'_s) \geq (h : \Delta \mid \sigma)$

c) $C'_s[\Lambda h'_s.e'_s] = e'$

Observe that Theorem 4.5 establishes that a derivation in S is also a derivation in P. But the converse is not true: not every derivation in P is a derivation in S; however, there is a way to relate derivations in both systems by using generalizations, conversions and the $\geq$ ordering, as stated in Theorem 4.6.

4.2 Extension of The Inference Algorithm, W

Here, we extend the algorithm presented in [Martínez López and Hughes, 2004] to construct a type specialization for a given source term containing dynamic sum-types, and prove that every specialization obtained by the extended $\vdash_{\mathrm{P}}$ can be expressed in terms of the output of this algorithm.

We also have to extend the following properties establishing that the result of unification, if it exists, is really a unifier.

PROPOSITION 4.7. *If $\sigma \sim^{U} \sigma'$ then $U\,\sigma = U\,\sigma'$.*

PROPOSITION 4.8. *If $S\,\sigma = S\,\sigma'$, then $\sigma \sim^{U} \sigma'$ and there exists a substitution T such that $S = TU$.*

4.2.1 An entailment algorithm

In addition to the entailment algorithm given in Section 2.3.4, we need to consider the following proposition — easily verified — for formal proofs.

PROPOSITION 4.9. *If $\Delta' \mid \Delta \Vdash_{\mathrm{W}} \delta$ then $\Delta', \Delta \Vdash \delta$.*

$$\text{(WSR-DDT)}\ \frac{(\Delta_{w_k} \vdash_{\text{W-SR}} D(C^k) \ \hookrightarrow\ \tau'_{w_k})_{k\in D}}{\begin{array}{l}\text{ConstrsOf } t,\\ (C^k \in t?\Delta_{w_k}, C^k \in t?\text{HasC } t\ \ C^k\ \ \tau'_{w_k})_{k\in D}\ \vdash_{\text{W-SR}} D^D\ \hookrightarrow\ t\end{array}}\ t\ \textit{fresh}$$

Figure 4.3: Rule calculating principal source-residual relationship

4.2.2 An algorithm for source-residual relationship

We add a rule to the algorithm described in Figure 2.12 so as to compute the residual type synthesized from a source type D^D — see Figure 4.3.

The following propositions are also extended to relate the algorithm $\vdash_{\text{W-SR}}$ with the specification of the $\vdash_{\text{SR}}$ but considering dynamic sum-types.

PROPOSITION 4.10. *If* $\Delta \vdash_{\text{W-SR}} \tau \ \hookrightarrow\ \tau'$ *then* $\Delta \vdash_{\text{SR}} \tau \ \hookrightarrow\ \tau'$.

PROPOSITION 4.11. *If* $\Delta \vdash_{\text{SR}} \tau \ \hookrightarrow\ \sigma$ *then* $\Delta'_w \vdash_{\text{W-SR}} \tau \ \hookrightarrow\ \tau'_w$ *with all the residual variables fresh, and there exists* C'_w *such that* $C'_w : \text{Gen}_\emptyset(\Delta'_w \Rightarrow \tau'_w) \geq (\Delta \mid \sigma)$.

This last property establishes that the residual type produced by $\vdash_{\text{W-SR}}$ can be generalized to a type that is more general than any other to which the source term can be specialized. This is important when using $\vdash_{\text{W-SR}}$ to constraint the type of a lambda-bound variable in rule (W-DLAM).

4.2.3 An algorithm for type specialisation

The rules that extend the algorithm W are shown in Figure 4.4. The rule (W-DCONSTR) has a similar structure to (S-DCONSTR) but involves some unifiers and the entailment algorithm. On the other hand, (W-DCASE) is complicated because of the need for a lot of unifiers to pass information between branches. On account of this complexity, we have introduced the abbreviation

$$A^m_n = \begin{cases} U^m_{m-1}\, T_{w_{m-1}} \cdots U^n_{n-1}\, T_{w_n}, & \text{if } n \leq m \\ \text{Id}, & \text{otherwise} \end{cases}$$

in (W-DCASE) to express complex compositions of unifiers. Moreover, the hypothesis of the (W-DCASE) must be ordered following the data dependencies.

The results obtained by W are equivalent, in the sense established in Theorems 4.13 and 4.14, to the results obtained by S. To establish the equivalence we will use, following [Martínez López and Hughes, 2004], a notion of similarity between substitutions defined as in [Jones, 1994], that is, two substitutions R and S are similar (written $R \approx S$), if they only differ in a finite number of new variables. This is useful in order to compare substitutions produced by the algorithm, given that it introduces several fresh variables that will be substituted.

We extend the following lemma and theorems

LEMMA 4.12. *If* $h : \Delta \mid S\,\Gamma \vdash_{\text{W}} e : \tau \ \hookrightarrow\ e' : \tau'$ *then* $EV(e') \subseteq h$

THEOREM 4.13. *If* $\Delta \mid S\,\Gamma \vdash_{\text{W}} e : \tau \ \hookrightarrow\ e' : \tau'$ *then* $\Delta \mid S\,\Gamma \vdash_{\text{S}} e : \tau \ \hookrightarrow\ e' : \tau'$.

$$
\text{(W-DCONSTR)}\;\frac{\begin{array}{l}\Delta_{w_j} \mid T_{w_j}\,\Gamma \vdash_{\mathrm{W}} e : D(C^j) \;\hookrightarrow\; e' : \tau'_{w_j} \\ (\Delta_{w_k} \vdash_{\mathrm{SR}} D(C^k) \;\hookrightarrow\; \tau'_{w_k})_{k\in D\wedge k\neq j} \\ \;\Delta_w \mid \Delta_{w_j} \Vdash_{\mathrm{W}} \\ \qquad (h^w_{v_k} : C^k \in t?\Delta_{w_k})_{k\in D\wedge k\neq j}, \\ \qquad (h^w_{v_{m_k}} : C^k \in t?\mathrm{HasC}\ t\ \ C^k\ \ \tau'_{w_k})_{k\in D\wedge k\neq j}, \\ \qquad h^w_{v_{m_j}} : \mathrm{HasC}\ t\ \ C^j\ \ \tau'_{w_j}, h^w_{v_{cs}} : \mathrm{ConstrsOf}\ t\end{array}}{\Delta_w, \Delta_{w_j} \mid T_{w_j}\,\Gamma \vdash_{\mathrm{W}} C^{D}_j\ \ e : D^{D} \;\hookrightarrow\; C^j_{h^w_{v_{m_j}}}\ \ e' : t}\; t\ \mathit{fresh}
$$

$$
\text{(W-DCASE)}\;\frac{\begin{array}{l}\Delta_{w_0} \vdash_{\text{W-SR}} \tau \;\hookrightarrow\; \tau'_{w^o_0} \\ \Delta_{w_e} \mid T_{w_e}\,\Gamma \vdash_{\mathrm{W}} e : D^{D} \;\hookrightarrow\; e'_w : \tau'_{w_e} \\ \;(h^w_{w_k} : \Delta_{w_k} \mid T_{w_k}\,(A^{k-1}_1\,T_{w_e}\,\Gamma) \\ \qquad \vdash_{\mathrm{W}} \lambda^{D} x_k.e_k : D(C^k) \to^{D} \tau \;\hookrightarrow\; \lambda x'_k.e'_{w_k} : \tau'_{w^i_k} \to \tau'_{w^o_k}\ )_{k=1\ldots B} \\ \tau'_{w^o_1} \sim^{U^1_0}\ \tau'_{w^o_0} \\ (\tau'_{w^o_k} \sim^{U^k_{k-1}}\ T_{w_k}\,U^{k-1}_{k-2}\,\tau'_{w^o_{k-1}})_{k=2\ldots B} \\ \;\Delta_w \mid A^B_1\,\Delta_{w_e}, A^B_2\,U^1_0\,\Delta_{w_0} \Vdash_{\mathrm{W}} \\ \quad (h^w_{v_{m_k}} : C^k \in A^B_1\,\tau'_{w_e}?\mathrm{HasC}\ (A^B_1\,\tau'_{w_e})\ \ C^k\ \ (A^B_{k+1}\,U^k_{k-1}\,\tau'_{w^i_k}))_{k=1\ldots B}, \\ \quad (h^w_{v_k} : C^k \in A^B_1\,\tau'_{w_e}?A^B_{k+1}\,U^k_{k-1}\,\Delta_{w_k})_{k=1\ldots B}, \\ \quad h^w_{v_{cs}} : \mathrm{ConstrsOf}\ A^B_1\,\tau'_{w_e}\end{array}}{\begin{array}{l}\Delta_w, A^B_1\,\Delta_{w_e}, A^B_2\,U^1_0\,\Delta_{w_0} \mid A^B_1\,T_{w_e}\ \Gamma \\ \vdash_{\mathrm{W}}\ \ \mathbf{case}\ e\ \mathbf{of} \\ \qquad\qquad (C^{D}_k\ \ x_k \;\to\; e_k)_{k=1\ldots B} \\ \qquad : \tau \\ \qquad \hookrightarrow \\ \qquad \mathbf{protocase}_v\ e'_w\ \mathbf{with}\ h^w_{v_{cs}}\ \mathbf{of} \\ \qquad\quad (C^k_{h^w_{v_{m_k}}}\ \ x'_k \;\to\; e'_{w_k}[h^w_{v_k}/h^w_{w_k}])_{k=1\ldots B} \\ \qquad : U^B_{B-1}\,\tau'_{w_B}\end{array}}
$$

Figure 4.4: Extension of Type Specialization Algorithm.

THEOREM 4.14. *If* $h : \Delta \mid S\,\Gamma \vdash_{\text{S}} e : \tau \hookrightarrow e' : \tau'$, *then* $h'_w : \Delta'_w \mid T'_w\,\Gamma \vdash_{\text{W}} e : \tau \hookrightarrow e'_w : \tau'_w$ *and there exists a substitution* R *and evidence* v'_w *such that*

a) $S \approx RT'_w$

b) $\tau' = R\,\tau'_w$

c) $h : \Delta \Vdash v'_w : R\,\Delta'_w$

d) $e' = e'_w[h'_w/v'_w]$

The meaning of this last theorem is that every residual term and type obtained by the syntax directed system can be expressed as a particular case of the residual term and type produced by the algorithm.

We are finally in a position to say that our extensions to Martínez López' work mantain the property of *principality* as it is stated in Theorem 2.28. To prove this we only need the properties that we have already proved here and in the preceding chapter. More details of the proof can be found in [Martínez López and Hughes, 2004].

4.3 Examples

In this section we present the principal type specialization of some examples as they were produced by the algorithm, with some simplification based on the entailment rules. It is notorious that the set of predicates produced is usually larger than the one expected, situation that is managed by the notions of *simplification* and *constraint solving* discussed in [Badenes and Martínez López, 2002].

EXAMPLE 4.15. A constructor can be applied to another constructor belonging to another dynamic sum-type. We use dynamic tuples to provide constructors with several different arguments.

Given the following declarations

$$\begin{array}{l} \mathit{data}\ D_A^{D} = \mathit{Only}_A^{D}\ (\mathit{Int}^{S}, D_B^{D})^{D} \\ \mathit{data}\ D_B^{D} = \mathit{Only}_B^{D}\ (\mathit{Bool}^{S}, D_C^{D})^{D} \\ \mathit{data}\ D_C^{D} = \mathit{Only}_C^{D}\ \mathit{Int}^{S} \end{array}$$

in the source code

$$\begin{array}{l} \mathbf{let}^{D}\ d_1 = \mathit{Only}_A^{D}\ (14^{S}, \mathit{Only}_B^{D}\ (\mathit{False}^{S}, \mathit{Only}_C^{D}\ 59^{S})^{D})^{D} \\ \ \mathbf{in\ let}^{D}\ d_2 = \mathit{Only}_B^{D}\ (\mathit{True}^{S}, \mathit{Only}_C^{D}\ 77^{S})^{D} \\ \quad\ \mathbf{in\ let}^{D}\ d_3 = \mathit{Only}_C^{D}\ 22^{S} \\ \qquad\ \mathbf{in}\ 4^{D} \\ : \mathit{Int}^{D}, \end{array}$$

its principal specialization is

$$\begin{array}{l} \Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9, h_{10}, h_{11}, h_{12}. \\ \ \mathbf{let}\ d_1 = \mathit{Only}^{A}_{h_1}(\bullet, \mathit{Only}^{B}_{h_3}(\bullet, \mathit{Only}^{C}_{h_5}\bullet)) \\ \ \ \mathbf{in\ let}\ d_2 = \mathit{Only}^{B}_{h_7}(\bullet, \mathit{Only}^{C}_{h_9}\bullet) \\ \quad\ \ \mathbf{in\ let}\ d_3 = \mathit{Only}^{C}_{h_{11}}\bullet \\ \qquad\ \ \mathbf{in}\ 4 \end{array}$$

with residual type

$$\begin{array}{l}\forall t_1, t_2, t_3, t_4, t_5, t_6.\\ \quad h_1 : \mathrm{HasC}\ t_1\ \mathit{Only}_A\ (\hat{14}, t_2), h_2 : \mathrm{ConstrsOf}\ t_1,\\ \quad h_3 : \mathrm{HasC}\ t_2\ \mathit{Only}_B\ (\hat{\mathit{False}}, t_3), h_4 : \mathrm{ConstrsOf}\ t_2,\\ \quad h_5 : \mathrm{HasC}\ t_3\ \mathit{Only}_C\ \hat{59}, h_6 : \mathrm{ConstrsOf}\ t_3,\\ \quad h_7 : \mathrm{HasC}\ t_4\ \mathit{Only}_B\ (\hat{\mathit{True}}, t_5), h_8 : \mathrm{ConstrsOf}\ t_4,\\ \quad h_9 : \mathrm{HasC}\ t_5\ \mathit{Only}_C\ \hat{77}, h_{10} : \mathrm{ConstrsOf}\ t_5,\\ \quad h_{11} : \mathrm{HasC}\ t_6\ \mathit{Only}_C\ \hat{22}, h_{12} : \mathrm{ConstrsOf}\ t_6 \Rightarrow \mathit{Int}\end{array}$$

Constraint solving produces the following declarations

$$\begin{array}{l}\mathbf{data}\ D_1 = \mathit{Only}_1^A(\hat{14}, D_2)\\ \mathbf{data}\ D_2 = \mathit{Only}_2^B(\hat{\mathit{False}}, D_3)\\ \mathbf{data}\ D_3 = \mathit{Only}_3^C\,\hat{59}\\ \mathbf{data}\ D_4 = \mathit{Only}_4^B(\hat{\mathit{True}}, D_5)\\ \mathbf{data}\ D_5 = \mathit{Only}_5^C\,\hat{77}\\ \mathbf{data}\ D_6 = \mathit{Only}_6^C\,\hat{22}\end{array}$$

and the residual code

$$\begin{array}{l}\mathbf{let}\ d_1 = \mathit{Only}_1^A(\bullet, \mathit{Only}_2^B(\bullet, \mathit{Only}_3^C\bullet))\\ \ \mathbf{in\ let}\ d_2 = \mathit{Only}_4^B(\bullet, \mathit{Only}_5^C\bullet)\\ \quad\ \mathbf{in\ let}\ d_3 = \mathit{Only}_6^C\bullet\\ \qquad\ \mathbf{in}\ 4\end{array}$$

The example shows how static information was removed while dynamic constructors were preserved in the residual code.

EXAMPLE 4.16. Here, we have an example where the residual type of constructor *Only*'s argument depends on an unknown static value. The rôle of the **if_then_else_** predicate is to differ the decision of which residual type has the *Only*'s argument until the value of b is known.

We have the dynamic sum-type declaration

$$\mathit{data}\ D^D = \mathit{Only}^D\ \mathit{Int}^S$$

used in the source code

$$\begin{array}{l}\lambda^D b.\mathit{Only}^D\ (\mathbf{if}^S\, b\ \mathbf{then}\ 3^S\ \mathbf{else}\ 5^S)\\ : \mathit{Bool}^S \to^D D^D\end{array}$$

and its principal specializacion is

$$\Lambda h_1, h_2, h_3, h_4, h_5.\lambda b.\mathit{Only}_{h_1}\ (\mathbf{if}_v\ h_2\ \mathbf{then}\ \bullet\ \mathbf{else}\ \bullet)$$

with the residual type

$$\begin{array}{l}\forall t_1, t_2, t_3.\\ \quad h_1 : \mathrm{HasC}\ t_2\ \mathit{Only}\ t_3,\\ \quad h_2 : \mathrm{IsBool}\ t_1.\\ \quad h_3 : t_3 := \mathrm{if}\ t_1\ \mathrm{then}\ \hat{3}\ \mathrm{else}\ \hat{5},\\ \quad h_4 : \mathrm{IsInt}\ t_3,\\ \quad h_5 : \mathrm{ConstrsOf}\ t_2 \Rightarrow t_1 \rightarrow t_2\end{array}$$

EXAMPLE 4.17. It is possible to make a polyvariant expression from a constructor — it should be remembered that they are treated as dynamic functions. Here, we can see how conversion h_1 will manipulate evidence h' in order to determine which residual instance of D^D will be considered.

We have the dynamic sum-type declaration

$$\mathit{data}\ D^D = \mathit{Only}^D\ \mathit{Int}^S$$

and the judgement

$$\begin{array}{l}\vdash_{\mathrm{P}} \mathbf{let}^D\ f = \mathbf{poly}\ \mathit{Only}^D\ \mathbf{in}\ 4^D\ : \mathit{Int}^D\\ \quad \hookrightarrow \Lambda h_1.\mathbf{let}\ f = h_1[\Lambda h, h', h''.\mathit{Only}_{h'}]\ \mathbf{in}\ 4\\ \quad : \forall s.\mathrm{IsMG}\ (\forall t, t'.h : \mathrm{IsInt}\ t, h' : \mathrm{HasC}\ t'\ \mathit{Only}\ t, h'' : \mathrm{ConstrsOf}\ t' \Rightarrow t \rightarrow t')\ s \Rightarrow \mathit{Int}\end{array}$$

Constraint solving produces

$$\mathbf{let}\ f = \Lambda h, h', h''.\mathit{Only}_{h'}\ \mathbf{in}\ 4$$

where there is no declaration of residual sum-types; this happens because constructor *Only* was applied to no argument. The variable f in the residual code will be reduced to constant $\bullet$ in the *evidence elimination* stage.

EXAMPLE 4.18. As in Example 4.17, we have a polyvariant constructor, but this time it is used twice, generating in this way two lower bounds to the type variable s.

We have the dynamic sum-type declaration

$$\mathit{data}\ D^D = \mathit{Only}^D\ \mathit{Int}^S$$

used in the source code

$$\begin{array}{l}\mathbf{let}^D\ f = \mathbf{poly}\ \mathit{Only}^D\\ \ \mathbf{in}\ \mathbf{let}^D\ d_1 = \mathbf{spec}\ f\ @^D\, 49^S\\ \quad\quad \mathbf{in}\ \mathbf{let}^D\ d_2 = \mathbf{spec}\ f\ @^D\, 62^S\\ \quad\quad\quad\quad \mathbf{in}\ 4^D\\ : \mathit{Int}^D\end{array}$$

and its principal specialization is

$$\begin{array}{l}\Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9.\\ \quad \mathbf{let}\ f = h_1[\Lambda h, h', h''.\mathit{Only}_{h'}]\\ \quad\ \mathbf{in}\ \mathbf{let}\ d_1 = h_5[f]@\bullet\\ \quad\quad\quad \mathbf{in}\ \mathbf{let}\ d_2 = h_9[f]@\bullet\\ \quad\quad\quad\quad\quad \mathbf{in}\ 4\end{array}$$

with the residual type

$$
\begin{array}{l}
\forall s, t_1, t_2, t_3, t_4. \\
\quad h_1 : \ \text{IsMG}\ (\forall t, t'. \\
\qquad\qquad\qquad h : \text{IsInt}\ t, \\
\qquad\qquad\qquad h' : \text{HasC}\ t'\ \mathit{Only}\, t, \\
\qquad\qquad\qquad h'' : \text{ConstrsOf}\ t' \Rightarrow t \rightarrow t') \\
\qquad\qquad\qquad s, \\
\quad h_2 : \mathit{Only} \in t_1 ?\text{HasC}\ t_1\ \mathit{Only}\, t_2, \\
\quad h_3 : \mathit{Only} \in t_1 ?\text{IsInt}\ t_2, \\
\quad h_4 : \text{ConstrsOf}\ t_1, \\
\quad h_5 : \text{IsMG}\ s\ (\hat{49} \rightarrow t_1), \\
\quad h_6 : \mathit{Only} \in t_3 ?\text{HasC}\ t_3\ \mathit{Only}\, t_4, \\
\quad h_7 : \mathit{Only} \in t_3 ?\text{IsInt}\ t_4, \\
\quad h_8 : \text{ConstrsOf}\ t_3, \\
\quad h_9 : \text{IsMG}\ s\ (\hat{62} \rightarrow t_3) \Rightarrow \mathit{Int}
\end{array}
$$

Constraint solving produces the sum-type declarations

$$
\begin{array}{l}
\textbf{data}\ D_1 = \mathit{Only}_1\ \hat{62} \\
\textbf{data}\ D_2 = \mathit{Only}_2\ \hat{49}
\end{array}
$$

and the residual code

$$
\begin{array}{l}
\textbf{let}\ f = \Lambda h, h', h''. \mathit{Only}_{h'} \\
\ \textbf{in let}\ d_1 = f((49, 2, \{\mathit{Only}\}))@\bullet \\
\qquad \textbf{in let}\ d_2 = f((62, 1, \{\mathit{Only}\}))@\bullet \\
\qquad\quad \textbf{in}\ 4
\end{array}
$$

The first evidence given to f proves a predicate IsInt while the second one proves a predicate HasC indicating the residual instance of D^D considered.

Example 4.19. In this example we have a polyvariant constructor used twice. Definition of source variable d_1 is responsible for forcing the constructors' arguments to be $\hat{2}$ and $\hat{3}$ respectively. Additionally, the definition of d_2 only involves information concerning one constructor; thus all the branches of c_1 will be preserved while some branches of c_2 will be deleted. On the other hand, set of constructor names appears as evidence to prove the predicate ConstrsOf. More predicates than expected are generated, but some of them are proved with evidence $\bullet$ and others are just replaced by simpler ones at constraint solving.

We have the dynamic sum-type declaration

$$\mathit{data}\ D^D = \mathit{Left}^D\ \mathit{Int}^S \mid \mathit{Right}^D\ \mathit{Int}^S$$

in the source code

$$
\begin{array}{l}
\mathbf{let}^D\ f = \mathbf{poly}\ \mathit{Left}^D \\
\ \mathbf{in\ let}^D\ d_1 = \mathbf{if}^D\ \mathit{True}\ \mathbf{then\ spec}\ f\ @^D\, 85^S\ \mathbf{else}\ \mathit{Right}^D\ 15^S \\
\quad \mathbf{in\ let}^D\ d_2 = \mathbf{spec}\ f\ @^D\, 92^S \\
\qquad \mathbf{in\ let}^D\ c_1 = \mathbf{case}^D\, d_1\ \mathbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Left}^D\ n\ \rightarrow\ \mathbf{lift}\ n \\
\qquad\qquad\qquad\qquad \mathit{Right}^D\ n\ \rightarrow\ \mathbf{lift}\ n \\
\qquad\quad \mathbf{in\ let}^D\ c_2 = \mathbf{case}^D\, d_2\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\quad \mathit{Left}^D\ n\ \rightarrow\ \mathbf{lift}\ n \\
\qquad\qquad\qquad\qquad\quad \mathit{Right}^D\ n\ \rightarrow\ \mathbf{lift}\ n \\
\qquad\qquad \mathbf{in}\ 4^D \\
: \mathit{Int}^D
\end{array}
$$

and its principal specialization is

$$
\begin{array}{l}
\Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9, h_{10}, h_{11}, h_{12}, h_{13}, h_{14}, h_{15}, h_{16}, h_{17}, h_{18}, h_{19}, h_{20}. \\
\ \mathbf{let}\ f = h_1[\Lambda h_a, h_b, h_c, h_d, h_e.\mathit{Left}_{h_b}] \\
\ \ \mathbf{in\ let}\ d_1 = \mathbf{if}\ \mathit{True}\ \mathbf{then}\ h_4[f]@\bullet\ \mathbf{else}\ \mathit{Right}_{h_5}\ \bullet \\
\quad \mathbf{in\ let}\ d_2 = h_{14}[f]@\bullet \\
\qquad \mathbf{in\ let}\ c_1 = \mathbf{protocase}_v\ d_1\ \mathbf{with}\ h_8\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{Left}_{h_{15}}\ n\ \rightarrow\ h_{16} \\
\qquad\qquad\qquad \mathit{Right}_{h_5}\ n\ \rightarrow\ 15 \\
\qquad\quad \mathbf{in\ let}\ c_2 = \mathbf{protocase}_v\ d_2\ \mathbf{with}\ h_{13}\ \mathbf{of} \\
\qquad\qquad\qquad\quad \mathit{Left}_{h_{17}}\ n\ \rightarrow\ h_{18} \\
\qquad\qquad\qquad\quad \mathit{Right}_{h_{19}}\ n\ \rightarrow\ h_{20} \\
\qquad\qquad \mathbf{in}\ 4
\end{array}
$$

with the residual type

$$
\begin{array}{l}
\forall s, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9. \\
\quad h_1 : \mathrm{IsMG}\ (\forall t_1, t_2, t_3. \\
\qquad\qquad\qquad h_a : \mathrm{IsInt}\ t_1, \\
\qquad\qquad\qquad h_b : \mathrm{HasC}\ t_2\ \mathit{Left}\ t_1, \\
\qquad\qquad\qquad h_c : \mathit{Right} \in t_2 ?\mathrm{HasC}\ t_2\ \mathit{Right}\ t_3, \\
\qquad\qquad\qquad h_d : \mathit{Right} \in t_2 ?\mathrm{HasC}\ t_2\ \mathit{Right}\ \mathrm{IsInt}\ t_3, \\
\qquad\qquad\qquad h_e : \mathrm{ConstrsOf}\ t_2 \Rightarrow t_1 \rightarrow t_2)\ s, \\
\quad h_2 : \mathit{Left} \in t_1 ?\mathrm{HasC}\ t_1\ \mathit{Left}\ t_2, h_3 : \mathit{Left} \in t_1 ?\mathrm{IsInt}\ t_2, \\
\quad h_4 : \mathrm{IsMG}\ s\ (\hat{85} \rightarrow t_1), \\
\quad h_5 : \mathrm{HasC}\ t_1\ \mathit{Right}\ \hat{15}, \\
\quad h_6 : \mathit{Left} \in t_1 ?\mathrm{HasC}\ t_1\ \mathit{Left}\ t_3, h_7 : \mathit{Left} \in t_1 ?\mathrm{IsInt}\ t_3, \\
\quad h_8 : \mathrm{ConstrsOf}\ t_1, \\
\quad h_9 : \mathit{Left} \in t_4 ?\mathrm{HasC}\ t_4\ \mathit{Left}\ t_5, h_{10} : \mathit{Left} \in t_4 ?\mathrm{IsInt}\ t_5, \\
\quad h_{11} : \mathit{Right} \in t_4 ?\mathrm{HasC}\ t_4\ \mathit{Right}\ t_6, h_{12} : \mathit{Right} \in t_4 ?\mathrm{IsInt}\ t_6, \\
\quad h_{13} : \mathrm{ConstrsOf}\ t_4, \\
\quad h_{14} : \mathrm{IsMG}\ s\ (\hat{92} \rightarrow t_4), \\
\quad h_{15} : \mathit{Left} \in t_1 ?\mathrm{HasC}\ t_1\ \mathit{Left}\ t_7, h_{16} : \mathit{Left} \in t_1 ?\mathrm{IsInt}\ t_7, \\
\quad h_{17} : \mathit{Left} \in t_4 ?\mathrm{HasC}\ t_4\ \mathit{Left}\ t_8, h_{18} : \mathit{Left} \in t_4 ?\mathrm{IsInt}\ t_8, \\
\quad h_{19} : \mathit{Right} \in t_4 ?\mathrm{HasC}\ t_4\ \mathit{Right}\ t_9, h_{20} : \mathit{Right} \in t_4 ?\mathrm{IsInt}\ t_9 \Rightarrow \mathit{Int}
\end{array}
$$

Constraint solving produces the sum-type declarations

$$\begin{array}{l}\textbf{data } D_1 = \mathit{Left}_1\ \hat{92} \\ \textbf{data } D_2 = \mathit{Left}_2\ \hat{85} \mid \mathit{Right}_2\ \hat{15}\end{array}$$

and the residual code

$$\begin{array}{l}
\textbf{let } f = \Lambda h_a, h_b, h_c, h_d, h_e.\mathit{Left}_{h_b} \\
\ \textbf{in let } d_1 = \textbf{if } \mathit{True} \textbf{ then } f((85, 2, 2, 15, \{\mathit{Left}, \mathit{Right}\}))@\bullet\ \ \textbf{else } \mathit{Right}_2\ \bullet \\
\quad \textbf{in let } d_2 = f((92, 1, \bullet, \bullet, \{\mathit{Left}\}))@\bullet \\
\qquad \textbf{in let } c_1 = \textbf{case } d_1 \textbf{ of} \\
\qquad\qquad\qquad \mathit{Left}_2\ n\ \rightarrow\ 85 \\
\qquad\qquad\qquad \mathit{Right}_2\ n\ \rightarrow\ 15 \\
\qquad\quad \textbf{in let } c_2 = \textbf{case } d_2 \textbf{ of} \\
\qquad\qquad\qquad\quad \mathit{Left}_1\ n\ \rightarrow\ 92 \\
\qquad\qquad \textbf{in } 4
\end{array}$$

The first two evidences for f concern the constructor *Left* and the other two, the constructor *Right*. We can appreciate that *Right*'s argument does not have any residual type in definition of d_2 (looking at evidence $\bullet$). The predicates correponding to evidence variables h_8 and h_{13} were proved with $\{\mathit{Left}, \mathit{Right}\}$ and $\{\mathit{Left}\}$ respectively.

Example 4.20. This is an interesting example showing how the unification produced by construct $\textbf{if}^D$ is responsible for forcing the constructor *Just* to be applied to a function with type $(\forall t.\text{IsInt } t \Rightarrow t \rightarrow t \rightarrow t)$.

We have the following sum-type declarations

$$\begin{array}{l}\mathit{data}\ {D_1}^D = \mathit{Only}^D\ \ {D_2}^D \\ \mathit{data}\ {D_2}^D = \mathit{Just}^D\ \textbf{poly}\ (\mathit{Int}^S \rightarrow^D \mathit{Int}^S \rightarrow^D \mathit{Int}^S)\end{array}$$

in the source code

$$\begin{array}{l}
\textbf{let}^D\ \ \mathit{dummy} = \textbf{if}^D\ \mathit{True} \\
\qquad\qquad\qquad\quad \textbf{then } \mathit{Only}^D\ \ (\mathit{Just}^D\ \ (\textbf{poly}\ (\lambda^D xy.y))) \\
\qquad\qquad\qquad\quad \textbf{else } \mathit{Only}^D\ \ (\mathit{Just}^D\ \ (\textbf{poly}\ (\lambda^D xy.x))) \\
\ \textbf{in } 4^D \\
: \mathit{Int}^D
\end{array}$$

and its principal specialization is

$$\begin{array}{l}
\Lambda h_1, h_2, h_3, h_4, h_5, h_6. \\
\ \textbf{let } \mathit{dummy} =\ \ \textbf{if } \mathit{True} \\
\qquad\qquad\qquad\quad \textbf{then } \mathit{Only}_{h_2}\ (\mathit{Just}_{h_4}\ (h_1[\Lambda h_d, h_e.\lambda x'y'.y'])) \\
\qquad\qquad\qquad\quad \textbf{else } \mathit{Only}_{h_2}\ (\mathit{Just}_{h_4}\ (h_6[\Lambda h_f, h_g.\lambda x'y'.x'])) \\
\ \textbf{in } 4
\end{array}$$

with residual type

$$
\begin{array}{l}
\forall s, t, t'. \\
\ h_1 : \mathrm{IsMG}\ (\forall t, t'.h_d : \mathrm{IsInt}\ t, h_e : \mathrm{IsInt}\ t' \Rightarrow t \to t' \to t')\ s, \\
\ h_2 : \mathrm{HasC}\ t_1\ \mathit{Only}\ t_2, \\
\ h_3 : \mathrm{ConstrsOf}\ t_1, \\
\ h_4 : \mathrm{HasC}\ t_2\ \mathit{Just}\ (\mathbf{poly}\ s), \\
\ h_5 : \mathrm{ConstrsOf}\ t_2, \\
\ h_6 : \mathrm{IsMG}\ (\forall t, t'.h_f : \mathrm{IsInt}\ t, h_g : \mathrm{IsInt}\ t' \Rightarrow t \to t' \to t)\ s \Rightarrow \mathit{Int}
\end{array}
$$

Constraint solving produces

$$
\begin{array}{l}
\mathbf{data}\ D_1 = \mathit{Only}_1\ D_2 \\
\mathbf{data}\ D_2 = \mathit{Just}_2\ \mathbf{poly}\ (\forall t.h : \mathrm{IsInt}\ t \Rightarrow t \to t \to t) \\
\\
\mathbf{let}\ \mathit{dummy} = \ \mathbf{if}\ \mathit{True} \\
\qquad\qquad\quad \mathbf{then}\ \mathit{Only}_1\ (\mathit{Just}_2\ (\Lambda h.\lambda x' y'.y')) \\
\qquad\qquad\quad \mathbf{else}\ \mathit{Only}_1\ (\mathit{Just}_2\ (\Lambda h.\lambda x' y'.x')) \\
\ \mathbf{in}\ 4
\end{array}
$$

Observe the use of **poly** in the declaration of the residual sum-type D_2.

EXAMPLE 4.21. Here, we can observe the interaction between static functions and dynamic sum-types. We define a static function that returns the same constructor but applied to different static values. In the residual code, f can return two different values belonging to different instances of D^D. Observe that the residual of f is the constant $\bullet$ — this is because it has no free variables. Again, as Example 4.19 shows, there are several predicates but they will be managed by constraint solving.

Given the following sum-type declaration

$$\mathit{data}\ D^D = \mathit{Only}^D\ \ \mathit{Int}^S$$

in the source code

$$
\begin{array}{l}
\mathbf{let}^D\ f = \lambda^S b.\mathbf{if}^S\, b\ \mathbf{then}\ \mathit{Only}^D\ 33^S\ \mathbf{else}\ \mathit{Only}^D\ 72^S \\
\ \mathbf{in}\ \mathbf{let}^D\ p = (f\ @^S\ \mathit{True}^S, f\ @^S\ \mathit{False}^S)^D \\
\qquad \mathbf{in}\ 4^D \\
: \mathit{Int}^D
\end{array}
$$

its principal specialization is

$$
\begin{array}{l}
\Lambda h_1, h_2, h_3, h_4, h_5, h_{6,7}, h_8, h_9. \\
\ \ \mathbf{let}\ f = \bullet \\
\ \ \ \mathbf{in}\ \mathbf{let}\ p = (h_5 @_v f @_v \bullet, h_9 @_v f @_v \bullet) \\
\qquad \mathbf{in}\ 4
\end{array}
$$

with the residual type

$$
\begin{array}{l}
\forall t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9. \\
\quad \text{IsFunS } \mathbf{clos}(\Lambda h_a, h_b, h_c, h_d, h_e, h_f, h_g, h_h, h_i, h_j, h_k, h_l, h_m. \\
\qquad\qquad \lambda f'.\lambda b'.\mathbf{if}_v\ h_a\ \mathbf{then}\ \mathit{Only}_{h_g}\ \bullet\ \mathbf{else}\ \mathit{Only}_{h_k}\ \bullet \\
\qquad\qquad :\ \forall t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9. \\
\qquad\qquad\quad h_a : \text{IsBool } t_1, \\
\qquad\qquad\quad h_b : t_2 := \text{if } t_1 \text{ then } t_3 \text{ else } t_4, \\
\qquad\qquad\quad h_c : \mathit{Only} \in t_2?\text{HasC } t_2\ \mathit{Only}\ t_5, \\
\qquad\qquad\quad h_d : \mathit{Only} \in t_2?\text{IsInt } t_5, \\
\qquad\qquad\quad h_e : \text{ConstrsOf } t_2, \\
\qquad\qquad\quad h_f : t_1?\text{IsInt } t_6, \\
\qquad\qquad\quad h_g : t_1?\text{HasC } t_7\ \mathit{Only}\ t_6, \\
\qquad\qquad\quad h_h : t_1?\text{ConstrsOf } t_7, \\
\qquad\qquad\quad h_i : t_1?(t_6 \to t_7) \sim (\hat{33} \to t_3), \\
\qquad\qquad\quad h_j : !t_1?\text{IsInt } t_8, \\
\qquad\qquad\quad h_k : !t_1?\text{HasC } t_9\ \mathit{Only}\ t_8, \\
\qquad\qquad\quad h_l : !t_1?\text{ConstrsOf } t_9, \\
\qquad\qquad\quad h_m : !t_1?(t_8 \to t_9) \sim (\hat{72} \to t_4) \Rightarrow t_1 \to t_2) \\
\qquad\qquad t_1, \\
h_2 : \mathit{Only} \in t_2?\text{HasC } t_2\ \mathit{Only}\ t_3, \\
h_3 : \mathit{Only} \in t_2?\text{IsInt } t_3, \\
h_4 : \text{ConstrsOf } t_2, \\
h_5 : \text{IsFunS } t\ \mathbf{clos}(t_5 : \hat{\mathit{True}} \to t_2), \\
h_6 : \mathit{Only} \in t_6?\text{HasC } t_6\ \mathit{Only}\ t_7, \\
h_7 : \mathit{Only} \in t_6?\text{IsInt } t_7, \\
h_8 : \text{ConstrsOf } t_6, \\
h_9 : \text{IsFunS } t\ \mathbf{clos}(t_9 : \hat{\mathit{False}} \to t_6) \Rightarrow \mathit{Int}
\end{array}
$$

Constraint solving produces the sum-type declarations

$$
\begin{array}{l}
\mathbf{data}\ D_1 = \mathit{Only}_1\ \hat{33} \\
\mathbf{data}\ D_2 = \mathit{Only}_2\ \hat{72}
\end{array}
$$

used in

$$
\begin{array}{l}
\mathbf{let}\ f = \bullet \\
\ \mathbf{in\ let}\ p = (\mathit{Only}_1\ \bullet, \mathit{Only}_2\ \bullet) \\
\qquad \mathbf{in}\ 4 \\
: \mathit{Int}
\end{array}
$$

Note how the static applications were replaced by their results.

EXAMPLE 4.22. We have a recursive static function that generates as many applications of the constructor *Only* as the value on its input argument. We only show here the source code and the produced residual code at constraint solving, excluding the predicates generated because they are too many. To get a grasp of the number of predicates, this example can be tried with the provided prototype.

$$
\begin{array}{l}
\mathit{data}\ D^{D} = \mathit{Only}^{D}\ \mathit{Int}^{S} \\
\\
\mathbf{let}^{D}\ f = \mathbf{fix}^{S}\ (\lambda^{S} g.\lambda^{S} n.\mathbf{if}^{S}\ n ==^{S} 0^{S} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ 4^{D} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else\ let}^{D}\ d = \mathit{Only}^{D}\ n \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ g\ @^{S}\,(n -^{S} 1^{S})) \\
\ \mathbf{in}\ f\ @^{S}\, 5^{S}
\end{array}
$$

Constraint solving produces

$$
\begin{array}{l}
\mathbf{data}\ D_1 = \mathit{Only}_1\ \hat{1} \\
\mathbf{data}\ D_2 = \mathit{Only}_2\ \hat{2} \\
\mathbf{data}\ D_3 = \mathit{Only}_3\ \hat{3} \\
\mathbf{data}\ D_4 = \mathit{Only}_4\ \hat{4} \\
\mathbf{data}\ D_5 = \mathit{Only}_5\ \hat{5} \\
\\
\mathbf{let}\ f = \bullet \\
\ \mathbf{in\ let}\ d = \mathit{Only}_5\ \bullet \\
\quad\ \ \mathbf{in\ let}\ d = \mathit{Only}_4\ \bullet \\
\qquad\ \ \mathbf{in\ let}\ d = \mathit{Only}_3\ \bullet \\
\qquad\quad\ \ \mathbf{in\ let}\ d = \mathit{Only}_2\ \bullet \\
\qquad\qquad\ \ \mathbf{in\ let}\ d = \mathit{Only}_1\ \bullet \\
\qquad\qquad\quad\ \ \mathbf{in}\ 4
\end{array}
$$

Chapter 5

Extension to the Prototype

" En cambio ahora creo que lo importante es el primer borrador; lo demás es cuestón de técnica, de aligerar las frases, evitar repeticiones."

Jorge Luis Borges
Jorge Luis Borges habla de los demás
Confirmado, Número 240, 1970

5.1 Implementation Language

The implementation of the prototype was written in the functional language *Haskell* [Peyton Jones and Hughes (editors), 1999]. This prototype was developed using the interpreter *Hugs* [Jones and Reid, 1994-2003] under GNU/Linux [Torvalds and Stallman, 2004]; but when efficiency was needed, `ghc` [The University Court of the University of Glasgow, 2004] was chosen.

Working on functional languages enables us to represent data structures in exactly the same way as we think about them, which is ideal for writing our language and specialization constructs. Additionally, referencial transparency and programming with equations make definitions of pre- and post-conditions easier than other paradigms — for example imperative or object oriented programming, where the semantics of each contruct depends on a global state and not just on its input arguments.

The specializer was based on a state monad [Wadler, 1995] which has a lot of information concerning the specialization state (variables, substitutions, etc.). The monadic style of programming abstracts the carried state and the side-effects produced, allowing us to concentrate better on the goal of each function.

5.2 Previous Work

The prototype used as a stepping stone was implemented in [Martínez López and Hughes, 2004] and extended in [Badenes and Martínez López, 2002], which already has data structures to represent source and residual terms and types, type schemes and predicates, in addition to all the other elements handled by the specialization process. The implemented functionality includes the kernel of the *specializer*, which specializes source terms according to the algorithm W specified in Figure 2.10, and the post-processing phases: *simplification* and *constraint solving* [Badenes and Martínez López, 2002], *evidence elimination* [Badenes, 2003].

5.3 Extensions

The prototype described implements relations and algorithms that we briefly recalled in Chapter 2. In order to add dynamic sum-types, it was necessary to extend every module of the implementation.

In what follows, we summarize each module extended by us, describing also all the capabilities of the prototype.

- *parsing source language*, defining internal representation of source and residual terms and types — see Definition 3.1, Definition 3.2, Definition 2.2 and Definition 2.18 — and pretty-printing functions.
- *infering and checking types* of both languages and the source-residual relationship — see Figures 3.1 and 4.3.
- *principal type specialization* of source terms, using W algorithm described in Figures 2.10 and 4.4, producing residual programs with qualified types and evidence.
- *simplification* of constraints as a needed intermediate stage for achieving efficiency, implementing basically the rules described in [Badenes and Martínez López, 2002]. The entailment rules concerning dynamic sum-types — see Figure 3.2 — were implemented, but their formalization into the formal system developed in [Badenes and Martínez López, 2002] was left out the scope of this work.
- *constraint solving* is performed according to the heuristics described in [Badenes and Martínez López, 2002]. An heuristic to solve constraints that involve predicates HasC, ConstrsOf and $C \in \tau'?\Delta$ was also implemented, but as before, its formalization was left out the scope of this work.

5.4 Potential Improvements

The prototype is really far from its final version. Many improvements remain to be done. First, we could optimize the algorithms already implemented: the efficiency of the prototype is not optimal. This task may involve *profiling* in order to find points in the code to speed up, e.g. memorizing values without recalculating them, finding points where we could force a function to be strict (avoiding lazy evaluation) so as to save up heap space, or even reimplementing all the prototype in other language, such as ML or an imperative or object oriented one.

Another possible improvement is to make extensions to the source language so as to specialize terms that would be closer to those used daily by programmers, thus reducing the distance between the prototype as a laboratory toy and the prototype as a really productive tool in a specific domain.

5.5 Conclusions of the Implementation

One important contribution of the implementation phase was "putting into effect" many of the concepts developed in the theory.

The execution of the prototype was a key element to view the results produced when specialization rules were applied, in particular, the rules developed in this work. We were able to fine-tune our definitions, which were introduced in a theoretical world. Sometimes, we found inconveniences in our definitions that were incompatible with the rest of the work, detecting these problems only in the implementation stage.

It is very important to put into effect ideas that come from a theoretical framework in order to obtain a continuous feedback between the theory and the practice, two worlds that must fit together.

Chapter 6

Conclusions and Future Work

" Hay que sopesar los argumentos de uno y otro bando para determinar su consistencia y plantear supuestos prácticos, puramente hipotéticos en más de un caso. Si pareciera que algunos de estos supuestos van demasiado lejos, solicitamos del lector que tenga paciencia, pues estamos tratando de forzar las diversas posturas hasta su punto de ruptura a fin de advertir sus debilidades y fallos."

Aborto :
¿es posible tomar al mismo tiempo partido por la vida y la elección?
Miles de Millones
Carl Sagan

Type specialization, which was created by John Hughes [Hughes, 1996], was presented as a new and alternative approach to overcome some inherited limits.

Martínez López' work [Martínez López and Hughes, 2004] continued that work developing a framework that includes the use of qualified types [Jones, 1994] to capture the notion of *principality*, which means that any possible specialization of a given source term could be obtained as an instance of its most general specialization.

We summarize the principal contribution of this work as the introduction of dynamic sum-types (data types without recursion) to the source language described in [Martínez López and Hughes, 2004] preserving *principality*.

In Chapters 3 and 4 were introduced all the technical tools needed to deal with a new source language construct (extension of the source and residual languages grammars, new source and residual types definitions, new predicates capturing facts about dynamic sum-types, etc.) as well as how the specialization of dynamic constructors and **case**s have to be carried out by the *specializer*. We also extended the systems P and S, and the algorithm W, proving formally that *principality* was preserved. Additionally, we also implement our theoretical ideas into the existing prototype of [Martínez López and Hughes, 2004].

The information captured by the predicates introduced in Chapter 3 give us the following advantages,

- detect which constructors are used and which are not in the source program, obtaining definitions of residual sum-types with only the useful constructors; thus eliminating useless ones.
- erase branches from **case**s which are never executed, leaving only the needed ones.
- freely combine polyvariance with dynamic sum-types.

One disadvantage of this approach is that residual sum-types are monomorphic, avoiding polymorphism inside of sum-type definitions in order to obtain a more efficient and simpler residual language. Despite efficiency and simplicity, this is a new inherited limit imposed by our heuristic used at *constraint solving* phase and it will be a future work to eliminate it.

Some aspects related to the whole process of specialization were left as future work; for example, we do not specify formally the *simplification* and *constraint solving* rules — even though they were implemented in the prototype. So, there is a theoretical work to be done in the future using the framework developed in [Badenes and Martínez López, 2002].

This work is a first step towards the inclusion of dynamic recursive sum-types to the principal specialization process.

Bibliography

“ El camino es largo y lleno de dificultades. A veces, por extraviar la ruta, hay que retroceder; otras, por caminar demasiado aprisa, nos separamos de las masas; en ocasiones, por hacerlo lentamente, sentimos el aliento cercano de los que nos pisan los talones. Es nuestra ambición de revolucionarios, tratamos de caminar tan aprisa como sea posible, abriendo caminos, pero sabemos que tenemos que nutrirnos de las masa y que ésta sólo podrá avanzar más rápido si la alentamos con nuestro ejemplo.”

La Educación Directa, El Socialismo y el Hombre en Cuba (semanario *Marcha*, 12 de Marzo de 1965)
Ernesto Che Guevara - La Revolución, Escritos Esenciales
Selección y Prólogo de Marcos Mayer

[Badenes and Martínez López, 2002] Hernán Badenes and Pablo E. Martínez López. Simplifying and solving qualified types for principal type specification. In *WAIT 2002, Argentine Workshop on Theoretical Computer Science*, 2002.

[Badenes, 2003] Hernán Badenes. Simplifying and solving qualified types for principal type specification, 2003. Licentiate Final Work, UNLP.

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

[Breazu-Tannen *et al.*, 1991] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[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.

[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.

[Danvy *et al.*, 1996] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Selected papers of the International Seminar “Partial Evaluation”*, volume 1110 of *Lecture Notes in Computer Science*, Dagstuhl, Germany, February 1996. Springer-Verlag, Heidelberg, Germany.

[Futamura, 1971] Yoshihiko Futamura. Partial evaluation of computation process - An approach to a compiler-compiler. *Computer, Systems, Controls*, 2(5):45–50, 1971.

[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.

[Hughes, 1996] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Danvy et al. [1996], pages 183–215.

[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 and Reid, 1994-2003] Mark P Jones and Alastair Reid. Haskell interpreter hugs98, 1994-2003. `http://www.haskell.org/hugs/`.

[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. In Bjørner et al. [1988], pages 1–14.

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

[Martínez López and Hughes, 2004] Pablo E. Martínez López and John Hughes. Principal type specialisation. In *Proceedings of Asian Symposium on Partial Evaluation and Semantic -Based Program Manipulation (ASIA-PEPM)*. ACM Press, September 2004.

[Martínez López, 2004] Pablo E. Martínez López. *Type Specialisation of Polymorphic Languages*. PhD thesis, University of Buenos Aires, 2004. (In preparation).

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

[Mogensen, 1996] Torben Æ. Mogensen. Evolution of partial evaluators: Removing inherited limits. In Danvy et al. [1996], pages 303–321.

[Mogensen, 1998] Torben Æ. Mogensen. Inherited limits. *ACM Computing Surveys*, 30(3), September 1998.

[Peyton Jones and Hughes (editors), 1999] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language, February 1999. URL: `http://www.haskell.org/onlinereport/`.

[Rèmy, 1989] Didier Rèmy. Typechecking records and variants in a natural extension of ML. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, January 1989. Austin, Texas.

[The University Court of the University of Glasgow, 2004] The University Court of the University of Glasgow. Haskell Compiler GHC, 2004. `http://www.haskell.org/ghc/`.

[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.

[Torvalds and Stallman, 2004] Linus Torvalds and Richard Stallman. Operating system GNU/Linux, 2004. `http://www.gnu.org` and `http://www.linux.org`.

[Wadler, 1995] Philip Wadler. Advanced functional programming, first international spring school on advanced functional programming techniques, båstad, sweden, may 24-30, 1995, tutorial text. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer, 1995.

[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.

Appendix A

Proofs

" Lo creía, pero la semilla de la duda estaba ahí, y permaneció, y de vez en cuando echaba una pequeña raíz. Esa semilla que crecía lo cambió todo. Hizo que Ender prestara más atención a lo que la gente quería decir, no a lo que decía. Le hizo más sabio."

VIII Rata
El juego de Ender
Orson Scott Card

In this appendix, we present the proofs of propositions, lemmas, and theorems used to prove the property of principality for the specialization of dynamic sum-types.

A.1 Proof of proposition 3.7 from section 3.3

PROPOSITION 3.7. *If $h : \Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : \sigma$, and $\Delta' \Vdash v : \Delta$, then $\Delta' \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e'[v/h] : \sigma$.*

Proof: By induction on the RT derivation.
Extending Proposition 6.11 from [Martínez López, 2004].

Case (RT-DCONSTR): We know that

$$\frac{\begin{array}{l}\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e' : \tau'_j \\ \quad (\Delta \Vdash v_k : C^k \in \tau'_e?\Delta_k, v_{m_k} : C^k \in \tau'_e?\text{HasC } \tau'_e \;\; C^k \;\; \tau'_k)_{k\in D\wedge k\neq j} \\ \quad \Delta \Vdash v_{m_j} : \text{HasC } \tau'_e \;\; C^j \;\; \tau'_j, v_{cs} : \text{ConstrsOf } \tau'_e\end{array}}{\Delta \mid \Gamma_{\text{R}} \vdash_{\text{RT}} C^j_{v_{m_j}} \; e' : \tau'_e}$$

By IH on the hypothesis of the rule, we know that

$$\Delta' \mid \Gamma_{\text{R}} \vdash_{\text{RT}} e'[v/h] : \tau'_j \tag{A.1}$$

Additionally, applying (Trans) to $\Delta' \Vdash \Delta$ and the entailments on the hypothesis of the rule, we obtain

$$\begin{array}{l}\Delta' \Vdash (v_k[v/h] : C^k \in \tau'_e?\Delta_k, v_{m_k}[v/h] : C^k \in \tau'_e?\text{HasC } \tau'_e \;\; C^k \;\; \tau'_k)_{k\in D\wedge k\neq j} \\ \Delta' \Vdash v_{m_j}[v/h] : \text{HasC } \tau'_e \;\; C^j \;\; \tau'_j, v_{cs}[v/h] : \text{ConstrsOf } \tau'_e\end{array} \tag{A.2}$$

Using A.1 and A.2 in order to apply (RT-DCONSTR), we have

$$\Delta' \mid \Gamma_{\text{R}} \vdash_{\text{RT}} C^j_{v_{m_j}[v/h]} \; e'[v/h] : \tau'_e$$

The result follows from $C^j_{v_{m_j}[v/h]} \; e'[v/h] = (C^j_{v_{m_j}} \; e')[v/h]$.

Case (RT-DCASE)**:** We know that

$$\frac{\begin{array}{l}\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \tau'_e \\ (h_k : \Delta_k \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \lambda x'_k.e'_k : \tau'_k \to \tau')_{k\in B} \\ (\Delta \Vdash v_{m_k} : C^k \in \tau'_e?\mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k, v_k : C^k \in \tau'_e?\Delta_k)_{k\in B} \\ \Delta \Vdash v_{cs} : \mathrm{ConstrsOf}\ \tau'_e\end{array}}{\begin{array}{l}\Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \mathbf{protocase}_v\ e'\ \mathbf{with}\ v_{cs}\ \mathbf{of} \\ \qquad (C^k_{v_{m_k}}\ x'_k\ \to\ e'_k[v_k/h_k])_{k\in B} \\ \quad : \tau'\end{array}}$$

Applying (Trans) to $\Delta' \Vdash \Delta$ and the entailments on the hypothesis of the rule, we obtain

$$\begin{array}{l}\Delta' \Vdash (v_{m_k}[v/h] : C^k \in \tau'_e?\mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k, v_k[v/h] : C^k \in \tau'_e?\Delta_k)_{k\in B} \\ \Delta' \Vdash v_{cs}[v/h] : \mathrm{ConstrsOf}\ \tau'_e\end{array} \tag{A.3}$$

By IH, we have

$$\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e'[v/h] : \tau'_e \tag{A.4}$$

Using A.3, A.4 and residual typing of lambda abstractions on the hypothesis of the rule in order to apply (RT-DCASE), we have

$$\begin{array}{c}\Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} \mathbf{protocase}_v\ e'[v/h]\ \mathbf{with}\ v_{cs}[v/h]\ \mathbf{of} : \tau' \\ C^k_{v_{m_k}[v/h]}\ x'_k\ \to\ e'_k[v_k[v/h]/h_k]\end{array}$$

The result follows from the fact that $e'_k[v_k[v/h]/h_k] = (e'_k[v_k/h_k])[v/h]$ since $h_k \neq h$ and

$$\begin{array}{ll}\mathbf{protocase}_v\ e'[v/h]\ \mathbf{with}\ v_{cs}[v/h]\ \mathbf{of} = & (\mathbf{protocase}_v\ e'\ \mathbf{with}\ v_{cs}\ \mathbf{of} \\ \quad C^k_{v_{m_k}[v/h]}\ x'_k\ \to\ e'_k[v_k[v/h]/h_k] & \qquad (C^k_{v_{m_k}}\ x'_k\ \to\ e'_k[v_k/h_k])_{k\in B}) \\ & [v/h]\end{array}$$

A.2 Proof of theorem 3.8 from section 3.3

THEOREM 3.8. *If* $h : \Delta \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} e' : \sigma$, *and* $C : (h : \Delta \mid \sigma) \geq (h' : \Delta' \mid \sigma')$, *then* $h' : \Delta' \mid \Gamma_{\mathrm{R}} \vdash_{\mathrm{RT}} C[e'] : \sigma'$.

Proof: The extension that we made to system RT does not modify the proof of Theorem 6.12 from [Martínez López, 2004]. That proof only applies Definition 2.20, (RT-QIN), (RT-QOUT), (RT-GEN), $\mathbf{let}_v$ and Proposition 3.7.

A.3 Proof of lemma 3.9 from section 3.4

LEMMA 3.9. *If* $h_1 : \Delta_1 \Vdash v_2 : \Delta_2$ *and* $h : \Delta \Vdash v_1 : C^k \in \tau'?\Delta_1, v_{cs} : \mathrm{ConstrsOf}\ \tau'$ *then* $h : \Delta \Vdash \mathbf{if}_v\ C^k \in v_{cs}\ \mathbf{then}\ v_2[v_1/h_1]\ \mathbf{else}\ \bullet : C^k \in \tau'?\Delta_2$

Proof: By hypothesis we have that

$$h_1 : \Delta_1 \Vdash v_2 : \Delta_2 \tag{A.5}$$

and

$$h : \Delta \Vdash v_{cs} : \text{ConstrsOf } \tau' \tag{A.6}$$

Now, applying (ENTL-GUARD) to A.5 and A.6, we obtain

$$h : \Delta, h_1 : C^k \in \tau'?\Delta_1 \Vdash \mathbf{if}_v\ C^k \in v_{cs}\ \mathbf{then}\ v_2\ \mathbf{else}\ \bullet : C^k \in \tau'?\Delta_2 \tag{A.7}$$

It is also trivially true that

$$h : \Delta \Vdash h : \Delta, v_1 : C^k \in \tau'?\Delta_1 \tag{A.8}$$

The result follows from applying (Trans) to A.7 and A.8, and since $EV(v_{cs}) \subseteq h$.

A.4 Proof of proposition 3.11 from section 3.4

PROPOSITION 3.11. *If* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$ *then* $S\,\Delta \vdash_{\text{SR}} \tau \hookrightarrow S\,\sigma$.

Proof: By induction on the SR derivation.
Extending Proposition 6.13 from [Martínez López, 2004].

Case (SR-DDT): We know that

$$\frac{\begin{array}{l}(\Delta_k \vdash_{\text{SR}} D(C^k) \hookrightarrow \tau'_k)_{k\in D}\\ (\Delta \Vdash C^k \in \tau'?\Delta_k, C^k \in \tau'?\text{HasC } \tau'\ \ C^k\ \ \tau'_k)_{k\in D}\\ \Delta \Vdash \text{ConstrsOf } \tau'\end{array}}{\Delta \vdash_{\text{SR}} D^D \hookrightarrow \tau'}$$

Applying IH and (Close) on the hypothesis, we obtain that

$$(S\,\Delta_k \vdash_{\text{SR}} D(C^k) \hookrightarrow S\,\tau'_k)_{k\in D} \tag{A.9}$$

and

$$\begin{array}{l}(S\,\Delta \Vdash S\,(C^k \in \tau'?\Delta_k), S\,(C^k \in \tau'?\text{HasC } \tau'\ \ C^k\ \ \tau'_k))_{k\in D}\\ S\,\Delta \Vdash S\,(\text{ConstrsOf } \tau')\end{array}$$

These last two entailments are equivalent to

$$\begin{array}{l}(S\,\Delta \Vdash C^k \in S\,\tau'?S\,\Delta_k, C^k \in S\,\tau'?\text{HasC } (S\,\tau')\ \ C^k\ \ (S\,\tau'_k))_{k\in D}\\ S\,\Delta \Vdash \text{ConstrsOf } (S\,\tau')\end{array} \tag{A.10}$$

Finally, applying (SR-DDT) with A.9 and A.10, we obtain

$$S\,\Delta \vdash_{\text{SR}} D^D \hookrightarrow S\,\tau'$$

A.5 Proof of proposition 3.12 from section 3.4

PROPOSITION 3.12. *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 Proposition 6.14 from [Martínez López, 2004].

Case (SR-DDT): We know that

$$\frac{\begin{array}{l}(\Delta_k \vdash_{\text{SR}} D(C^k) \hookrightarrow \tau_k')_{k\in D}\\ (\Delta \Vdash C^k \in \tau'?\Delta_k, C^k \in \tau'?\text{HasC } \tau' \;\; C^k \;\; \tau_k')_{k\in D}\\ \Delta \Vdash \text{ConstrsOf } \tau'\end{array}}{\Delta \vdash_{\text{SR}} D^D \hookrightarrow \tau'}$$

and we obtain

$$\begin{array}{l}(\Delta' \Vdash C^k \in \tau'?\Delta_k, C^k \in \tau'?\text{HasC } \tau' \;\; C^k \;\; \tau_k')_{k\in D}\\ \Delta' \Vdash \text{ConstrsOf } \tau'\end{array} \tag{A.11}$$

by applying (Trans) to $\Delta' \Vdash \Delta$ and

$$\begin{array}{l}(\Delta \Vdash C^k \in \tau'?\Delta_k, C^k \in \tau'?\text{HasC } \tau' \;\; C^k \;\; \tau_k')_{k\in D}\\ \Delta \Vdash \text{ConstrsOf } \tau'\end{array}$$

where Δ_k satisfies

$$(\Delta_k \vdash_{\text{SR}} D(C^k) \hookrightarrow \tau_k')_{k\in D} \tag{A.12}$$

The result follows from applying (SR-DDT) with A.11 and A.12 , proving

$$\Delta' \vdash_{\text{SR}} D^D \hookrightarrow \tau'$$

A.6 Proof of theorem 3.13 from section 3.4

THEOREM 3.13. *If* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$ *and* $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ *then* $\Delta' \vdash_{\text{SR}} \tau \hookrightarrow \sigma'$*.*

Proof: The extension that we have made to system SR does not modify the proof of Theorem 6.15 from [Martínez López, 2004]. This proof uses Definition 2.20, (SR-INST), (SR-QIN), (SR-QOUT), (SR-GEN) and Proposition 3.12.

A.7 Proof of theorem 3.14 from section 3.4

THEOREM 3.14. *If* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma$*, and for all* $x : \tau_x \hookrightarrow x' : \tau_x' \in \Gamma$*,* $\Delta \vdash_{\text{SR}} \tau_x \hookrightarrow \tau_x'$*, then* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$*.*

Proof: By induction on the P derivation.
Extending Theorem 6.19 from [Martínez López, 2004].

Case (DCONSTR): We know that

$$\frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} e : D(C^j) \hookrightarrow e' : \tau'_j \\ (\Delta_k \vdash_{\mathrm{SR}} D(C^k) \hookrightarrow \tau'_k)_{k \in D \wedge k \neq j} \\ (\Delta \Vdash v_k : C^k \in \tau'_e ? \Delta_k, v_{m_k} : C^k \in \tau'_e ? \mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k)_{k \in D \wedge k \neq j} \\ \Delta \Vdash v_{m_j} : \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j, v_{cs} : \mathrm{ConstrsOf}\ \tau'_e\end{array}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} C^D_j\ \ e : D^D \hookrightarrow C^j_{v_{m_j}}\ \ e' : \tau'_e}$$

By IH we know that

$$\Delta \vdash_{\mathrm{SR}} D(C^j) \hookrightarrow \tau'_j$$

and by hypothesis we also know that

$$(\Delta_k \vdash_{\mathrm{SR}} D(C^k) \hookrightarrow \tau'_k)_{k \in D \wedge k \neq j}$$

So, defining $\Delta_j = \Delta$, we have that

$$(\Delta_k \vdash_{\mathrm{SR}} D(C^k) \hookrightarrow \tau'_k)_{k \in D} \tag{A.13}$$

Now, we have to prove that

$$\begin{array}{l}(\Delta \Vdash C^k \in \tau'_e ? \Delta_k, C^k \in \tau'_e ? \mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k)_{k \in D} \\ \Delta \Vdash \mathrm{ConstrsOf}\ \tau'_e\end{array} \tag{A.14}$$

By (Univ), it is enough to prove

$$(\Delta \Vdash C^k \in \tau'_e ? \Delta_k, C^k \in \tau'_e ? \mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k)_{k \in D \wedge k \neq j} \tag{A.15}$$

$$\Delta \Vdash C^j \in \tau'_e ? \Delta, C^j \in \tau'_e ? \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \tag{A.16}$$

$$\Delta \Vdash \mathrm{ConstrsOf}\ \tau'_e \tag{A.17}$$

Item **A.15)** holds trivially by hypothesis.

For item **A.16)** we know that $\Delta \Vdash \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j$ and so

$$\Delta \Vdash \Delta, \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \tag{A.18}$$

$$\Delta, \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \Vdash \Delta \tag{A.19}$$

$$\Delta, \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \Vdash \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \tag{A.20}$$

are trivially true. Applying (HASC-GUARD) to A.19 and A.20, we obtain

$$\Delta, \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \Vdash C^j \in \tau'_e ? \Delta \tag{A.21}$$

$$\Delta, \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \Vdash C^j \in \tau'_e ? \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \tag{A.22}$$

Then, applying (Univ) to A.21 and A.22 , we have

$$\Delta, \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \Vdash C^j \in \tau'_e ? \Delta, C^j \in \tau'_e ? \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \tag{A.23}$$

Finally, we apply (Trans) to A.18 and A.23.

Item **A.17)** holds trivially by hypothesis.

The result follows from applying (SR-DDT) to A.13 and A.14.

Case (DCASE): It holds trivially by hypothesis.

A.8 Proof of theorem 3.15 from section 3.4

THEOREM 3.15. *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$, *then* $\Delta \mid \Gamma_{(\mathrm{RT})} \vdash_{\mathrm{RT}} e' : \sigma$.

Proof: By induction on the P derivation.
Extending Theorem 6.20 from [Martínez López, 2004]. The proof is trivial by applying IH when necessary.

A.9 Proof of proposition 3.16 from section 3.4

PROPOSITION 3.16. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \tau'$ *and* $h' : \Delta' \Vdash v : \Delta$, *then* $h' : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'[h/v] : \tau'$

Proof: By induction on the P derivation.
Extending Proposition 6.21 from [Martínez López, 2004].

Case (DCONSTR): We know that

$$\frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} e : D(C^j) \hookrightarrow e' : \tau'_j \\ (\Delta_k \vdash_{\mathrm{SR}} D(C^k) \hookrightarrow \tau'_k)_{k \in D \wedge k \neq j} \\ (\Delta \Vdash v_k : C^k \in \tau'_e ? \Delta_k, v_{m_k} : C^k \in \tau'_e ? \mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k)_{k \in D \wedge k \neq j} \\ \Delta \Vdash v_{m_j} : \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j, v_{cs} : \mathrm{ConstrsOf}\ \tau'_e\end{array}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} C_j^D\ \ e : D^D \hookrightarrow C^j_{v_{m_j}}\ e' : \tau'_e}$$

By IH, we have

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} e : D(C^j) \hookrightarrow e'[v/h] : \tau'_e \tag{A.24}$$

Applying (Trans) to $\Delta' \Vdash \Delta$ and the entailments on the hypothesis, we have that

$$\begin{array}{l}(\Delta' \Vdash v_k[v/h] : C^k \in \tau'_e ? \Delta_k, v_{m_k}[v/h] : C^k \in \tau'_e ? \mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k)_{k \in D \wedge k \neq j} \\ \Delta' \Vdash v_{m_j}[v/h] : \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \\ \Delta' \Vdash v_{cs}[v/h] : \mathrm{ConstrsOf}\ \tau'_e\end{array} \tag{A.25}$$

Applying (DCONSTR) to A.24, A.25 and the SR derivations in the hypothesis, we obtain

$$\Delta' \mid \Gamma \vdash_{\mathrm{P}} C_j^D\ \ e : D^D \hookrightarrow C^j_{v_{m_j}[v/h]}\ e'[v/h] : \tau'_e \tag{A.26}$$

The result follows since $C^j_{v_{m_j}[v/h]}\ e'[v/h] = (C^j_{v_{m_j}}\ \ e')[v/h]$.

Case (DCASE)**:** We know that

$$\frac{\begin{array}{l}\Delta \vdash_{\text{SR}} \tau \ \hookrightarrow\ \tau' \\ \Delta \mid \Gamma \vdash_{\text{P}} e : D^{D} \ \hookrightarrow\ e' : \tau'_e \\ (h_k : \Delta_k \mid \Gamma \vdash_{\text{P}} \lambda^{D} x_k.e_k : D(C^k) \rightarrow^{D} \tau \ \hookrightarrow\ \lambda x'_k.e'_k : \tau'_k \rightarrow \tau')_{k \in B} \\ \ (\Delta \Vdash v_{m_k} : C^k \in \tau'_e ?\text{HasC } \tau'_e \ \ C^k \ \ \tau'_k, v_k : C^k \in \tau'_e ?\Delta_k)_{k\in B} \\ \ \Delta \Vdash v_{cs} : \text{ConstrsOf } \tau'_e\end{array}}{\begin{array}{ll}\Delta \mid \Gamma \vdash_{\text{P}} & \textbf{case } e \textbf{ of} \\ & \quad (C^{D}_k \ \ x_k \ \rightarrow\ e_k)_{k\in B} \\ & : \tau \\ & \hookrightarrow \\ & \textbf{protocase}_v\ e' \textbf{ with } v_{cs} \textbf{ of} \\ & \quad (C^k_{v_{m_k}} \ x'_k \ \rightarrow\ e'_k[v_k/h_k])_{k\in B} \\ & : \tau'\end{array}}$$

By Proposition 3.12, IH and applying (Trans) to $\Delta' \Vdash \Delta$ and the entailments on the hypothesis, we have that

$$\begin{array}{l}\Delta' \vdash_{\text{SR}} \tau \ \hookrightarrow\ \tau' \\ \Delta' \mid \Gamma \vdash_{\text{P}} e : D^{D} \ \hookrightarrow\ e'[v/h] : \tau'_e \\ (\Delta' \Vdash v_{m_k}[v/h] : C^k \in \tau'_e ?\text{HasC } \tau'_e \ \ C^k \ \ \tau'_k, v_k[v/h] : C^k \in \tau'_e ?\Delta_k)_{k\in B} \\ \Delta' \Vdash v_{cs}[v/h] : \text{ConstrsOf } \tau'_e\end{array}$$

Now, applying (DCASE) to these last facts and the third hypothesis, we obtain

$$\begin{array}{l}\textbf{protocase}_v\ e'[v/h] \textbf{ with } v_{cs}[v/h] \textbf{ of} \\ \quad (C^k_{v_{m_k}[v/h]} \ x'_k \ \rightarrow\ e'_k[v_k[v/h]/h_k])_{k\in B}\end{array}$$

The result follows since $e'_k[v_k[v/h]/h_k] = (e'_k[v_k/h_k])[v/h]$ and $h_k \neq h$.

A.10 Proof of proposition 3.17 from section 3.4

PROPOSITION 3.17. *If* $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \ \hookrightarrow\ e' : \sigma$ *then* $S\,\Delta \mid S\,\Gamma \vdash_{\text{P}} e : \tau \ \hookrightarrow\ e' : S\,\sigma$*.*

Proof: By induction on the P derivation.
Extending Proposition 6.22 from [Martínez López, 2004].

Case (DCONSTR)**:** It holds by applying Proposition 3.11 on all judgements from system SR in the hypothesis of the rule, IH and (Close). Finally, we must apply (DCONSTR).

Case (DCASE)**:** It holds by applying Proposition 3.11, IH on all judgements from system P in the hypothesis of the rule and (Close). Finally, we must apply (DCASE).

A.11 Proof of lemma 3.18 from section 3.4

Lemma 3.18. *If* $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ *then* $EV(e') \subseteq h$

Proof: By induction on the P derivation.
Extending Lemma 6.23 from [Martínez López, 2004].

Case (dconstr): We know that

$$\frac{\begin{array}{l}\Delta \mid \Gamma \vdash_{\mathrm{P}} e : D(C^j) \hookrightarrow e' : \tau'_j \\ (\Delta_k \vdash_{\mathrm{SR}} D(C^k) \hookrightarrow \tau'_k)_{k \in D \wedge k \neq j} \\ (\Delta \Vdash v_k : C^k \in \tau'_e?\Delta_k, v_{m_k} : C^k \in \tau'_e?\mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k)_{k \in D \wedge k \neq j} \\ \Delta \Vdash v_{m_j} : \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j, v_{cs} : \mathrm{ConstrsOf}\ \tau'_e\end{array}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} C_j^{D}\ e : D^{D} \hookrightarrow C^j_{v_{m_j}}\ e' : \tau'_e}$$

Let us take h as evidence of Δ. Applying IH we have that $EV(e') \subseteq h$ and we also know, by entailment, that $EV(v_{m_j}) \subseteq h$, so the result follows since $EV(C^j_{v_{m_j}}\ e') = EV(v_{m_j}) \bigcup EV(e')$.

Case (dcase): We know that

$$\frac{\begin{array}{l}\Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow \tau' \\ \Delta \mid \Gamma \vdash_{\mathrm{P}} e : D^{D} \hookrightarrow e' : \tau'_e \\ (h_k : \Delta_k \mid \Gamma \vdash_{\mathrm{P}} \lambda^{D} x_k.e_k : D(C^k) \to^{D} \tau \hookrightarrow \lambda x'_k.e'_k : \tau'_k \to \tau')_{k \in B} \\ (\Delta \Vdash v_{m_k} : C^k \in \tau'_e?\mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k, v_k : C^k \in \tau'_e?\Delta_k)_{k \in B} \\ \Delta \Vdash v_{cs} : \mathrm{ConstrsOf}\ \tau'_e\end{array}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \begin{array}{l}\mathbf{case}\ e\ \mathbf{of} \\ \quad (C_k^{D}\ x_k \ \to\ e_k)_{k \in B} \\ : \tau \\ \hookrightarrow \\ \mathbf{protocase}_v\ e'\ \mathbf{with}\ v_{cs}\ \mathbf{of} \\ \quad (C^k_{v_{m_k}}\ x'_k \ \to\ e'_k[v_k/h_k])_{k \in B} \\ : \tau'\end{array}}$$

Let us take h as evidence of Δ. The evidence variables of the residual term produced are

$$EV(e') \bigcup EV(v_{cs}) \bigcup (EV(v_{m_k}))_{k \in B} \bigcup (EV(e'_k[v_k/h_k]))_{k \in B} \qquad \text{(A.27)}$$

By IH on all judgements from system P in the hypothesis of the rule we obtain that $EV(e') \subseteq h$ and $(EV(e'_k) \subseteq h_k)_{k \in B}$. We have that $(EV(e'_k[v_k/h_k]) = EV(v_k))_{k \in B}$, by substitution. Because of entailments of predicates in the hypothesis of the rule, we have that $(EV(v_k) \subseteq h)_{k \in B}$, $EV(v_{cs}) \subseteq h$ and $(EV(v_{m_k}) \subseteq h)_{k \in B}$. Finally, the result follows from A.27 and the described remarks.

A.12 Proof of lemma 3.19 from section 3.4

LEMMA 3.19. *If* $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ *then there exist* β_j, Δ_σ, *and* τ' *such that* $\sigma = \forall \beta_j.\Delta_\sigma \Rightarrow \tau'$.

Proof: By induction on the P derivation.
Extending Lemma 6.24 from [Martínez López, 2004].

Case (DCONSTR): It holds trivially by just taking $\sigma = \forall\emptyset.\emptyset \Rightarrow \tau'_e$.

Case (DCASE): It holds trivially by just taking $\sigma = \forall\emptyset.\emptyset \Rightarrow \tau'$.

A.13 Proof of proposition 4.2 from section 4.1

PROPOSITION 4.2. *If* $\Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \Delta \Rightarrow \tau'$, *then* $\Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \mathrm{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau')$, *and both derivations only differ in the application of rule* (GEN).

Proof: The extension that we have made to the system P does not modify the proof of Proposition 7.2 from [Martínez López, 2004]. This proof uses repeated applications of (GEN).

A.14 Proof of proposition 4.3 from section 4.1

PROPOSITION 4.3. *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 Proposition 7.4 from [Martínez López, 2004]. The proof follows the same structure that Proposition 3.17.

A.15 Proof of proposition 4.4 from section 4.1

PROPOSITION 4.4. *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 S derivation.
Extending Proposition 7.5 from [Martínez López, 2004]. The proof follows the same structure that Proposition 3.16 .

A.16 Proof of theorem 4.5 from section 4.1

THEOREM 4.5. *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 Proposition 7.6 from [Martínez López, 2004]. The proof is trivial using IH.

A.17 Proof of theorem 4.6 from section 4.1

THEOREM 4.6. *If $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \;\hookrightarrow\; e' : \sigma$, then there exist $h'_s, \Delta'_s, e'_s, \tau'_s$, and C'_s such that*

a) $h'_s : \Delta'_s \mid \Gamma \vdash_{\mathrm{S}} e : \tau \;\hookrightarrow\; e'_s : \tau'_s$

b) $C'_s : \mathrm{Gen}_\Gamma(\Delta'_s \Rightarrow \tau'_s) \geq (h : \Delta \mid \sigma)$

c) $C'_s[\Lambda h'_s.e'_s] = e'$

Proof: By induction on the P derivation.
Extending Proposition 7.7 from [Martínez López, 2004].

Case (DCONSTR): We know that

$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash_{\mathrm{P}} e : D(C^j) \;\hookrightarrow\; e' : \tau'_j \\ (\Delta_k \vdash_{\mathrm{SR}} D(C^k) \;\hookrightarrow\; \tau'_k)_{k \in D \wedge k \neq j} \\ (\Delta \Vdash v_k : C^k \in \tau'_e ? \Delta_k, v_{m_k} : C^k \in \tau'_e ? \mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k)_{k \in D \wedge k \neq j} \\ \Delta \Vdash v_{m_j} : \mathrm{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j, v_{cs} : \mathrm{ConstrsOf}\ \tau'_e\end{array}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} C^D_j\ \ e : D^D \;\hookrightarrow\; C^j_{v_{m_j}}\ e' : \tau'_e}$$

By IH, we have that there exist h^s_e, Δ^s_e, e'_s, τ'_{j^s} and C^s_e such that

$$\Delta^s_e \mid \Gamma \vdash_{\mathrm{S}} e : D(C^j) \;\hookrightarrow\; e'_s : \tau'_{j^s} \tag{A.28}$$
$$C^s_e : \mathrm{Gen}_\Gamma(\Delta^s_e \Rightarrow \tau'_{j^s}) \geq (\Delta \mid \tau'_j) \tag{A.29}$$
$$C^s_e[\Lambda h^s_e.e'_s] = e' \tag{A.30}$$

By Definition 2.20 and A.29, we also know that there exist a substitution S^s_e and evidence v^s_e such that

$$\tau'_j = S^s_e \tau'_{j^s} \tag{A.31}$$
$$\Delta \Vdash v^s_e : S^s_e \Delta^s_e \tag{A.32}$$
$$C^s_e = \mathbf{let}_v\ x = []\ \mathbf{in}\ x((v^s_e)) \tag{A.33}$$

By hypothesis, we also have that

$$(\Delta_k \vdash_{\mathrm{SR}} D(C^k) \;\hookrightarrow\; \tau'_k)_{k \in D \wedge k \neq j} \tag{A.34}$$
$$(\Delta \Vdash v_k : C^k \in \tau'_e ? \Delta_k, v_{m_k} : C^k \in \tau'_e ? \mathrm{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k)_{k \in D \wedge k \neq j} \tag{A.35}$$
$$\Delta \Vdash v_{cs} : \mathrm{ConstrsOf}\ \tau'_e \tag{A.36}$$

Let us take $h'_s = (h, h_{v_{m_j}}, h^s_e)$, $\Delta'_s = (\Delta, \mathrm{HasC}\ \tau'_e\ \ C^j\ \ (S^s_e \tau'_{j^s}), S^s_e \Delta^s_e)$, $\tau'_s = \tau'_e$ and $C'_s = \mathbf{let}_v\ x = []\ \mathbf{in}\ x((h, h_{v_{m_j}}, v^s_e))$, so we need to prove that

a) $\Delta'_s \mid \Gamma \vdash_{\mathrm{S}} C^D_j\ \ e : D^D \;\hookrightarrow\; C^j_{h_{v_{m_j}}}\ \ e'_s : \tau'_e$

b) $C'_s : \text{Gen}_\Gamma(\Delta'_s \Rightarrow \tau'_e) \geq (\Delta \mid \tau'_e)$

c) $C'_s[\Lambda h'_s.C^j_{h_{v_{m_j}}}\ e'_s] = C^j_{v_{m_j}}\ e'$

For item **a)** we apply Proposition 4.3 with S^s_e and A.28, obtaining

$$S^s_e\Delta^s_e \mid S^s_e\Gamma \vdash_{\text{S}} e : D(C^j) \ \hookrightarrow\ e'_s : S^s_e\tau'_{j^s}$$

By A.31 and because we do not generalize variables appearing in Γ (and so, they do not appear in the domain of S^s_e), this last judgement is equivalent to

$$S^s_e\Delta^s_e \mid \Gamma \vdash_{\text{S}} e : D(C^j) \ \hookrightarrow\ e'_s : \tau'_j \tag{A.37}$$

Then, applying Proposition 4.4 to A.37 and $\Delta'_s \Vdash S^s_e\Delta^s_e$, we have that

$$\Delta'_s \mid \Gamma \vdash_{\text{S}} e : D(C^j) \ \hookrightarrow\ e'_s : \tau'_j \tag{A.38}$$

By definition of Δ'_s and A.31, it is very easy to verify that

$$\Delta'_s \Vdash h_{v_{m_j}} : \text{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j \tag{A.39}$$

Applying (Trans) to $\Delta'_s \Vdash \Delta$ and A.35 and A.36, we have that

$$\begin{aligned}&(\Delta'_s \Vdash v_k : C^k \in \tau'_e?\Delta_k, \\ &\qquad v_{m_k} : C^k \in \tau'_e?\text{HasC}\ \tau'_e\ \ C^k\ \ \tau'_k)_{k\in D \wedge k\neq j}\end{aligned} \tag{A.40}$$

$$\Delta'_s \Vdash v_{cs} : \text{ConstrsOf}\ \tau'_e \tag{A.41}$$

Finally, applying (S-DCONSTR) with A.38, A.34, A.40, A.39 and A.41, we obtain the desired property.

For item **b)** we take the substitution Id and evidence $v = (h, v_{m_j}, v^s_e)$. By Definition 2.20, we need to prove that

$$\tau'_e = \text{Id}\tau'_e \tag{A.42}$$

$$\Delta \Vdash v : \text{Id}\Delta'_s \tag{A.43}$$

Item **A.42)** holds trivially by definition of Id.

Item **A.43)** holds by applying (Univ) several times and then definition of Δ'_s, A.32 and the hypothesis $\Delta \Vdash \text{HasC}\ \tau'_e\ \ C^j\ \ \tau'_j$.

For item **c)**, we have that $(\Lambda h'_s.C^j_{h_{v_{m_j}}}\ e'_s) = (\Lambda h, h_{v_{m_j}}, h^s_e.C^j_{h_{v_{m_j}}}\ e'_s)$ by definition of h'_s and Δ'_s. Using Definition 2.20 and evidence v, we obtain that

$$C'_s[\Lambda h, h_{v_{m_j}}, h^s_e.C^j_{h_{v_{m_j}}}\ e'_s] = (\Lambda h, h_{v_{m_j}}, h^s_e.C^j_{h_{v_{m_j}}}\ e'_s)((h, v_{m_j}, v^s_e))$$

On the other hand, we have that $EV(e'_s) \subseteq h^s_e$ by Lemma 3.18 and $h^s_e \neq h \neq h_{v_{m_j}}$, so we can rewrite the last equation as

$$C'_s[\Lambda h, h_{v_{m_j}}, h^s_e.C^j_{h_{v_{m_j}}}\ e'_s] = C^j_{v_{m_j}}\ e'_s[v^s_e/h^s_e]$$

The result follows since $e'_s[v^s_e/h^s_e] = e'$ by A.30.

Case (DCASE): We know that

$$\frac{\begin{array}{l}\Delta \vdash_{\text{SR}} \tau \hookrightarrow \tau' \\ \Delta \mid \Gamma \vdash_{\text{P}} e : D^{D} \hookrightarrow e' : \tau'_e \\ (h_k : \Delta_k \mid \Gamma \vdash_{\text{P}} \lambda^{D} x_k.e_k : D(C^k) \rightarrow^{D} \tau \hookrightarrow \lambda x'_k.e'_k : \tau'_k \rightarrow \tau')_{k \in B} \\ (\Delta \Vdash v_{m_k} : C^k \in \tau'_e?\text{HasC } \tau'_e \;\; C^k \;\; \tau'_k, v_k : C^k \in \tau'_e?\Delta_k)_{k \in B} \\ \Delta \Vdash v_{cs} : \text{ConstrsOf } \tau'_e\end{array}}{\begin{array}{ll}\Delta \mid \Gamma \vdash_{\text{P}} & \mathbf{case}\ e\ \mathbf{of} \\ & \quad (C^{D}_k\ x_k \rightarrow e_k)_{k \in B} \\ & : \tau \\ & \hookrightarrow \\ & \mathbf{protocase}_v\ e'\ \mathbf{with}\ v_{cs}\ \mathbf{of} \\ & \quad (C^{k}_{v_{m_k}}\ x'_k \rightarrow e'_k[v_k/h_k])_{k \in B} \\ & : \tau'\end{array}}$$

By IH on the second hypothesis of the rule, we have that there exist $h^s_{e'_s}$, $\Delta^s_{e'_s}$, $\tau'_{e'_s}$ and $C_{e'_s}$ such that

$$h^s_{e'_s} : \Delta^s_{e'_s} \mid \Gamma \vdash_{\text{S}} e : D^{D} \hookrightarrow e'_s : \tau'_{e'_s} \tag{A.44}$$
$$C_{e'_s} : \text{Gen}_{\Gamma}(\Delta^s_{e'_s} \Rightarrow \tau'_{e'_s}) \geq (\Delta \mid \tau'_e) \tag{A.45}$$
$$C_{e'_s}[\Lambda h^s_{e'_s}.e'_s] = e' \tag{A.46}$$

On the other hand, we apply IH on judgements related to abstractions (the third hypothesis of the rule), and obtain that for each k, there exist h^s_k, Δ^s_k, τ'_{k^s}, τ'_s and C_k such that

$$h^s_k : \Delta^s_k \mid \Gamma \vdash_{\text{S}} \lambda^{D} x_k.e_k : D(C^k) \rightarrow^{D} \tau \hookrightarrow \lambda x'_{k^s}.e'_{k^s} : \tau'_{k^s} \rightarrow \tau'_s \tag{A.47}$$
$$C_k : \text{Gen}_{\Gamma}(\Delta^s_k \Rightarrow \tau'_{k^s} \rightarrow \tau'_s) \geq (\Delta \mid \tau'_k \rightarrow \tau') \tag{A.48}$$
$$C_k[\Lambda h^s_k.\lambda x'_{k^s}.e'_{k^s}] = \lambda x'_k.e'_k \tag{A.49}$$

By Definition 2.20 applied to A.45, we know that there exist a substitution $S_{e'_s}$ and evidence $v_{e'_s}$ such that

$$\tau'_e = S_{e'_s}\tau'_{e'_s} \tag{A.50}$$
$$h : \Delta \Vdash v_{e'_s} : \Delta^s_{e'_s} \tag{A.51}$$
$$C_{e'_s} = \mathbf{let}_v\ x = []\ \mathbf{in}\ x(\!(v_{e'_s})\!)$$

The conversion C_k from A.48 implies the existence of a substitution S_k and evidence v'_k such that

$$\tau'_k \rightarrow \tau' = S_k(\tau'_{k^s} \rightarrow \tau'_s) \tag{A.52}$$
$$h_k : \Delta_k \Vdash v'_k : S_k\Delta^s_k \tag{A.53}$$
$$C_k = \mathbf{let}_v\ x = []\ \mathbf{in}\ x(\!(v'_k)\!)$$

Let us take $h'_s = (h, h^s_{e'_s}, (h^s_{v_{m_k}}, h^s_{v_k})_{k\in B}, h^s_{v_{cs}})$ and

$$\begin{aligned}\Delta'_s = (&\Delta, \Delta^s_{e'_s},\\ &(C^k \in S_{e'_s}\tau'_{e'_s}?\text{HasC } (S_{e'_s}\tau'_{e'_s})\ \ C^k\ \ (S_k\tau'_{k^s}), C^k \in S_{e'_s}\tau'_{e'_s}?S_k\Delta^s_k)_{k\in B},\\ &\text{ConstrsOf } S_{e'_s}\tau'_{e'_s})\end{aligned} \tag{A.54}$$

So, we need to prove that

a) $\Delta'_s \mid \Gamma \vdash_{\mathrm{S}} \textbf{case } e \textbf{ of } (C^D_k\ \ x_k\ \rightarrow\ e_k)_{k\in B} : \tau \hookrightarrow \textbf{protocase}_v\ e'_s \textbf{ with } h^s_{v_{cs}} \textbf{ of } (C^k_{h^s_{v_{m_k}}}\ \ x'_{k^s}\ \rightarrow\ e'_{k^s}[h^s_{v_k}/h^s_k])_{k\in B} : \tau'$

b) $C'_s : \text{Gen}_\Gamma(\Delta'_s \Rightarrow \tau') \geq (\Delta \mid \tau')$

c) $C'_s[\Lambda h'_s.\textbf{protocase}_v\ e'_s \textbf{ with } h^s_{v_{cs}} \textbf{ of } (C^k_{h^s_{v_{m_k}}}\ \ x'_{k^s}\ \rightarrow\ e'_{k^s}[h^s_{v_k}/h^s_k])_{k\in B}] = \textbf{protocase}_v\ e' \textbf{ with } v_{cs} \textbf{ of } (C^k_{v_{m_k}}\ x'_k\ \rightarrow\ e'_k[v_k/h_k])_{k\in B}$

Item **a)** By definition of Δ'_s (A.54) and application of Proposition 3.12 to the first hypothesis of the rule and (DCASE) with $\Delta'_s \Vdash \Delta$, we have that

$$\Delta'_s \vdash_{\mathrm{SR}} \tau\ \hookrightarrow\ \tau' \tag{A.55}$$

Additionally, by applying Proposition 4.3 to A.44 with $S_{e'_s}$, A.50 , Proposition 4.4 with $\Delta'_s \Vdash S_{e'_s}\Delta^s_{e'_s}$ and the fact that $\text{dom}(S_{e'_s})\bigcap FTV(\Gamma) = \emptyset$, we know that

$$\Delta'_s \mid \Gamma \vdash_{\mathrm{S}} e : D^D\ \hookrightarrow\ e'_s : \tau'_e \tag{A.56}$$

Beside this, by applying Proposition 4.3 to A.47 with S_k, A.52 and the fact that $\text{dom}(S_k)\bigcap FTV(\Gamma) = \emptyset$, we have that

$$\Delta'_s \mid \Gamma \vdash_{\mathrm{S}} \lambda^D x_k.e_k : D(C^k) \rightarrow^D \tau\ \hookrightarrow\ \lambda x'_{k^s}.e'_{k^s} : \tau'_k \rightarrow \tau' \tag{A.57}$$

Besides, by definition of Δ'_s, A.50 and A.52, it is very easy to verify that

$$\begin{aligned}&(\Delta'_s \Vdash h^s_{v_{m_k}} : C^k \in \tau'_e?\text{HasC } \tau'_e\ \ C^k\ \ \tau'_k, h^s_{v_k} : C^k \in \tau'_e?S_k\Delta^s_k)_{k\in B}\\ &\Delta'_s \Vdash h^s_{v_{cs}} : \text{ConstrsOf } \tau'_e\end{aligned} \tag{A.58}$$

The result follows from applying (S-DCASE) with A.55, A.56, A.57 and A.58.

Item **b)** Let us take the substitution Id and evidence

$$v = (h, v_{e'_s}, (v_{m_k}, \textbf{if}_v\ C^k \in v_{cs} \textbf{ then } v'_k[v_k/h_k] \textbf{ else } \bullet)_{k\in B}, v_{cs})$$

We have to prove that

$$\tau' = \text{Id}\tau' \tag{A.59}$$

$$\Delta \Vdash v : \Delta'_s \tag{A.60}$$

Item **A.59)** holds trivially by definition of Id.

For item **A.60)**, it is enough to prove that (by virtue of A.54)

$$h : \Delta \Vdash \Delta \tag{A.61}$$

$$h : \Delta \Vdash \Delta^s_{e'_s} \tag{A.62}$$

$$(h : \Delta \Vdash C^k \in S_{e'_s}\tau'_{e'_s}?\text{HasC }(S_{e'_s}\tau'_{e'_s})\ \ C^k\ \ (S_k\tau'_{k^s}))_{k\in B} \tag{A.63}$$

$$(h : \Delta \Vdash C^k \in S_{e'_s}\tau'_{e'_s}?S_k\Delta^s_k)_{k\in B} \tag{A.64}$$

$$h : \Delta \Vdash \text{ConstrsOf }(S_{e'_s}\tau'_{e'_s}) \tag{A.65}$$

Item **A.61)** holds trivially with evidence h.

Item **A.62)** holds by A.51 with evidence $v_{e'_s}$.

Item **A.63)** holds by A.50, A.52 and hypothesis with evidence v_{m_k}.

For item **A.64)**, it is very easy to verify by hypothesis that

$$\Delta \Vdash v_k : C^k \in \tau'_e?\Delta_k, v_{cs} : \text{ConstrsOf } \tau'_e$$

By A.50 and application of Lemma 3.9 to this last entailment and A.53, we have that

$$\Delta \Vdash \mathbf{if}_v\ C^k \in v_{cs}\ \mathbf{then}\ v'_k[v_k/h_k]\ \mathbf{else}\ \bullet : C^k \in \tau'_e?S_k\Delta^s_k$$

Item **A.65)** holds by A.50 and hypothesis with evidence v_{cs}.

Finally, by Definition 2.20 we know that

$$\begin{aligned}C'_s = {} & \mathbf{let}_v\ x = []\\ & \mathbf{in}\ x(\!(h, v_{e'_s}, (v_{m_k}, \mathbf{if}_v\ C^k \in v_{cs}\ \mathbf{then}\ v'_k[v_k/h_k]\ \mathbf{else}\ \bullet)_{k\in B}, v_{cs})\!)\end{aligned}$$

For item **c)**, by definition of C'_s and Δ'_s, we have that

$$\begin{aligned}&(\mathbf{protocase}_v\ e'_s\ \mathbf{with}\ h^s_{v_{cs}}\ \mathbf{of}\\ &\quad (C^k_{h^s_{v_{m_k}}}\ x'_{k^s}\ \rightarrow\ e'_{k^s}[h^s_{v_k}/h^s_k])_{k\in B})\\ &[h, v_{e'_s}, (v_{m_k}, \mathbf{if}_v\ C^k \in v_{cs}\ \mathbf{then}\ v'_k[v_k/h_k]\ \mathbf{else}\ \bullet)_{k\in B}, v_{cs}/\\ &h, h_{e'_s}, (h^s_{v_{m_k}}, h^s_{v_k})_{k\in B}, h^s_{v_{cs}}]\end{aligned} \tag{A.66}$$

Since $EV(e'_s) \subseteq h^s_{e'_s}, EV(v_{cs}) \subseteq h$, $EV(v'_k) \subseteq h_k$, $EV(v'_k[v_k/h_k]) \subseteq h$ in addition to $EV(e'_{k^s}[h^s_{v_k}/h^s_k]) \subseteq h^s_{v_k}$ and $h^s_{e'_s} \neq h \neq h_k \neq h^s_k \neq h^s_{v_k} \neq h^s_{v_{cs}}$, so A.66 can be rewritten as

$$\begin{aligned}&\mathbf{protocase}_v\ e'_s[v_{e'_s}/h^s_{e'_s}]\ \mathbf{with}\ h^s_{v_{cs}}[v_{cs}/h^s_{v_{cs}}]\ \mathbf{of}\\ &\quad (C^k_{h^s_{v_{m_k}}[v_{m_k}/h^s_{v_{m_k}}]}\ x'_{k^s}\ \rightarrow\ e'_{k^s}[\mathbf{if}_v\ C^k \in v_{cs}\ \mathbf{then}\ v'_k[v_k/h_k]\ \mathbf{else}\ \bullet\ /h^s_k])_{k\in B}\end{aligned} \tag{A.67}$$

By A.46, A.51 and substitutions, this last term is equivalent to

$$\begin{aligned}&\mathbf{protocase}_v\ e'\ \mathbf{with}\ v_{cs}\ \mathbf{of}\\ &\quad (C^k_{v_{m_k}}\ x'_{k^s}\ \rightarrow\ e'_{k^s}[\mathbf{if}_v\ C^k \in v_{cs}\ \mathbf{then}\ v'_k\ \mathbf{else}\ \bullet\ /h^s_k][v_k/h_k])_{k\in B}\end{aligned}$$

So, we need to prove that

$$\begin{aligned}&\mathbf{protocase}_v\ e'\ \mathbf{with}\ v_{cs}\ \mathbf{of}\\ &\quad (C^k_{v_{m_k}}\ x'_{k^s}\ \rightarrow\ e'_{k^s}[\mathbf{if}_v\ C^k \in v_{cs}\ \mathbf{then}\ v'_k\ \mathbf{else}\ \bullet\ /h^s_k][v_k/h_k])_{k\in B}\\ &= \mathbf{protocase}_v\ e'\ \mathbf{with}\ v_{cs}\ \mathbf{of}\\ &\qquad (C^k_{v_{m_k}}\ x'_k\ \rightarrow\ e'_k[v_k/h_k])_{k\in B}\end{aligned}$$

We have to consider two cases accordingly to equality of $\mathbf{protocase}_v$.

Case $v_{cs} = \{C^k\}_{k\in I}$) By Definition 3.6, it is enough to prove that

$$(\lambda x'_k.e'_k[v_k/h_k] = \\ \lambda x'_{k^s}.e'_{k^s}[\mathbf{if}_v\ C^k \in v_{cs}\ \mathbf{then}\ v'_k\ \mathbf{else}\ \bullet\ /h^s_k][v_k/h_k])_{k\in B\bigcap I}$$

By the form of v_{cs} and the reduction rule for $\mathbf{if}_v$ in Figure 2.4, this last equation is equivalent to

$$(\lambda x'_k.e'_k[v_k/h_k] = \lambda x'_{k^s}.e'_{k^s}[v'_k/h^s_k][v_k/h_k])_{k\in B\bigcap I} \tag{A.68}$$

The result follows from A.49 and A.53.

Case $v_{cs} = h'$) By Definition 3.7, it is enough to prove that

$$(\lambda x'_k.e'_k[v_k/h_k][D_n/h'] = \\ \lambda x'_{k^s}.e'_{k^s}[\mathbf{if}_v\ C^k \in h'\ \mathbf{then}\ v'_k\ \mathbf{else}\ \bullet\ /h^s_k][v_k/h_k][D_n/h'])_{k\in B\bigcap I}$$

for any D_n of the form $\{C^k\}_{k\in I}$. So, by substitution, $EV(v'_k) \subseteq h_k$ and $h_k \neq h'$, we have that this last equation is equivalent to

$$(\lambda x'_k.e'_k[v_k/h_k][D_n/h'] = \\ \lambda x'_{k^s}.e'_{k^s}[\mathbf{if}_v\ C^k \in D_n\ \mathbf{then}\ v'_k\ \mathbf{else}\ \bullet\ /h^s_k][v_k/h_k][D_n/h'])_{k\in B\bigcap I}$$

Finally, we have to proceed in the same way as for $v_{cs} = \{C^k\}_{k\in I}$.

A.18 Proof of proposition 4.7 from section 4.2

Proposition 4.7. *If* $\sigma \sim^U \sigma'$ *then* $U\,\sigma = U\,\sigma'$.

Proof: By induction on the derivation of $\sigma \sim^U \sigma'$. Extending Proposition 7.8 from [Martínez López, 2004].

The result follows trivially from IH and the definition of substitution.

A.19 Proof of proposition 4.8 from section 4.2

Proposition 4.8. *If* $S\,\sigma = S\,\sigma'$*, then* $\sigma \sim^U \sigma'$ *and there exists a substitution* T *such that* $S = TU$.

Proof: Extending Proposition *nose* from [Martínez López, 2004].

The extension that we have made does not modify the proof of this property. This proof consists on showing that every derivation of the form $\sigma \sim^U \sigma'$ is finite and then we have four cases to consider, which are proved using properties of substitutions.

A.20 Proof of proposition 4.9 from section 4.2

Proposition 4.9. *If* $\Delta' \mid \Delta \Vdash_{\mathrm{W}} \delta$ *then* $\Delta', \Delta \Vdash \delta$.

Proof: By definition of $\Delta' \mid \Delta \Vdash_{\mathrm{W}} \delta$ and (Fst).

A.21 Proof of proposition 4.10 from section 4.2

PROPOSITION 4.10. *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 Proposition 7.11 from [Martínez López, 2004].

Case (WSR-DDT)**:** It holds trivially by applying IH, taking

$$\Delta = (C^k \in t?\Delta_{w_k}, C^k \in t?\text{HasC } t \;\; C^k \;\; \tau'_{w_k})_{k \in D}, \text{ConstrsOf } t$$

and then applying (SR-DDT) because $\Delta \Vdash \Delta$.

A.22 Proof of proposition 4.11 from section 4.2

PROPOSITION 4.11. *If* $\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma$ *then* $\Delta'_w \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'_w$ *with all the residual variables fresh, and there exists* C'_w *such that* $C'_w : \text{Gen}_\emptyset(\Delta'_w \Rightarrow \tau'_w) \geq (\Delta \mid \sigma)$.

Proof: By induction on the SR derivation.
Extending Proposition 7.12 from [Martínez López, 2004].

Case (SR-DDT)**:** We know that

$$\frac{\begin{array}{l}(\Delta_k \vdash_{\text{SR}} D(C^k) \hookrightarrow \tau'_k)_{k \in D} \\ (\Delta \Vdash C^k \in \tau'?\Delta_k, C^k \in \tau'?\text{HasC } \tau' \;\; C^k \;\; \tau'_k)_{k \in D} \\ \Delta \Vdash \text{ConstrsOf } \tau'\end{array}}{\Delta \vdash_{\text{SR}} D^D \hookrightarrow \tau'}$$

By IH we know that there exist Δ_{w_k}, τ_{w_k} and C_{w_k} conversions such that

$$(C_{w_k} : \text{Gen}_\emptyset(\Delta_{w_k} \Rightarrow \tau_{w_k}) \geq (\Delta_k \mid \tau'_k))_{k \in D}$$

which means that there exist substitutions $(S_k)_{k \in D}$ and evidence $(v_k)_{k \in D}$ such that

$$(\tau'_k = S_k \, \tau'_{w_k})_{k \in D} \tag{A.69}$$
$$(h_k : \Delta_k \Vdash v_k : S_k \, \Delta_{w_k})_{k \in D} \tag{A.70}$$

We are under the assumption that $\bigcap_{k \in D} FTV(\text{dom}(S_k)) = \emptyset$ — If this is not the case, we can carry out α-conversions. We also know by hypothesis that

$$(\Delta \Vdash v^{sr}_k : C^k \in \tau'?\Delta_k)_{k \in D} \tag{A.71}$$
$$(\Delta \Vdash v^{sr}_{m_k} : C^k \in \tau'?\text{HasC } \tau' \;\; C^k \;\; \tau'_k)_{k \in D} \tag{A.72}$$
$$\Delta \Vdash v^{sr}_{cs} : \text{ConstrsOf } \tau' \tag{A.73}$$

We define $\tau'_w = t$, for t fresh,

$$\Delta'_w = ((C^k \in t?\Delta_{w_k}, C^k \in t?\text{HasC } t \;\; C^k \;\; \tau'_{w_k})_{k \in D}, \text{ConstrsOf } t)$$

and the substitution

$$S(t') = \begin{cases} \tau' & \text{if } t' = t \\ S_k(t') & \text{if } t' \in \mathrm{dom}(S_k) \end{cases}$$

Let us also define evidence

$$v = ((\mathbf{if}_v\ C^k \in v^{sr}_{cs}\ \mathbf{then}\ v_k[v^{sr}_k/h_k]\ \mathbf{else}\ \bullet, v^{sr}_{m_k})_{k\in D}, v^{sr}_{cs})$$

Now, we have to prove that

a) $\tau' = S\,t$

b) $\Delta \Vdash v : (S\,(C^k \in t?\Delta_{w_k}), S\,(C^k \in t?\mathrm{HasC}\ t\ \ C^k\ \ \tau'_{w_k}))_{k\in D}, S\,(\mathrm{ConstrsOf}\ t)$

in order to establish the existence of the desired convertion.

Item **a)** holds trivially by definition of S.

For item **b)**, distribute S and by its definition together with A.69, we obtain that

$$\Delta \Vdash C^k \in \tau'?S_k\,\Delta_{w_k}, C^k \in \tau'?\mathrm{HasC}\ \tau'\ \ C^k\ \ \tau'_k, \mathrm{ConstrsOf}\ \tau'$$

By (Univ), it is enough to prove that

$$\Delta \Vdash C^k \in \tau'?S_k\,\Delta_{w_k} \tag{A.74}$$
$$\Delta \Vdash C^k \in \tau'?\mathrm{HasC}\ \tau'\ \ C^k\ \ \tau'_k \tag{A.75}$$
$$\Delta \Vdash \mathrm{ConstrsOf}\ \tau' \tag{A.76}$$

Item **A.74)** holds by applying Lemma 3.9 to A.70, A.71 and A.73, obtaining

$$(\Delta \Vdash \mathbf{if}_v\ C^k \in v^{sr}_{cs}\ \mathbf{then}\ v_k[v^{sr}_k/h_k]\ \mathbf{else}\ \bullet : C^k \in \tau'?S_k\Delta_{w_k})_{k\in D}$$

Item **A.75)** holds trivially by A.72 with evidence $v^{sr}_{m_k}$.

Item **A.76)** holds trivially by A.73 with evidence v^{sr}_{cs}.

A.23 Proof of lemma 4.12 from section 4.2

Lemma 4.12. *If $h : \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau\ \hookrightarrow\ e' : \tau'$ then $EV(e') \subseteq h$*

Proof: By induction on the W derivation. Extending Lemma 7.13 from [Martínez López, 2004]. It holds trivially.

A.24 Proof of theorem 4.13 from section 4.2

THEOREM 4.13. *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 Theorem 7.14 from [Martínez López, 2004].

Case (W-DCONSTR): By applying IH, Proposition 4.10 on the SR derivations that we have in the hypothesis and Proposition 4.4 together with the fact that $h_w \neq h_{w_j}$.

Case (W-DCASE): We have to use a doble induction like in Proposition 4.14.

The proof follows easily by applying IH when necessary, Proposition 4.10, Proposition 4.4, Proposition 3.12, the unifications that we have in our hypothesis and the fact that every evidence variable is a fresh one.

A.25 Proof of theorem 4.14 from section 4.2

THEOREM 4.14. *If* $h : \Delta \mid S\,\Gamma \vdash_{\mathrm{S}} e : \tau \hookrightarrow e' : \tau'$, *then* $h'_w : \Delta'_w \mid T'_w\,\Gamma \vdash_{\mathrm{W}} e : \tau \hookrightarrow e'_w : \tau'_w$ *and there exists a substitution* R *and evidence* v'_w *such that*

a) $S \approx RT'_w$

b) $\tau' = R\,\tau'_w$

c) $h : \Delta \Vdash v'_w : R\,\Delta'_w$

d) $e' = e'_w[h'_w/v'_w]$

Proof:
By induction on S derivation. Extending Theorem 7.15 from [Martínez López, 2004].

Case (S-DCONSTR): We know that

$$\frac{\begin{array}{c}\Delta_s \mid S\,\Gamma \vdash_{\mathrm{S}} e : D(C^j) \hookrightarrow e'_s : \tau'_{j^s} \\ (\Delta^s_k \vdash_{\mathrm{SR}} D(C^k) \hookrightarrow \tau'_{k^s})_{k \in D \wedge k \neq j} \\ (\Delta_s \Vdash v^s_k : C^k \in \tau'_{e_s} ? \Delta^s_k, v^s_{m_k} : C^k \in \tau'_{e_s} ?\mathrm{HasC}\ \tau'_{e_s}\ \ C^k\ \ \tau'_{k^s})_{k \in D \wedge k \neq j} \\ \Delta_s \Vdash v^s_{m_j} : \mathrm{HasC}\ \tau'_{e_s}\ \ C^j\ \ \tau'_{j^s}, v^s_{cs} : \mathrm{ConstrsOf}\ \tau'_{e_s}\end{array}}{\Delta_s \mid S\,\Gamma \vdash_{\mathrm{S}} C^D_j\ e : D^D \hookrightarrow C^j_{v^s_{m_j}}\ e'_s : \tau'_{j^s}}$$

By IH on the first hypothesis of the rule, we have that

$$h_{w_j} : \Delta_{w_j} \mid T_{w_j}\,\Gamma \vdash_{\mathrm{W}} e : D(C^j) \hookrightarrow e'_{w_j} : \tau'_{w_j}$$

and there exist a substitution R^{IH} and evidence v_{IH} such that

$$S \approx R^{IH} T_{w_j} \tag{A.77}$$
$$\tau'_{j^s} = R^{IH}\, \tau'_{w_j} \tag{A.78}$$
$$h_s : \Delta_s \Vdash v_{IH} : R^{IH}\, \Delta_{w_j} \tag{A.79}$$
$$e' = e'_{w_j}[v_{IH}/h_{w_j}] \tag{A.80}$$

By Proposition 4.11 on the second hypothesis of the rule, we also know that there exist conversions $\{C_{w_k}\}_{k \in D \wedge k \neq j}$ such that

$$(C_{w_k} : \mathrm{Gen}_{\emptyset}(\Delta_{w_k} \Rightarrow \tau'_{w_k}) \geq (\Delta^s_k \mid \tau'_{k^s}))_{k \in D \wedge k \neq j} \tag{A.81}$$

By Definition 2.20, A.81 is equivalent to the existence of substitutions S_k and evidences v_k (for each k) such that

$$(\tau'_{k^s} = S_k \tau'_{w_k})_{k \in D \wedge k \neq j} \tag{A.82}$$
$$(h^s_k : \Delta^s_k \Vdash v_k : S_k \Delta_{w_k})_{k \in D \wedge k \neq j} \tag{A.83}$$

Without loosing generality, we assume that the sets $\mathrm{dom}(R^{IH})$, $\mathrm{dom}(S_k)_{k \in D \wedge k \neq j}$ and $\{t\}$ are disjoint (if that is not the case, α-conversions can be performed).

By application of $\Vdash_{\mathrm{W}}$, we obtain Δ_w such that

$$\begin{aligned} \Delta_w = (&(C^k \in t?\Delta_{w_k}, C^k \in t?\mathrm{HasC}\ t\ \ C^k\ \ \tau'_{w_k})_{k \in D \wedge k \neq j},\\ &\mathrm{HasC}\ t\ \ C^j\ \ \tau'_{w_j},\\ &\mathrm{ConstrsOf}\ t) \end{aligned} \tag{A.84}$$

Let us take $h'_w = ((h^w_{v_k}, h^w_{v_{m_k}})_{k \in D \wedge k \neq j}, h^w_{v_{m_j}}, h^w_{v_{cs}}, h_{w_j})$, $\Delta'_w = (\Delta_w, \Delta_{w_j})$, $T'_w = T_{w_j}$, $\tau'_w = t$,

$$R(t') = \begin{cases} \tau'_{e_s} & \text{if } t' = t \\ S_k(t') & \text{if } t' \in \mathrm{dom}(S_k) \\ R^{IH}(t') & \text{otherwise} \end{cases}$$

and $v'_w = ((\mathbf{if}_v\ C^k \in v^s_{cs}\ \mathbf{then}\ v_k[v^s_k/h^s_k]\ \mathbf{else}\ \bullet, v^s_{m_k})_{k \in D \wedge k \neq j}, v^s_{m_j}, v^s_{cs}, v_{IH})$. Now, we have to prove that

a) $S \approx R T_{w_j}$

b) $\tau'_{e_s} = R\, t$

c) $\Delta_s \Vdash v'_w : R\,(\Delta_w, \Delta_{w_j})$

d) $C^j_{v^s_{m_j}}\ e' = (C^j_{h^w_{v_{m_j}}}\ e'_{w_j})[v'_w/h_w, h_{w_j}]$

Item **a)** holds by definition of R and A.77. Observe that R just behaves as R^{IH} when it is composed with T_{w_j} — none of the variables that belongs to $\{t\}$ or $\mathrm{dom}(S_k)$ can appear in T_{w_j}.

Item **b)** holds trivially by definition of R.

For item **c)**, by A.84 we have to prove

$$\begin{aligned}\Delta_s \Vdash R(\ & (C^k \in t?\Delta_{w_k}, C^k \in t?\text{HasC } t \ \ C^k \ \ \tau'_{w_k})_{k\in D\wedge k\neq j},\\ & \text{HasC } t \ \ C^j \ \ \tau'_{w_j},\\ & \text{ConstrsOf } t,\\ & \Delta_{w_j}\ )\end{aligned}$$

This last entailment is equivalent to

$$\begin{aligned}\Delta_s \Vdash (& C^k \in Rt?R\Delta_{w_k}, C^k \in Rt?\text{HasC } (Rt) \ \ C^k \ \ (R\tau'_{w_k}))_{k\in D\wedge k\neq j},\\ & \text{HasC } (Rt) \ \ C^j \ \ (R\tau'_{w_j}),\\ & \text{ConstrsOf } (Rt),\\ & R\Delta_{w_j}\end{aligned}$$

which can be rewritten as follows

$$\begin{aligned}\Delta_s \Vdash (& C^k \in \tau'_{e_s}?S_k\Delta_{w_k}, C^k \in \tau'_{e_s}?\text{HasC } \tau'_{e_s} \ \ C^k \ \ (S_k\tau'_{w_k}))_{k\in D\wedge k\neq j},\\ & \text{HasC } \tau'_{e_s} \ \ C^j \ \ (R^{IH}\tau'_{w_j}),\\ & \text{ConstrsOf } \tau'_{e_s},\\ & R^{IH}\Delta_{w_j}\end{aligned}$$

By A.78 and A.82, we have that

$$\begin{aligned}\Delta_s \Vdash (& C^k \in \tau'_{e_s}?S_k\Delta_{w_k}, C^k \in \tau'_{e_s}?\text{HasC } \tau'_{e_s} \ \ C^k \ \ \tau'_{k^s})_{k\in D\wedge k\neq j},\\ & \text{HasC } \tau'_{e_s} \ \ C^j \ \ \tau'_{j^s},\\ & \text{ConstrsOf } \tau'_{e_s},\\ & R^{IH}\Delta_{w_j}\end{aligned}$$

In order to prove this entailment, it is enough to prove

$$(\Delta_s \Vdash C^k \in \tau'_{e_s}?S_k\Delta_{w_k})_{k\in D\wedge k\neq j} \tag{A.85}$$
$$(\Delta_s \Vdash C^k \in \tau'_{e_s}?\text{HasC } \tau'_{e_s} \ \ C^k \ \ \tau'_{k^s})_{k\in D\wedge k\neq j} \tag{A.86}$$
$$\Delta_s \Vdash \text{HasC } \tau'_{e_s} \ \ C^j \ \ \tau'_{j^s} \tag{A.87}$$
$$\Delta_s \Vdash \text{ConstrsOf } \tau'_{e_s} \tag{A.88}$$
$$\Delta_s \Vdash R^{IH}\Delta_{w_j} \tag{A.89}$$

A.85) can be proved using the third and fourth hypothesis, that is

$$(\Delta_s \Vdash v^s_k : C^k \in \tau'_{e_s}?\Delta^s_k)_{k\in D\wedge k\neq j} \tag{A.90}$$
$$\Delta_s \Vdash v^s_{cs} : \text{ConstrsOf } \tau'_{e_s} \tag{A.91}$$

Thus we can apply Lemma 3.9 with A.83, A.90 and A.91, proving

$$(\Delta_s \Vdash \mathbf{if}_v \ C^k \in v^s_{cs} \ \mathbf{then} \ v_k[v^s_k/h^s_k] \ \mathbf{else} \ \bullet : C^k \in \tau'_{e_s}?S_k\,\Delta_{w_k})_{k\in D\wedge k\neq j}$$

A.86) holds by third hypothesis with evidence $v^s_{m_k}$.

A.87) holds by fourth hypothesis with evidence $v^s_{m_j}$.

A.88) holds by fourth hypothesis with evidence v^s_{cs}.

A.89) holds by A.79 with evidence v_{IH}.

Item **d)**

$$\begin{aligned}
C^j_{v^s_{m_j}}\ e' &= (C^j_{h^w_{v_{m_j}}}\ e'_{w_j})[v'_w/h_w, h_{w_j}] \\
&= (C^j_{h^w_{v_{m_j}}}\ e'_{w_j})[(\textbf{if}_v\ C^k \in v^s_{cs}\ \textbf{then}\ v_k[v^s_k/h^s_k]\ \textbf{else}\ \bullet, \\
&\qquad v^s_{m_k})_{k\in D\wedge k\neq j}, v^s_{m_j}, v^s_{cs}, v_{IH}/h_w, h_{w_j}] \\
&= (C^j_{h^w_{v_{m_j}}}\ e'_{w_j})[(\textbf{if}_v\ C^k \in v^s_{cs}\ \textbf{then}\ v_k[v^s_k/h^s_k]\ \textbf{else}\ \bullet, \qquad (A.92)\\
&\qquad v^s_{m_k})_{k\in D\wedge k\neq j}, v^s_{m_j}, v^s_{cs}, v_{IH} \\
&\qquad /(h^w_{v_k}, h^w_{v_{m_k}})_{k\in D\wedge k\neq j}, h^w_{v_{m_j}}, h^w_{v_{cs}}, h_{w_j}]
\end{aligned}$$

We know, by Lemma 4.12, that evidence variables h_{w_j} appear only in e'_{w_j}. Additionally, h_{m_j} is a fresh evidence variable, so A.92 can be rewritten as

$$C^j_{v^s_{m_j}}\ e' = (C^j_{h^w_{v_{m_j}}}\ e'_{w_j})[v^s_{m_j}, v_{IH}/h^w_{v_{m_j}}, h_{w_j}]$$

which is the same that

$$C^j_{v^s_{m_j}}\ e' = C^j_{h^w_{v_{m_j}}[v^s_{m_j}/h^w_{v_{m_j}}]}\ e'_{w_j}[v_{IH}/h_{w_j}]$$

Applying substitution and A.80, we obtain

$$C^j_{v^s_{m_j}}\ e' = C^j_{v^s_{m_j}}\ e'$$

as we wanted.

Case (S-DCASE)**:** A **case** have several branches. So, it is necessary to do induction on number of them with a slightly more general property (observe **a′**)).

Case (1 – branch):

$$\frac{\begin{array}{l}
\Delta_s \vdash_{\text{SR}} \tau \hookrightarrow \tau'_s \\
\Delta_s \mid S\,\Gamma \vdash_{\text{S}} e : D^D \hookrightarrow e'_s : \tau'_{e_s} \\
h^s_1 : \Delta^s_1 \mid S\,\Gamma \vdash_{\text{S}} \lambda^D x_1.e_1 : D(C^1) \rightarrow^D \tau \hookrightarrow \lambda x'_{1^s}.e'_{1^s} : \tau'_{1^s} \rightarrow \tau'_s \\
\Delta_s \Vdash\ v^s_{m_1} : C^1 \in \tau'_{e_s} ?\text{HasC}\ \tau'_{e_s}\ \ C^1\ \ \tau'_{1^s}, \\
\qquad v^s_1 : C^1 \in \tau'_{e_s} ?\Delta^s_1, \\
\qquad v^s_{cs} : \text{ConstrsOf}\ \tau'_{e_s}
\end{array}}{\begin{array}{l}
\Delta_s \mid S\,\Gamma \vdash_{\text{S}} \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad C^D_1\ x_1 \rightarrow e_1 \\
\qquad : \tau \\
\qquad \hookrightarrow \\
\qquad \textbf{protocase}_v\ e'_s\ \textbf{with}\ v^s_{cs}\ \textbf{of} \\
\qquad\quad C^1_{v^s_{m_1}}\ x'_{1^s} \rightarrow e'_{1^s}[v^s_1/h^s_1] \\
\qquad : \tau'_s
\end{array}}$$

By the outer IH, we obtain a substitution R_e and evidences v_e such that

$$h_{w_e} : \Delta_{w_e} \mid T_{w_e}\,\Gamma \vdash_{\mathrm{W}} e : D^{D} \;\hookrightarrow\; e'_{w_e} : \tau'_{w_e} \tag{A.93}$$
$$S \approx R_e T_{w_e} \tag{A.94}$$
$$\tau'_{e_s} = R_e\,\tau'_{w_e} \tag{A.95}$$
$$\Delta_s \Vdash v_e : R_e\,\Delta_{w_e} \tag{A.96}$$
$$e'_s = e'_{w_e}[v_e/h_{w_e}] \tag{A.97}$$

By the third hypothesis of the rule and A.94, we have that

$$h^s_1 : \Delta^s_1 \mid R_e\,(T_{w_e}\,\Gamma) \vdash_{\mathrm{S}} \lambda^{D} x_1.e_1 : D(C^1) \rightarrow^{D} \tau \;\hookrightarrow\; \lambda x'_{1^s}.e'_{1^s} : \tau'_{1^s} \rightarrow \tau'_s \tag{A.98}$$

Applying the outer IH on A.98, we obtain a substitution R_1 and evidence v_1 such that

$$\begin{aligned} h^w_{w_1} : \Delta_{w_1} \mid T_{w_1}\,(T_{w_e}\,\Gamma) \vdash_{\mathrm{W}} \;& \lambda^{D} x_1.e_1 : D(C^1) \rightarrow^{D} \tau \\ & \hookrightarrow\; \lambda x'_1.e'_{w_1} : \tau'_{w^i_1} \rightarrow \tau'_{w^o_1} \end{aligned} \tag{A.99}$$
$$R_e \approx R_1 T_{w_1} \tag{A.100}$$
$$\tau'_{1^s} = R_1\,\tau'_{w^i_1} \tag{A.101}$$
$$\tau'_s = R_1\,\tau'_{w^o_1} \tag{A.102}$$
$$h^s_1 : \Delta^s_1 \Vdash v_1 : R_1\,\Delta_{w_1} \tag{A.103}$$
$$\lambda x'_{1^s}.e'_{1^s} = (\lambda x'_1.e'_{w_1})[v_1/h^w_{w_1}] \tag{A.104}$$

By Proposition 4.11, there exist a substitution R_{sr} and evidence v_{sr} such that

$$\Delta_{w_0} \vdash_{\text{W-SR}} \tau \;\hookrightarrow\; \tau'_{w_0} \tag{A.105}$$
$$\tau'_s = R_{sr}\,\tau'_{w_0} \tag{A.106}$$
$$h_s : \Delta_s \Vdash v_{sr} : R_{sr}\,\Delta_{w_0} \tag{A.107}$$

Every type variable appearing in the judgement from system W-SR in the hypothesis of the rule is fresh. So, we know that

$$\mathrm{dom}(R_{sr}) \bigcap FTV(R_1) = \emptyset \tag{A.108}$$

and

$$R_1 \approx R_{sr} R_1 \tag{A.109}$$

From this last two facts together with A.102 and A.106, we obtain

$$(R_{sr} R_1)\,\tau'_{w^o_1} = (R_{sr} R_1)\,\tau'_{w^o_0} \tag{A.110}$$

Now, from this last equation we know that exists an unifier U^1_0 and a substitution R'_1 such that

$$R_{sr} R_1 = R'_1 U^1_0 \tag{A.111}$$
$$\tau'_{w^o_1} \sim^{U^1_0} \tau'_{w^o_0} \tag{A.112}$$

Applying (W-DCASE) rule to A.105, A.93, A.99, A.112, and the predicates Δ_w (returned by $\Vdash_{\mathrm{W}}$) such that

$$\begin{aligned}\Delta_w = (&C^1 \in U_0^1\, T_{w_1}\, \tau'_{w_e} ? U_0^1\, \Delta_{w_1},\\ &C^1 \in U_0^1\, T_{w_1}\, \tau'_{w_e} ? \mathrm{HasC}\ (U_0^1\, T_{w_1}\, \tau'_{w_e})\ \ C^1\ \ (U_0^1\, \tau'_{w_1^i}),\\ &\mathrm{ConstrsOf}\ (U_0^1\, T_{w_1}\, \tau'_{w_e}))\end{aligned} \tag{A.113}$$

we obtain

$$\begin{array}{ll} & \Delta_w, U_0^1\, T_{w_1}\, \Delta_{w_e}, U_0^1\, \Delta_{w_0} \mid U_0^1\, T_{w_1}\, T_{w_e}\, \Gamma \\ \vdash_{\mathrm{W}} & \mathbf{case}\ e\ \mathbf{of} \\ & \qquad C_1^{D}\ x_1\ \rightarrow\ e_1 \\ & : \tau \\ & \hookrightarrow \\ & \mathbf{protocase}_v\ e'_{w_e}\ \mathbf{with}\ h^w_{v_{cs}}\ \mathbf{of} \\ & \quad C^1_{h^w_{v_{m_1}}}\ x'_1\ \rightarrow\ e'_{w_1}[h^w_{v_1}/h^w_{w_1}] \\ & : U_0^1\, \tau'_{w_1^o} \end{array} \tag{A.114}$$

Let us take $R = R'_1$ and

$$v'_w = (\mathbf{if}_v\ C^1 \in v^s_{cs}\ \mathbf{then}\ v_1[v^s_1/h^s_1]\ \mathbf{else}\ \bullet, v^s_{m_1}, v^s_{cs}, v_e, v_{sr})$$

Now, we have to prove that

a) $S \approx R'_1 U_0^1 T_{w_1} T_{w_e}$

a') $\tau'_{e_s} = R'_1\, U_0^1\, T_{w_1}\, T_{w_e}\, \tau'_{w_e}$

b) $\tau'_s = R'_1\, U_0^1\, \tau'_{w_1^o}$

c) $\Delta_s \Vdash v'_w : R'_1\, (\Delta_w, U_0^1\, T_{w_1}\, \Delta_{w_e}, U_0^1\, \Delta_{w_0})$

d) $$\begin{array}{ll}\mathbf{protocase}_v\ e'_s\ \mathbf{with}\ v^s_{cs}\ \mathbf{of} = & (\mathbf{protocase}_v\ e'_{w_e}\ \mathbf{with}\ h^w_{v_{cs}}\ \mathbf{of} \\ \quad C^1_{v^s_{m_1}}\ x'_{1^s}\ \rightarrow\ e'_{1^s}[v^s_1/h^s_1] & \quad C^1_{h^w_{v_{m_1}}}\ x'_1\ \rightarrow\ e'_{w_1}[h^w_{v_1}/h^w_{w_1}]) \\ & [\mathbf{if}_v\ C^1 \in v^s_{cs}\ \mathbf{then}\ v_1[v^s_1/h^s_1]\ \mathbf{else}\ \bullet \\ & , v^s_{m_1}, v^s_{cs}, v_e, v_{sr}/h^w_{v_1}, h^w_{v_{m_1}}, h^w_{v_{cs}}, h_{w_e}, h_{w_0}]\end{array}$$

Item **a)** holds by definition of R, A.94, A.100, A.109, and A.111.

Item **a')** holds by definition of R, A.95, A.100, A.109, and A.111.

Item **b)** holds by definition of R, A.102, A.109, and A.111.

Item **c)**. Using A.113, we have to prove

$$\begin{aligned}\Delta_s \Vdash R'_1\, (&C^1 \in U_0^1\, T_{w_1}\, \tau'_{w_e} ? U_0^1\, \Delta_{w_1},\\ &C^1 \in U_0^1\, T_{w_1}\, \tau'_{w_e} ? \mathrm{HasC}\ U_0^1\, T_{w_1}\, \tau'_{w_e}\ \ C^1\ \ U_0^1\, \tau'_{w_1^i},\\ &\mathrm{ConstrsOf}\ U_0^1\, T_{w_1}\, \tau'_{w_e},\\ &U_0^1\, T_{w_1}\, \Delta_{w_e},\\ &U_0^1\, \Delta_{w_0})\end{aligned}$$

which is equivalent to

$$\begin{aligned}\Delta_s \Vdash\ &C^1 \in R'_1\, U_0^1\, T_{w_1}\, \tau'_{w_e} ? R'_1\, U_0^1\, \Delta_{w_1},\\ &C^1 \in R'_1\, U_0^1\, T_{w_1}\, \tau'_{w_e} ? \mathrm{HasC}\ (R'_1\, U_0^1\, T_{w_1}\, \tau'_{w_e})\ \ C^1\ \ (R'_1\, U_0^1\, \tau'_{w_1^i}),\\ &\mathrm{ConstrsOf}\ (R'_1\, U_0^1\, T_{w_1}\, \tau'_{w_e}),\\ &R'_1\, U_0^1\, T_{w_1}\, \Delta_{w_e},\\ &R'_1\, U_0^1\, \Delta_{w_0})\end{aligned}$$

Applying A.95, A.100, A.101, A.108, A.109, and A.111, we obtain

$$\begin{aligned}\Delta_s \Vdash\ & C^1 \in \tau'_{e_s}?R_1\,\Delta_{w_1},\\ & C^1 \in \tau'_{e_s}?\mathrm{HasC}\ \tau'_e\ \ C^1\ \ \tau'_{1^s},\\ & \mathrm{ConstrsOf}\ \tau'_{e_s},\\ & R_e\,\Delta_{w_e},\\ & R_{sr}\,\Delta_{w_0}\end{aligned}$$

In order to prove this entailment, it is enough to prove

$$\Delta_s \Vdash C^1 \in \tau'_{e_s}?R_1\,\Delta_{w_1} \tag{A.115}$$

$$\Delta_s \Vdash C^1 \in \tau'_{e_s}?\mathrm{HasC}\ \tau'_{e_s}\ \ C^1\ \ \tau'_{1^s} \tag{A.116}$$

$$\Delta_s \Vdash \mathrm{ConstrsOf}\ \tau'_{e_s} \tag{A.117}$$

$$\Delta_s \Vdash R_e\,\Delta_{w_e} \tag{A.118}$$

$$\Delta_s \Vdash R_{sr}\,\Delta_{w_0} \tag{A.119}$$

Item **A.115)**. By hypothesis, it is very easy to verify that

$$\Delta_s \Vdash v_1^s : C^1 \in \tau'_{e_s}?\Delta_1^s, v_{cs}^s : \mathrm{ConstrsOf}\ \tau'_{e_s} \tag{A.120}$$

Applying Lemma 3.9 with A.103 and A.120, we obtain

$$\Delta_s \Vdash \mathbf{if}_v\ C^1 \in v_{cs}^s\ \mathbf{then}\ v_1[v_1^s/h_1^s]\ \mathbf{else}\ \bullet : C^1 \in \tau'_{e_s}?R_1\,\Delta_{w_1}$$

Item **A.116)** holds trivially by hypothesis with evidence $v_{m_1}^s$.

Item **A.118)** holds by A.96.

Item **A.117)** holds by hypothesis with evidence v_{cs}^s.

Item **A.119)** holds by A.107 with evidence v_{sr}

Item **d)** We have that

$$\begin{array}{ll}\mathbf{protocase}_v\ e'_s\ \mathbf{with}\ v_{cs}^s\ \mathbf{of} = & (\mathbf{protocase}_v\ e'_{w_e}\ \mathbf{with}\ h_{v_{cs}}^w\ \mathbf{of}\\ \quad C^1_{v_{m_1}^s}\ x'_{1^s}\ \rightarrow\ e'_{1^s}[v_1^s/h_1^s] & \quad C^1_{h^w_{v_{m_1}}}\ x'_1\ \rightarrow\ e'_{w_1}[h_{v_1}^w/h_{w_1}^w])\\ & [\mathbf{if}_v\ C^1 \in v_{cs}^s\ \mathbf{then}\ v_1[v_1^s/h_1^s]\ \mathbf{else}\ \bullet\\ & , v_{m_1}^s, v_{cs}^s, v_e, v_{sr}/h_{v_1}^w, h_{v_{m_1}}^w, h_{v_{cs}}^w, h_{w_e}, h_{w_0}]\end{array}$$

Since h_{w_0} does not appear in the residual $\mathbf{protocase}_v$, the last equation is equivalent to

$$\begin{array}{l}(\mathbf{protocase}_v\ e'_{w_e}\ \mathbf{with}\ h_{v_{cs}}^w\ \mathbf{of}\\ \quad C^1_{h^w_{v_{m_1}}}\ x'_1\ \rightarrow\ e'_{w_1}[h_{v_1}^w/h_{w_1}^w])\\ \quad [\mathbf{if}_v\ C^1 \in v_{cs}^s\ \mathbf{then}\ v_1[v_1^s/h_1^s]\ \mathbf{else}\ \bullet, v_{m_1}^s, v_{cs}^s, v_e/h_{v_1}^w, h_{v_{m_1}}^w, h_{v_{cs}}^w, h_{w_e}]\end{array}$$

By Lemma 4.12 and substitutions, we know that $EV(e'_{w_e}) \subseteq h_{w_e}$ and $EV(e'_{w_1}[h_{v_1}^w/h_{w_1}^w]) \subseteq h_{v_1}^w$, where $h_{w_e} \neq h_{w_1}^w \neq h_{v_1}^w$. The evidence variables $h_{v_{m_1}}^w$ and $h_{v_{cs}}^w$ are fresh, so the last equation is equivalent to

$$\begin{array}{l}\mathbf{protocase}_v\ e'_{w_e}[v_e/h_{w_e}]\ \mathbf{with}\ h_{v_{cs}}^w[v_{cs}^s/h_{v_{cs}}^w]\ \mathbf{of}\\ \quad C^1_{h^w_{v_{m_1}}[v_{m_1}^s/h^w_{v_{m_1}}]}\ x'_1\ \rightarrow\ e'_{w_1}[h_{v_1}^w/h_{w_1}^w][\mathbf{if}_v\ C^1 \in v_{cs}^s\ \mathbf{then}\ v_1[v_1^s/h_1^s]\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \bullet\ /h_{v_1}^w]\end{array}$$

Applying substitutions and A.97, we obtain that

$$\begin{array}{l}\textbf{protocase}_v\ e'_s\ \textbf{with}\ v^s_{cs}\ \textbf{of}\\ \quad C^1_{v^s_{m_1}}\ x'_1\ \rightarrow\ e'_{w_1}[h^w_{v_1}/h^w_{w_1}][\textbf{if}_v\ C^1 \in v^s_{cs}\ \textbf{then}\ v_1[v^s_1/h^s_1]\ \textbf{else}\ \bullet/h^w_{v_1}]\end{array}$$

Since $EV(v_1) \subseteq h^s_1$, $EV(v^s_1) \subseteq h_s$, $EV(v^s_{cs}) \subseteq h_s$ and $h^s_1 \neq h_s \neq h^w_{v_1}$, we have that

$$\begin{array}{l}\textbf{protocase}_v\ e'_s\ \textbf{with}\ v^s_{cs}\ \textbf{of}\\ \quad C^1_{v^s_{m_1}}\ x'_1\ \rightarrow\ e'_{w_1}[\textbf{if}_v\ C^1 \in v^s_{cs}\ \textbf{then}\ v_1\ \textbf{else}\ \bullet/h^w_{w_1}][v^s_1/h^s_1]\end{array}$$

Summarizing, we have to prove that

$$\begin{array}{lcl}\textbf{protocase}_v\ e'_s\ \textbf{with}\ v^s_{cs}\ \textbf{of} & = & \textbf{protocase}_v\ e'_s\ \textbf{with}\ v^s_{cs}\ \textbf{of}\\ \quad C^1_{v^s_{m_1}}\ x'_{1^s}\ \rightarrow\ e'_{1^s}[v^s_1/h^s_1] & & \quad C^1_{v^s_{m_1}}\ x'_1\ \rightarrow\ e'_{w_1}\ [\textbf{if}_v\ C^1 \in cs^s\ \textbf{then}\ v_1\\ & & \qquad\qquad\qquad \textbf{else}\ \bullet/h^w_{w_1}][v^s_1/h_1]\end{array}$$

In order to prove this, we have to use the definitions appearing in Figures 3.6 and 3.7.

Case $v^s_{cs} = \{C^k\}_{k\in I}$) By Definition 3.6, we only need to consider when $1 \in I$. So, it is enough to prove that

$$\begin{array}{l}\lambda x'_{1^s}.e'_{1^s}[v^s_1/h^s_1] =\\ \lambda x'_1.e'_{w_1}[\textbf{if}_v\ C^1 \in v^s_{cs}\ \textbf{then}\ v_1\ \textbf{else}\ \bullet/h^w_{w_1}][v^s_1/h^s_1]\end{array} \tag{A.121}$$

By the form of v^s_{cs} and the reduction rule for $\textbf{if}_v$ in Figure 2.4, this last equation is equivalent to

$$\lambda x'_{1^s}.e'_{1^s}[v^s_1/h^s_1] = \lambda x'_1.e'_{w_1}[v_1/h^w_{w_1}][v^s_1/h^s_1]$$

The result follows from A.104.

Case $v^s_{cs} = h'$) By Definition 3.7, it is enough to prove that

$$\begin{array}{l}\lambda x'_{1^s}.e'_{1^s}[v^s_1/h^s_1][D_n/h'] =\\ \lambda x'_1.e'_{w_1}[\textbf{if}_v\ C^1 \in h'\ \textbf{then}\ v_1\ \textbf{else}\ \bullet/h^w_{w_1}][v^s_1/h^s_1][D_n/h']\end{array} \tag{A.122}$$

for any D_n of the form $\{C^k\}_{k\in I}$. By substitution, $EV(v_1) \subseteq h^s_1$ and $h' \neq h^s_1$, we have that this last equation is equivalent to

$$\begin{array}{l}\lambda x'_{1^s}.e'_{1^s}[v^s_1/h^s_1][D_n/h'] =\\ \lambda x'_1.e'_{w_1}[\textbf{if}_v\ C^1 \in D_n\ \textbf{then}\ v_1\ \textbf{else}\ \bullet/h^w_{w_1}][v^s_1/h^s_1][D_n/h']\end{array}$$

Finally, we only have to consider when $1 \in D_n$, proceeding in the same way as for $v^s_{cs} = \{C^k\}_{k\in I}$.

Case $((B+1) -$ **branches):** We suppose that this theorem is valid for a set of indexes B and we will prove that it will be also valid for a set of indexes $B+1$, which has one more index than B and this index is the maximum,

being also annotated as $B+1$. So, observe that $B+1$ is overloaded and its meaning depends on the context where it is used.
We know that

$$\begin{aligned}\Delta_s \mid S\,\Gamma \vdash_{\mathrm{S}} & \ \mathbf{case}\ e\ \mathbf{of} \\ & \qquad (C^{D}_{k}\ \ x_k \ \rightarrow\ e_k)_{k\in B} \\ & : \tau \\ & \hookrightarrow \\ & \mathbf{protocase}_v\ e'_s\ \mathbf{with}\ v^s_{cs}\ \mathbf{of} \\ & \quad (C^{k}_{v^s_{m_k}}\ \ x'_{k^s} \ \rightarrow\ e'_{k^s}[v^s_k/h^s_k])_{k\in B} \\ & : \tau'_s\end{aligned} \tag{A.123}$$

is related to

$$\begin{aligned}& \Delta^B_w, A^B_1\,\Delta_{w_e}, A^B_2\,U^1_0\,\Delta_{w_0} \mid A^B_1\,T_{w_e}\,\Gamma \\ & \vdash_{\mathrm{W}}\ \ \mathbf{case}\ e\ \mathbf{of} \\ & \qquad\qquad (C^{D}_{k}\ \ x_k \ \rightarrow\ e_k)_{k\in B} \\ & \quad\ : \tau \\ & \quad\ \hookrightarrow \\ & \quad\ \mathbf{protocase}_v\ e'_{w_e}\ \mathbf{with}\ h^w_{v_{cs}}\ \mathbf{of} \\ & \qquad (C^{k}_{h^w_{v m_k}}\ \ x'_{k} \ \rightarrow\ e'_{w_k}[h^w_{v_k}/h^w_{w_k}])_{k\in B} \\ & \quad\ : U^B_{B-1}\,\tau'_{w_B}\end{aligned} \tag{A.124}$$

by means of a substitution R^{IH_B} and evidence $v_{w_{IH_B}}$ such that

$$S \approx R^{IH_B} A^B_1\,T_{w_e} \tag{A.125}$$

$$\tau'_{e_s} = R^{IH_B} A^B_1\,T_{w_e}\,\tau'_{w_e} \tag{A.126}$$

$$\tau'_s = R^{IH_B}\,U^B_{B-1}\,\tau'_{w_B} \tag{A.127}$$

$$\Delta_s \Vdash v_{IH_B} : R^{IH_B}\ \ (\Delta^B_w, A^B_1\,\Delta_{w_e}, A^B_2\,U^1_0\,\Delta_{w_0}) \tag{A.128}$$

$$\begin{aligned}& \mathbf{protocase}_v\ e'_s\ \mathbf{with}\ v^s_{cs}\ \mathbf{of} \\ & \quad (C^{k}_{v^s_{m_k}}\ \ x'_{k^s} \ \rightarrow\ e'_{k^s}[v^s_k/h^s_k])_{k\in B} \\ & = (\mathbf{protocase}_v\ e'_{w_e}\ \mathbf{with}\ h^w_{v_{cs}}\ \mathbf{of} \\ & \qquad (C^{k}_{h^w_{v m_k}}\ \ x'_{k} \ \rightarrow\ e'_{w_k}[h^w_{v_k}/h^w_{w_k}])_{k\in B} \\ & \qquad [v_{w_{IH_B}}/h^B_w, h_{w_e}, h_{w_0}]\end{aligned} \tag{A.129}$$

Observe that A.126 is the property that we have added for branch induction. Now, we use A.125 in order to rewrite the specialization of the $(B+1)$th branch in the hypothesis of the S derivation, obtaining

$$\begin{aligned}& h^s_{B+1} : \Delta^s_{B+1} \mid R^{IH_B}(A^B_1\,T_{w_e}\,\Gamma) \vdash_{\mathrm{S}} \\ & \quad \lambda^D x_{B+1}.e_{B+1} : D(C^{B+1}) \rightarrow^D \tau\ \hookrightarrow\ \lambda x'_{B+1^s}.e'_{B+1^s} : \tau'_{B+1^s} \rightarrow \tau'_s\end{aligned} \tag{A.130}$$

We apply the outer IH on A.130, so there exist a substitution R_{B+1} and evidence v_{B+1} such that

$$\begin{aligned}& h^w_{w_{B+1}} : \Delta_{w_{B+1}} \mid T_{w_{B+1}}\,A^B_1\,T_{w_e}\,\Gamma \vdash_{\mathrm{W}} \\ & \quad \lambda^D x_{B+1}.e_{B+1} : D(C^{B+1}) \rightarrow^D \tau\ \hookrightarrow\ \lambda x'_{B+1}.e'_{w_{B+1}} : \tau'_{w^i_{B+1}} \rightarrow \tau'_{w^o_{B+1}}\end{aligned} \tag{A.131}$$

$$R^{IH_B} \approx R_{B+1} T_{B+1} \tag{A.132}$$
$$\tau'_{B+1^s} = R_{B+1}\, \tau'_{w^i_{B+1}} \tag{A.133}$$
$$\tau'_s = R_{B+1}\, \tau'_{w^o_{B+1}} \tag{A.134}$$
$$\Delta^s_{B+1} \Vdash v_{B+1} : R_{B+1}\, \Delta_{w_{B+1}} \tag{A.135}$$
$$\lambda x'_{B+1^s}.e'_{B+1^s} = (\lambda x'_{B+1}.e'_{w_{B+1}})[v_{B+1}/h^w_{w_{B+1}}] \tag{A.136}$$

Using A.127, A.132, and A.134, we obtain

$$R_{B+1}\, \tau'_{w^o_{B+1}} = R_{B+1}\, T_{B+1}\, U^B_{B-1}\, \tau'_{w^o_B} \tag{A.137}$$

Then, there exist an unifier U^{B+1}_B and a substitution R'_{B+1} such that

$$\tau'_{w^o_{B+1}} \sim^{U^{B+1}_B} T_{w_{B+1}}\, U^B_{B-1}\, \tau'_{w^o_B} \tag{A.138}$$
$$R_{B+1} = R'_{B+1} U^{B+1}_B \tag{A.139}$$

We can apply (W-DCASE) rule with judgements $\vdash_{\mathrm{W}}$ and unifiers appearing in hypothesis of A.124, A.131, and A.138, obtaining

$$\begin{array}{l}
\Delta_w, U^{B+1}_B\, T_{B+1}\, A^B_1\, \Delta_{w_e}, U^{B+1}_B\, T_{B+1}\, A^B_2\, U^1_0\, \Delta_{w_0} \mid U^{B+1}_B\, T_{B+1}\, A^B_1\, T_{w_e}\, \Gamma \\
\vdash_{\mathrm{W}} \ \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad (C^D_k\ \ x_k\ \ \rightarrow\ \ e_k)_{k \in B+1} \\
\quad : \tau \\
\quad \hookrightarrow \\
\quad \textbf{protocase}_v\ e'_{w_e}\ \textbf{with}\ h^w_{v_{cs}}\ \textbf{of} \\
\qquad (C^k_{h^w_{v_{m_k}}}\ \ x'_k\ \ \rightarrow\ \ e'_{w_k}[h^w_{v_k}/h^w_{w_k}])_{k \in B+1} \\
\quad : U^{B+1}_B\, \tau'_{w^o_{B+1}}
\end{array} \tag{A.140}$$

We define $v_{IH_B} = (v^B_w, v_{w_e}, v_{sr})$ where v^B_w, v_{w_e} and v_{sr} come from applying (Univ) three times to A.128. In other words, we have

$$\Delta_s \Vdash v^B_w : R^{IH_B}\, \Delta^B_w \tag{A.141}$$
$$\Delta_s \Vdash v_{w_e} : R^{IH_B}\, A^B_1\, \Delta_{w_e} \tag{A.142}$$
$$\Delta_s \Vdash v_{sr} : R^{IH_B}\, A^B_2\, U^1_0\, \Delta_{w_0} \tag{A.143}$$

Let us take $R = R'_{B+1}$ and

$$v'_w = (v^B_w, \mathbf{if}_v\ C^{B+1} \in v^s_{cs}\ \mathbf{then}\ v_{B+1}[v^s_{B+1}/h^s_{B+1}]\ \mathbf{else}\ \bullet, v^s_{m_{B+1}}, v_{w_e}, v_{sr})$$

Now, we have to prove that

a) $S \approx R'_{B+1} U^{B+1}_B\, T_{B+1}\, A^B_1\, T_{w_e}$

a') $\tau'_{e_s} = R'_{B+1} U^{B+1}_B\, T_{B+1}\, A^B_1\, T_{w_e}\, \tau'_{w_e}$

b) $\tau'_s = R'_{B+1}\, U^{B+1}_B\, \tau'_{w^o_{B+1}}$

c) $\Delta_s \Vdash v'_w : R'_{B+1}\, (\Delta_w, U^{B+1}_B\, T_{B+1}\, A^B_1\, \Delta_{w_e}, U^{B+1}_B\, T_{B+1}\, A^B_2\, U^1_0\, \Delta_{w_0})$

Item **a)** holds by A.125, A.132, and A.139.

Item **a')** holds by A.126, A.132, and A.139.

Item **b)** holds by A.134, and A.139.

For item **c)**, it is very easy to verify that

$$\begin{aligned}\Delta_w = {} & U_B^{B+1}\, T_{B+1}\, \Delta_w^B, \\ & C^{B+1} \in U_B^{B+1}\, T_{B+1}\, A_1^B\, \tau_{w_e} ? U_B^{B+1}\, \Delta_{w_{B+1}}, \\ & C^{B+1} \in U_B^{B+1}\, T_{B+1}\, A_1^B\, \tau_{w_e} ? \\ & \quad \text{HasC}\ (U_B^{B+1}\, T_{B+1}\, A_1^B\, \tau_{w_e})\ \ C^{B+1}\ \ (U_B^{B+1}\, \tau'_{w^i_{B+1}})\end{aligned} \tag{A.144}$$

Applying (Univ) several times to item **c)** and distributing substitution inside predicates, it is enough to prove

$$\Delta_s \Vdash R'_{B+1}\, U_B^{B+1}\, T_{B+1}\, \Delta_w^B \tag{A.145}$$

$$\Delta_s \Vdash C^{B+1} \in R'_{B+1}\, U_B^{B+1}\, T_{B+1}\, A_1^B\, \tau_{w_e} ? R'_{B+1}\, U_B^{B+1}\, \Delta_{w_{B+1}} \tag{A.146}$$

$$\begin{aligned}\Delta_s \Vdash\ \ & C^{B+1} \in U_B^{B+1}\, T_{B+1}\, A_1^B\, \tau_{w_e} ? \\ & \quad \text{HasC}\ (U_B^{B+1}\, T_{B+1}\, A_1^B\, \tau_{w_e})\ \ C^{B+1}\ \ (U_B^{B+1}\, \tau'_{w^i_{B+1}})\end{aligned} \tag{A.147}$$

$$\Delta_s \Vdash R'_{B+1}\, U_B^{B+1}\, T_{B+1}\, A_1^B\, \Delta_{w_e} \tag{A.148}$$

$$\Delta_s \Vdash R'_{B+1}\, U_B^{B+1}\, T_{B+1}\, A_2^B\, U_0^1\, \Delta_{w_0} \tag{A.149}$$

Item **A.145)** It holds by A.132, A.139, and A.141.

For item **A.146)**, it is very easy to verify, by hypothesis, that

$$\Delta_s \Vdash v^s_{B+1} : C^{B+1} \in \tau'_{e_s} ? \Delta^s_{B+1}, v^s_{cs} : \text{ConstrsOf}\ \tau'_{e_s} \tag{A.150}$$

By A.135, A.139, item **a')**, and application of Lemma 3.9 to A.135 and A.150, we obtain

$$\begin{aligned}\Delta_s \Vdash {} & \textbf{if}_v\ \ C^{B+1} \in v^s_{cs}\ \textbf{then}\ v_{B+1}[v^s_{B+1}/h^s_{B+1}]\ \textbf{else}\ \bullet : \\ & \quad C^{B+1} \in R'_{B+1}\, U_B^{B+1}\, T_{B+1}\, A_1^B\, \tau_{w_e} ? R'_{B+1}\, U_B^{B+1}\, \Delta_{w_{B+1}}\end{aligned}$$

Item **A.147)** holds by Item **a')**, A.133, and A.139, obtaining that

$$\Delta_s \Vdash C^{B+1} \in \tau'_{e_s} ? \text{HasC}\ \tau'_{e_s}\ \ C^{B+1}\ \ \tau'_{B+1^s} \tag{A.151}$$

which it is true by hypothesis with evidence v^s_{B+1}.

Item **A.148)** It holds by A.132, A.139, and A.142.

Item **A.149)** It holds by A.132, A.139, and A.143.

Item **d)** We have that

$$\begin{aligned}& (\textbf{protocase}_v\ e'_{w_e}\ \textbf{with}\ h^w_{v_{cs}}\ \textbf{of} \\ & \quad (C^k_{h^w_{v_{m_k}}}\ \ x'_k\ \rightarrow\ e'_{w_k}[h^w_{v_k}/h^w_{w_k}])_{k \in B+1}) \\ & \quad [v^B_w, \textbf{if}_v\ \ C^{B+1} \in v^s_{cs}\ \textbf{then}\ v_{B+1}[v^s_{B+1}/h^s_{B+1}]\ \textbf{else}\ \bullet, \\ & \quad v^s_{m_{B+1}}, v_{w_e}, v_{sr}/h^B_w, h^w_{w_{B+1}}, h^w_{v_{m_{B+1}}}, h_{w_e}, h_{w_0}]\end{aligned} \tag{A.152}$$

Since $EV(v^s_{cs}) \subseteq h_s$, $EV(v_{B+1}[v^s_{B+1}/h^s_{B+1}]) \subseteq h_s$, $EV(v_{w_e}) \subseteq h_{w_e}$ in addition to $EV(v_{sr}) \subseteq h_{sr}$, and $h_s \neq h_{w_e} \neq h_{sr}$, we obtain that A.152 is equivalent to

$$\begin{aligned}& (\textbf{protocase}_v\ e'_{w_e}\ \textbf{with}\ h^w_{v_{cs}}\ \textbf{of} \\ & \quad (C^k_{h^w_{v_{m_k}}}\ \ x'_k\ \rightarrow\ e'_{w_k}[h^w_{v_k}/h^w_{w_k}])_{k \in B+1}) \\ & \quad [v^B_w, v_{w_e}, v_{sr}, \textbf{if}_v\ \ C^{B+1} \in v^s_{cs}\ \textbf{then}\ v_{B+1}[v^s_{B+1}/h^s_{B+1}]\ \textbf{else}\ \bullet, \\ & \quad v^s_{m_{B+1}}/h^B_w, h_{w_e}, h_{w_0}, h^w_{w_{B+1}}, h^w_{v_{m_{B+1}}}]\end{aligned} \tag{A.153}$$

and by definition of v_{IH_B}, this can be rewritten as

$$\begin{array}{l}(\textbf{protocase}_v\ e'_{w_e}\ \textbf{with}\ h^w_{v_{cs}}\ \textbf{of}\\ \quad (C^k_{h^w_{v_{m_k}}}\ x'_k\ \rightarrow\ e'_{w_k}[h^w_{v_k}/h^w_{w_k}])_{k\in B+1})\\ \quad [v_{IH_B}, \textbf{if}_v\ C^{B+1} \in v^s_{cs}\ \textbf{then}\ v_{B+1}[v^s_{B+1}/h^s_{B+1}]\ \textbf{else}\ \bullet,\\ \quad v^s_{m_{B+1}}/h^B_w, h_{w_e}, h_{w_0}, h^w_{w_{B+1}}, h^w_{v_{m_{B+1}}}]\end{array} \tag{A.154}$$

Because of evidence variables $h^w_{w_{B+1}}$ and $h^w_{v_{B+1}}$ appear only in the $(B+1)$th branch, A.154 is equivalent to

$$\begin{array}{l}(\textbf{protocase}_v\ e'_{w_e}\ \textbf{with}\ h^w_{v_{cs}}\ \textbf{of}\\ \quad (C^k_{h^w_{v_{m_k}}}\ x'_k\ \rightarrow\ e'_{w_k}[h^w_{v_k}/h^w_{w_k}])_{k\in B})[v_{IH_B}/h^B_w, h_{w_e}, h_{w_0}]\\ \quad (C^{B+1}_{h^w_{v_{m_{B+1}}}}\ x'_{B+1}\ \rightarrow\ e'_{w_{B+1}}[h^w_{v_{B+1}}/h^w_{w_{B+1}}])\\ \quad [\textbf{if}_v\ C^{B+1} \in v^s_{cs}\ \textbf{then}\ v_{B+1}[v^s_{B+1}/h^s_{B+1}]\ \textbf{else}\ \bullet, v^s_{m_{B+1}}/h^w_{w_{B+1}}, h^w_{v_{m_{B+1}}}]\end{array}$$

To sum up, we have to prove that

$$\begin{array}{l}\textbf{protocase}_v\ e'_s\ \textbf{with}\ v^s_{cs}\ \textbf{of}\\ \quad (C^k_{v^s_{m_k}}\ x'_{k^s}\ \rightarrow\ e'_{k^s}[v^s_k/h^s_k])_{k\in B+1}\\ = (\textbf{protocase}_v\ e'_{w_e}\ \textbf{with}\ h^w_{v_{cs}}\ \textbf{of}\\ \qquad (C^k_{h^w_{v_{m_k}}}\ x'_k\ \rightarrow\ e'_{w_k}[h^w_{v_k}/h^w_{w_k}])_{k\in B})[v_{IH_B}/h^B_w, h_{w_e}, h_{w_0}]\\ \qquad (C^{B+1}_{h^w_{v_{m_{B+1}}}}\ x'_{B+1}\ \rightarrow\ e'_{w_{B+1}}[h^w_{v_{B+1}}/h^w_{w_{B+1}}])\\ \qquad [\textbf{if}_v\ C^{B+1} \in v^s_{cs}\ \textbf{then}\ v_{B+1}[v^s_{B+1}/h^s_{B+1}]\ \textbf{else}\ \bullet, v^s_{m_{B+1}}\\ \qquad /h^w_{w_{B+1}}, h^w_{v_{m_{B+1}}}]\end{array}$$

In order to do this, we only need to consider the $(B+1)$th branch because A.129 already satisfies Definitions 3.6 and 3.7.

Case $v_{cs} = \{C^k\}_{k\in I}$) We have to follow the same steps used to prove A.121 but considering $B+1, x'_{B+1^s}, e'_{B+1^s}, v_{B+1}, h^w_{w_{B+1}}$, and A.136 instead of $1, x'_{1^s}, e'_{1^s}, v_1, h^w_{w_1}$, and A.104.

Case $v_{cs} = h'$) We have to follow the same steps used to prove A.122 but this time considering $B+1, x'_{B+1^s}, e'_{B+1^s}, v_{B+1}$, and $h^w_{w_{B+1}}$ instead of $1, x'_{1^s}, e'_{1^s}, v_1$, and $h^w_{w_1}$.

Index