Principal Type Specialization of Dynamic Sum-Types
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

Copyright © 2004 by Alejandro C. Russo
e-mail: russo@fceia.unr.edu.ar
http://www.fceia.unr.edu.ar/~russo/
For my grandfather Antonio,
who was a skilled worker that always worked to improve my country.
# Contents

Agradecimientos  xi  
Acknowledgments  xiii  

1 Introduction  1  
   1.1 Program Specialization  1  
   1.2 Principal Type Specialization  3  
   1.3 Contribution of this Work  3  
   1.4 Overview  4  

2 Specialization  5  
   2.1 Type Specialization  5  
      2.1.1 Source Language  5  
      2.1.2 Residual Language  6  
      2.1.3 Specializing  7  
      2.1.4 Examples  7  
   2.2 Qualified Types  9  
      2.2.1 Predicates and Entailment  9  
      2.2.2 Type inference with Qualified Types  11  
      2.2.3 Coherence and Evidence  13  
   2.3 Principal Type Specialization  14  
      2.3.1 Residual Language  14  
      2.3.2 Residual Types  16  
      2.3.3 Specifying Principal Type Specialization  17  
      2.3.4 Algorithm  21  
      2.3.5 Extensions  23  

3 Principal Specialization of Dynamic Sum-Types  27  
   3.1 Extending Source and Residual Language  27  
   3.2 Extending Source And Residual Types  29  
   3.3 Extending RT Relation  30  

vii
3.4 Specializations Rules for Dynamic Sum-Types .................................. 30
3.4.1 Entailment Relationship ...................................................... 31
3.4.2 Source-Residual Relationship .............................................. 32
3.4.3 Rules for Constructors and Case ........................................... 33
3.5 Examples ................................................................. 35

4  The Algorithm and the Proof, Extended  .............................................. 43
4.1 Extension of The Syntax Directed System, S .................................. 43
4.2 Extension of The Inference Algorithm, W .................................... 45
4.2.1 An entailment algorithm .................................................... 45
4.2.2 An algorithm for source-residual relationship ......................... 46
4.2.3 An algorithm for type specialisation .................................... 46
4.3 Examples ................................................................. 48

5  Extension to the Prototype ....................................................... 57
5.1 Implementation Language .................................................... 57
5.2 Previous Work ............................................................ 57
5.3 Extensions ................................................................. 58
5.4 Potential Improvements ...................................................... 58
5.5 Conclusions of the Implementation ........................................... 58

6  Conclusions and Future Work .................................................... 61

7  Simplification and Constraint Solving ............................................ 63
7.1 Simplification ............................................................. 63
7.1.1 Motivation ............................................................. 63
7.1.2 Specification .......................................................... 64
7.1.3 Implementing a Simplification ........................................ 66
7.1.4 Simplification during specialisation ................................... 67
7.1.5 Extension to Simplification ............................................ 68
7.2 Constraint Solving ....................................................... 70
7.2.1 Motivation ............................................................. 70
7.2.2 Specifying Solutions .................................................. 71
7.2.3 Solving and Specialisation ........................................... 71
7.2.4 Extending the Algorithm for Constraint Solving ..................... 72

8  An Interpreter With Error Handling .............................................. 75
8.1 Running Example .......................................................... 75

Bibliography ................................................................. 79

A  Proofs ................................................................. 83
A.1 Proof of proposition 3.7 from section 3.3 ................................. 83
A.2 Proof of theorem 3.8 from section 3.3 ..................................... 84
A.3 Proof of lemma 3.9 from section 3.4 ...................................... 84
A.4 Proof of proposition 3.11 from section 3.4 ............................... 85
A.5  Proof of proposition 3.12 from section 3.4 .......................... 86
A.6  Proof of theorem 3.13 from section 3.4 ............................ 86
A.7  Proof of theorem 3.14 from section 3.4 ............................ 86
A.8  Proof of theorem 3.15 from section 3.4 ............................ 88
A.9  Proof of proposition 3.16 from section 3.4 ........................ 88
A.10 Proof of proposition 3.17 from section 3.4 ........................ 89
A.11 Proof of lemma 3.18 from section 3.4 .............................. 90
A.12 Proof of lemma 3.19 from section 3.4 .............................. 91
A.13 Proof of proposition 4.2 from section 4.1 ........................ 91
A.14 Proof of proposition 4.3 from section 4.1 ........................ 91
A.15 Proof of proposition 4.4 from section 4.1 ........................ 91
A.16 Proof of theorem 4.5 from section 4.1 ........................... 91
A.17 Proof of theorem 4.6 from section 4.1 ........................... 92
A.18 Proof of proposition 4.7 from section 4.2 ........................ 97
A.19 Proof of proposition 4.8 from section 4.2 ........................ 97
A.20 Proof of proposition 4.9 from section 4.2 ........................ 97
A.21 Proof of proposition 4.10 from section 4.2 ......................... 98
A.22 Proof of proposition 4.11 from section 4.2 ......................... 98
A.23 Proof of lemma 4.12 from section 4.2 ............................ 99
A.24 Proof of theorem 4.13 from section 4.2 .......................... 100
A.25 Proof of theorem 4.14 from section 4.2 .......................... 100
A.26 Proof of theorem 7.11 from section 7.1 .......................... 111
A.27 Proof of theorem 7.12 from section 7.1 .......................... 112
A.28 Proof of theorem 7.16 from section 7.2 .......................... 113
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 $x^n$

$$\text{power } n \ x = \begin{cases} \ x, & \text{if } n = 1 \\ x \times \text{power } (n-1) \times, & \text{else} \end{cases}$$

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

$$\text{power3 } x = x \times (x \times x)$$
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 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 28 that is, the source expression 28 : _Int_ can be specialized to ∗ : 28, where ∗ 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 inherited 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, 1994a], 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 summarize 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: \( \_ \) or \( \_ \). 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:

\[
\begin{align*}
  e & ::= x \\
  & | \text{lift } e \\
  & | \lambda^0 x.e \\
  & | (e, \ldots, e)^D \\
  & | \text{poly } e
\end{align*}
\]

\[
\begin{align*}
  n^p & | e^+^p e \\
  n^s & | e^+^s e \\
  e \text{ @ } n & | \text{let }^p x = e \text{ in } e \\
  \pi^D_{n,n} e & | \text{spec } e
\end{align*}
\]

where \( (e_1, \ldots, e_n)^D \) is a finite tuple of \( e \)'s for every possible arity \( n \). The projections \( \pi_{1,2}^D e \) and \( \pi_{2,2}^D e \) may be abbreviated \( \text{fst}^D e \) and \( \text{snd}^D e \) 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 $\tau$, is an element of the language defined by the following grammar:

$$\tau ::= \text{Int}^o \mid \text{Int}^s \mid (\tau, \ldots, \tau)^D \mid \tau \rightarrow^D \tau \mid \text{poly } \tau$$

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:

$$e' ::= x' \mid n \mid e' + e' \mid \bullet \mid \lambda x'.e' \mid e'@e' \mid \text{let } x' = e' \text{ in } e'$$

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 $\text{fst } e'$ and $\text{snd } e'$ 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.
2.1. Type Specialization

Definition 2.4. A residual type, denoted by \( \tau' \), is an element of the language defined by the grammar:

\[
\tau' ::= \text{Int} \mid \hat{n} \mid \tau' \rightarrow \tau' \mid (\tau'_1, \ldots, \tau'_n)
\]

where \((\tau'_1, \ldots, \tau'_n)\) is a finite tuple of \(\tau's\) 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 \(\Gamma\), is a (finite) list of specialization statements of the form \(x : \tau \leftarrow e' : \tau'\), 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

\[\Gamma \vdash e : \tau \leftarrow e' : \tau'\]

which denotes that the program \(e\) with source type \(\tau\) can be specialized to a residual program \(e'\) with residual type \(\tau'\), under the hypothesis \(\Gamma\) (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 \(\check{\_}^D\), specializes to a copy of itself with the annotations removed).

1. \(\vdash 42^D : \text{Int}^D \leftarrow 42 : \text{Int}\)
2. \(\vdash 42^s : \text{Int}^s \leftarrow 42\)
3. \(\vdash (2^D + 1^D) + 1^D : \text{Int}^D \leftarrow (2 + 1) + 1 : \text{Int}\)
4. \(\vdash (2^s + 1^s) + 1^s : \text{Int}^s \leftarrow \hat{4}\)
5. \(\vdash \text{lift} (2^s + 1^s) + 1^D : \text{Int}^D \leftarrow 3 + 1 : \text{Int}\)

Also observe in 5 how the use of \text{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. \( x : \text{Int}^s \leftrightarrow \circ : \hat{3} \vdash x +^{s} 1^{s} : \text{Int}^s \leftrightarrow \circ : \hat{4} \)

2. \( \vdash (\lambda x.x +^{s} 1^{s}) \circ \circ (2^{s} +^{s} 1^{s}) : \text{Int}^s \leftrightarrow (\lambda x'.\circ) \circ \circ : \hat{4} \)

3. \( \vdash (\lambda x.\text{lift} x +^{D} 1^{D}) \circ \circ (2^{s} +^{s} 1^{s}) : \text{Int}^D \leftrightarrow (\lambda x'.3 + 1) \circ \circ : \text{Int} \)

In 3 there is information that was moved from the context of the function to the function’s body, where \text{lift} places it.

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

\[
\text{let}\, f = \lambda x.\text{lift} \, x +^{D} 1^{D} \\
\text{in} \, (f \circ \circ 42^s, f \circ \circ 17^s)^D : (\text{Int}^D, \text{Int}^D)^D.
\]

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 \text{poly}.

Example 2.9. Observe the use of \text{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 \text{spec} in every application of \( f \) to an argument (and how that produces the corresponding projections).

1. \( \vdash \text{let}\, f = \text{poly} \, (\lambda x.\text{lift} \, x +^{D} 1^{D}) \\
\text{in} \, (\text{spec} \, f \circ \circ 42^s, \text{spec} \, f \circ \circ 17^s)^D : (\text{Int}^D, \text{Int}^D)^D \leftrightarrow \\
\text{let} \, f' = (\lambda x'.42 + 1, \lambda x'.17 + 1) \\
\text{in} \, (\text{fst} \, f' \circ \circ \circ, \text{snd} \, f' \circ \circ \circ) : \text{(Int, Int)} \)

2. \( \vdash \text{let}\, f = \text{poly} \, (\lambda x.\text{lift} \, x +^{D} 1^{D}) \\
\text{in} \, (\text{spec} \, f \circ \circ 42^s, \text{spec} \, f \circ \circ 17^s)^D : (\text{Int}^D, \text{Int}^D)^D \leftrightarrow \\
\text{let} \, f' = (\lambda x'.17 + 1, \lambda x'.55 + 1, \lambda x'.42 + 1) \\
\text{in} \, (\pi_{3,3} f' \circ \circ \circ, \pi_{1,3} f' \circ \circ \circ) : \text{(Int, Int)} \)

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 shows 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. \( \lambda x. x +^{s} 1^{s} : \text{Int}^s \rightarrow^{D} \text{Int}^s \)
2.2. Qualified Types

2. \textbf{poly} \( (\lambda^x x. \text{lift} x +^n 1^D) : \text{poly} \ (\mathit{Int}^s \rightarrow^D \mathit{Int}^D) \)

3. \( \lambda f. \text{spec} f @^n 13^s : \text{poly} \ (\mathit{Int}^s \rightarrow^D \mathit{Int}^D) \rightarrow^D \mathit{Int}^D \)

All have many different \textit{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 \textit{lack of principality} and fixing it requires a big change in the residual language.

2.2 Qualified Types

The theory of qualified types [Jones, 1994a] 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 \textit{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 \textit{overloading}, in which a function may have different interpretations according to the type's or 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 \textit{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 \textit{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 predators that we use are described in Section 2.3.1.

In [Jones, 1994a], 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
altogether in a set. For example, this amounts to define either that \( \rho := \delta \Rightarrow \tau \) or
\( \rho := \Delta \Rightarrow \tau \), being \( \Delta \) 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 \( \Delta = \delta_1, \ldots, \delta_m \), a list of type vari-
ables \( \alpha = \alpha_1, \ldots, \alpha_m \), a list of evidence variables \( h = h_1, \ldots, h_m \), and a list of evidence
expressions \( v = v_1, \ldots, v_m \), we use the abbreviations:

<table>
<thead>
<tr>
<th>Object</th>
<th>Expression</th>
<th>Abbreviation(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qualified type</td>
<td>( \delta_1 \Rightarrow \cdots \Rightarrow \delta_m \Rightarrow \tau' )</td>
<td>( \Delta \Rightarrow \tau' )</td>
</tr>
<tr>
<td>Type scheme</td>
<td>( \forall \alpha_1, \ldots, \forall \alpha_m, \rho )</td>
<td>( \forall \alpha, \rho )</td>
</tr>
<tr>
<td>Evidence abstr.</td>
<td>( \Lambda h_1, \ldots, \Lambda h_m, e' )</td>
<td>( \Lambda h, e' )</td>
</tr>
<tr>
<td>Evidence app.</td>
<td>( (e'(v_1)) \cdots (v_m) )</td>
<td>( e'(v) )</td>
</tr>
</tbody>
</table>

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 \( \tau \) 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 \( \Delta \) 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.
2.2. Qualified Types

\[
\begin{align*}
(Fst) & \quad h : \Delta, h' : \Delta' \vdash h : \Delta \\
(Snd) & \quad h : \Delta, h' : \Delta' \vdash h' : \Delta' \\
(Univ) & \quad \frac{h : \Delta \vdash v' : \Delta', h : \Delta \vdash v'' : \Delta''}{h : \Delta \vdash v', v'' : \Delta', \Delta''} \\
(Trans) & \quad \frac{h : \Delta \vdash v' : \Delta', h' : \Delta' \vdash v'' : \Delta''}{h : \Delta \vdash v'[h'/v'] : \Delta''} \\
(Close) & \quad \frac{h : \Delta \vdash v' : \Delta'}{h : S \Delta \vdash v' : S \Delta'}
\end{align*}
\]

Figure 2.1: Structural laws satisfied by entailment.

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

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 \(\tau\)) are defined by a grammar with at least these productions \(\tau := t | \tau \rightarrow \tau\). On top of types are constructed qualified types of the form \(\Delta \Rightarrow \tau\) (denoted by \(\rho\)), and then type schemes of the form \(\forall \{\alpha_i\}, \rho\) (denoted by \(\sigma\)). 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 \(\lambda\)-calculus (it has, at least, variables, applications, abstractions, and the \text{let} construct); it is called OML,
\[
\begin{align*}
\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 \quad \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} \quad \text{[\(\alpha \notin \text{FV}(\Delta) \cup \text{FV}(\Gamma)\)]} \\
\text{(INST)} & \quad \frac{\Delta \mid \Gamma \vdash e : \forall \alpha. \sigma}{\Delta \mid \Gamma \vdash e : S \sigma} \quad \text{[\(\text{dom}(s) = \alpha\)}]
\end{align*}
\]

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 \(\Delta\) are satisfied, and the types of the free variables of \(e\) are as specified by \(\Gamma\), then the term \(e\) has type \(\sigma\). The type system is in essence similar to a Hindley-Milner one, with the rules showed in Figure 2.2 added. The rules \text{(GEN)} and \text{(INST)} are used to generalize and instantiate type variables in type schemes, and \text{(QIN)} and \text{(QOUT)} to manage predicates.

Definition 2.14. A constrained type scheme is an expression of the form \((\Delta \mid \sigma)\) where \(\Delta\) is a set of predicates and \(\sigma\) is a type scheme.

In order to find all the ways in which a particular \(e\) can be used within a given \(\Gamma\), 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 \(\sigma\) in an environment satisfying the predicates in \(\Delta\) whenever an object of type \(\sigma'\) is required in an environment satisfying the predicates in \(\Delta'\). To define it formally, the notion of generic instance is needed.

Definition 2.15. A qualified type \(\Delta_r \Rightarrow \tau\) is a generic instance of the constrained type scheme \((\Delta \mid \forall \alpha_i. \Delta' \Rightarrow \tau')\) if there are types \(\tau_i\) such that

\[
\Delta_r \vdash \Delta, \Delta'[\alpha_i/\tau_i] \text{ and } \tau = \tau'[\alpha_i/\tau_i]
\]

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.
2.2.Qualified Types

\[
\begin{align*}
\text{(QIN)} & \quad \frac{\Delta, h : \delta \mid \Gamma \vdash e \leftrightarrow e' : \rho}{\Delta \mid \Gamma \vdash \lambda h.e' : \delta \Rightarrow \rho} \\
\text{(QOUT)} & \quad \frac{\Delta \mid \Gamma \vdash e \leftrightarrow e' : \delta \Rightarrow \rho \quad \Delta \vdash v : \delta}{\Delta \mid \Gamma \vdash e \leftrightarrow e'(v) : \rho}
\end{align*}
\]

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, 1994a] introduces the notion of evidence, and provide a translation from the original language of terms, OML, to one manipulating evidence explicitly 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 \(\Delta\) 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 \(\sigma\) to \(\sigma'\) is a collection of OP terms that allow the transformation of any OP term of type \(\sigma\) into an OP term of type \(\sigma'\) 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 \(\sigma = \forall \alpha_i.\Delta_i \Rightarrow \tau\) and \(\sigma' = \forall \beta_i.\Delta'_i \Rightarrow \tau'\) be two type schemes, and suppose that none of the \(\beta_i\) appears free in \(\sigma, \Delta\), or \(\Delta'\). 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 $\tau_i$, evidence variables $h'$ and $h'_r$, and evidence expressions $v$ and $v'$ such that:

- $\tau' = \tau[\alpha_i/\tau_i]$
- $h': \Delta', h'_r: \Delta'_r \vdash v : \Delta, v': \Delta_r[\alpha_i/\tau_i]$, and
- $C = (\lambda x. h', h'_r, x((v')))$

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 $\lambda$-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, 1994a].

#### 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 ($\rho$) and type schemes ($\sigma$); also particular predicates ($\delta$) are defined. The most important innovations with respect to the theory of qualified types are the new type construct $\text{poly } \sigma$, 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
2.3. Principal Type Specialization

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

\[(\eta_v) \quad \Lambda h.e'_1((h)) \triangleright e'_1 \quad (h \notin \text{EV}(e'_1))\]

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

\[(\text{case}_v) \quad (v_1 \circ v_2)[e'] \triangleright v_1[v_2[e']]\]

\[(\text{if}_v \text{-True}) \quad \text{if}_v \ \text{True} \ \text{then} \ e'_1 \ \text{else} \ e'_2 \triangleright e'_1\]

\[(\text{if}_v \text{-False}) \quad \text{if}_v \ \text{False} \ \text{then} \ e'_1 \ \text{else} \ e'_2 \triangleright e'_2\]

Figure 2.4: Reduction for residual terms.

disjoint with any other set of variables already used. A residual type, denoted by \(\tau'\), is an element of the language given by the grammar

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

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 \(\text{IsInt}\) and \(\_ := \_ + \_\) and conversions, as evidence for predicates of the form \(\text{IsMG}\). 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, 1994a].

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

\[\begin{align*}
e' &::= x' \mid n \mid e' + e' \mid \bullet \\
&\quad \mid \lambda x'.e' \mid e'@e' \\
&\quad \mid (e'_1, \ldots, e'_n) \mid \pi_{n,n} e' \\
&\quad \mid h \mid v[e'] \mid \Lambda h.e' \mid e'((v)) \\
&\quad \mid \text{let}_v \ x = e' \ \text{in} \ e' \\
v &::= h \mid n \mid C \\
&\quad \mid v @ v \\
C &::= [] \mid \Lambda h.C \mid C((v)) \mid \text{let}_v \ x = C \ \text{in} \ e'
\end{align*}\]

We will be working under a equivalence = on residual terms, defined as the minimal relation that contains \(\alpha\)-conversions for both \(\lambda\) and \(\Lambda\)-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 \(\hat{+}\), whose structural properties 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 \(\text{IsInt}\) is provable when the type is a one-point
\(\{\text{IsInt}\} \quad \Delta \vdash n : \text{IsInt} \; \hat{n}\)

\(\{\text{IsOp}\} \quad \Delta \vdash n : \hat{n} := \hat{n}_1 + \hat{n}_2 \quad \text{(wherever } n = n_1 + n_2)\)

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

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

\(\{\text{Comp}\} \quad \frac{\Delta \vdash v : \text{IsMG} \; \sigma_1 \; \sigma_2}{\Delta \vdash \tau' \circ v : \text{IsMG} \; \sigma_1 \; \sigma_3}\)

Figure 2.5: Entailment for evidence construction.

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 \(\sigma = \forall \alpha_i. \Delta_x \Rightarrow \tau\) and \(\sigma' = \forall \beta_i. \Delta'_x \Rightarrow \tau'\) be two type schemes, and suppose that none of the \(\beta_i\) appears free in \(\sigma\), \(h : \Delta\), 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 \(\tau_i\), evidence variables \(h_r\) and \(h'_r\), and evidence expressions \(v\) and \(v'\) such that:

- \(\tau' = \tau[\alpha_i/\tau_i]\)
- \(h' : \Delta', h'_r : \Delta'_r \vdash v : \Delta, v' : \Delta_r[\alpha_i/\tau_i], \text{ and}\)
- \(C = (\text{let}_o \; x = \lambda h._[\square] \; \text{in} \; \lambda h'_r.x((v)((v'))))\)

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

The following assertions hold when \(\sigma, \sigma', \sigma''\) are scheme variables:

1. \(\square : (\Delta \mid \sigma) \geq (\Delta \mid \sigma)\)
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 $\Delta$ it holds that

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

In [Martínez López and Hughes, 2004], a type system to infer the type of a residual expression 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 expression 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 $\lambda^x.x : \text{Int}^s \rightarrow^p \text{Int}^s$ can be specialized to $\lambda^x'x' : \text{Bool} \rightarrow \text{Bool}$.

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

Rules (QN) 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

$\vdash_r \lambda^x.x : \text{Int}^s \rightarrow^p \text{Int}^s \rightsquigarrow \lambda h_t, h_v. \lambda x'. h_v : \forall t, t'. \text{IsInt } t, t' := t + \hat{1} \Rightarrow t \rightarrow t'$

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.
Figure 2.6: Rules defining the source-residual relationship.
2.3. Principal Type Specialization

\[
\begin{align*}
\text{(VAR)} & \quad x : \tau \Rightarrow x' : \tau' \in \Gamma \\
\Delta & \mid \Gamma \vdash_p x : \tau \Rightarrow x' : \tau'
\end{align*}
\]

\[
\begin{align*}
\text{(INT)} & \quad \Delta \mid \Gamma \vdash_p n^\rho : \text{Int}^\rho \Rightarrow n : \text{Int} \\
\text{(D+)} & \quad (\Delta \mid \Gamma \vdash_p e_i : \text{Int}^\rho \Rightarrow e'_i : \text{Int})_{i=1,2} \\
& \quad \Delta \mid \Gamma \vdash_p e_1 +^\rho e_2 : \text{Int}^\rho \Rightarrow e'_1 + e'_2 : \text{Int} \\
\text{(LIFT)} & \quad \Delta \mid \Gamma \vdash_p e : \text{Int}^s \Rightarrow e' : \tau' \\
& \quad \Delta \mid \Gamma \vdash_p \text{lift } e : \text{Int}^\rho \Rightarrow v : \text{Int} \\
\text{(INT)} & \quad \Delta \mid \Gamma \vdash_p n^s : \text{Int}^s \Rightarrow \bullet : \hat{n} \\
\text{(S+)} & \quad (\Delta \mid \Gamma \vdash_p e_i : \text{Int}^s \Rightarrow e'_i : \tau'_i)_{i=1,2} \\
& \quad \Delta \mid \Gamma \vdash_p e_1 +^s e_2 : \text{Int}^s \Rightarrow \bullet : \tau' \\
\text{(DUPL)} & \quad \Delta \mid \Gamma \vdash_p (e_1, \ldots, e_n)^\rho : (\tau_1, \ldots, \tau_n)^\rho \Rightarrow (e'_1, \ldots, e'_n) : (\tau'_1, \ldots, \tau'_n) \\
\text{(DPRJ)} & \quad \Delta \mid \Gamma \vdash_p \pi_{i,n} e : \tau_i \Rightarrow \pi_{i,n} e' : \tau'_i \\
\text{(LAM)} & \quad \Delta \mid \Gamma, x : \tau_2 \Rightarrow x' : \tau'_2 \vdash_p e : \tau_1 \Rightarrow e' : \tau'_1 \\
& \quad \Delta \mid \Gamma \vdash_s \tau_2 \Rightarrow \tau'_2 \quad \text{(x' fresh)} \\
\text{(DAPP)} & \quad \Delta \mid \Gamma \vdash_p e_1 : \tau_2 \Rightarrow e'_1 : \tau'_1 \\
& \quad \Delta \mid \Gamma \vdash_p e_2 : \tau_2 \Rightarrow e'_2 : \tau'_2 \\
\text{(LET)} & \quad \Delta \mid \Gamma \vdash_p \text{let } e : \tau_2 \Rightarrow x : \tau'_2 \vdash_p e_1 : \tau_1 \Rightarrow e'_1 : \tau'_1 \\
& \quad \text{(x' fresh)} \\
\text{(POLY)} & \quad \Delta \mid \Gamma \vdash_p e : \tau \Rightarrow e' : \sigma' \\
& \quad \Delta \vdash v : \text{IsMG } \sigma' \sigma \\
& \quad \Delta \mid \Gamma \vdash_p \text{poly } e : \text{poly } \tau \Rightarrow v[e'] : \text{poly } \sigma \\
\text{(SPEC)} & \quad \Delta \mid \Gamma \vdash_p e : \text{poly } \tau \Rightarrow e' : \text{poly } \sigma \\
& \quad \Delta \vdash v : \text{IsMG } \sigma' \tau' \\
& \quad \Delta \mid \Gamma \vdash_s \tau \Rightarrow \tau' \\
\end{align*}
\]

Figure 2.7: Specialisation rules (first part)
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:

\[
\vdash_{\mathfrak{p}} \textbf{poly} \left( \lambda^{x}. \textbf{lift} \ x +^{D} 1^{D} \right) : \textbf{poly} \ (\textbf{Int}^{s} \rightarrow^{D} \textbf{Int}^{D})
\]

\[
\leftarrow \Lambda h, h[\Lambda h_{x}, \lambda^{x’.} h_{x} + 1] : \forall s. \textbf{IsMG} \ (\forall t. \textbf{IsInt} \ t \Rightarrow t \rightarrow \textbf{Int}) \ s \ \Rightarrow \textbf{poly} \ s
\]

In this example, the polyvariant function is abstracted by conversion \(h\), which abstracts
the evidence of the type \(\forall t. \textbf{IsInt} \ t \Rightarrow t \rightarrow \textbf{Int}\), 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 polyvariant 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.

\[
\vdash_{\mathfrak{p}} \textbf{let}^{D} \ f = \textbf{poly} \left( \lambda^{x}. \textbf{lift} \ x +^{D} 1^{D} \right)
\]

\[
\text{in} \ (\textbf{spec} \ f @^{D} 42^{s} , \textbf{spec} \ f @^{D} 17^{s})^{D}
\]

\[
: (\textbf{Int}^{D} , \textbf{Int}^{D})^{D}
\]

\[
\leftarrow \textbf{let} \ f’ = \Lambda h_{x}, \lambda x’. h_{x} + 1
\]

\[
\text{in} \ (f'(42) @ \bullet , f'(17) @ \bullet)
\]

\[
: (\textbf{Int} , \textbf{Int})
\]

Observe the interaction between annotations \textbf{poly} and \textbf{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:

\[
\vdash_{\mathfrak{p}} \lambda^{x}. \textbf{spec} \ f @^{D} 13^{s} : \textbf{poly} \ (\textbf{Int}^{s} \rightarrow^{D} \textbf{Int}^{D}) \rightarrow^{D} \textbf{Int}^{D}
\]

\[
\leftarrow \Lambda h_{u}, h_{u} , \lambda f’. h_{y} [f’] @ \bullet : \forall s. \textbf{IsMG} \ (\forall t. \textbf{IsInt} \ t \Rightarrow t \rightarrow \textbf{Int}) \ s,
\]

\[
\textbf{IsMG} \ s \ (13 \rightarrow \textbf{Int}) , \Rightarrow \textbf{poly} \ s \rightarrow \textbf{Int}
\]
This example shows a higher-order function that receives a polyvariant function as its argument to apply it to a specific static value. The argument function must be an instance of \( \forall t. {\text{IsInt}} \ t \Rightarrow t \Rightarrow {\text{Int}} \), but at least as general as \( 13 \Rightarrow {\text{Int}} \). 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 \( \vdash_p \) is stable under substitutions, an essential property for principality.

**Proposition 2.26.** If \( \Delta \vdash \Gamma \vdash_p \ e : \tau \iff e' : \sigma \) then \( S \Delta \vdash S \Gamma \vdash_p \ e : \tau \iff e' : S \sigma \).

Specialization also respects the \( \vdash \) relation.

**Proposition 2.27.** If \( h : \Delta \vdash \Gamma \vdash_p \ e : \tau \iff e' : \tau' \) and \( h' : \Delta' \vdash v : \Delta \), then \( h' : \Delta' \vdash \Gamma \vdash_p \ e \iff e'[v/h] : \tau' \).

### 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, 1994a]. This algorithm is based on the Milner’s W algorithm [Milner, 1978], and the rules can be interpreted as an attribute grammar [Remy, 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 \( \sigma_c \sim^U \sigma_e \), 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 \( \Delta \) in order to be able to entail a given predicate \( \delta \). The input is the current predicate assignment and the predicate \( \delta \) to entail, and the output is the set of predicates to add and the evidence proving \( \delta \). The result can be easily achieved by adding \( \delta \) to \( \Delta \) with a new variable \( h \). So, the only rule that is necessary for this algorithm is

\[
h : \delta \vdash \Delta \vdash_W h : \delta \quad (h \text{ fresh})
\]

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 \( \text{IsInt} \ n \)) or predicates already appearing in \( \Delta \), but all these cases can be handled by simplification and constraint solving phases [Badenes and Martínez López, 2002].
\[
\begin{align*}
(W\text{-VAR}) & \quad \frac{x : \tau \mapsto e' : \tau' \in \Gamma}{\emptyset \mid \text{Id} \Gamma \vdash_w x : \tau \mapsto e' : \tau'} \\
(W\text{-DINT}) & \quad \frac{\Delta_1 \mid S_1 \Gamma \vdash_w e_1 : \text{Int}^s \quad \Delta_2 \mid S_2 \Gamma \vdash_w e_2 : \text{Int}^s}{\Delta_1, \Delta_2 \mid S_2 \Gamma \vdash_w e_1 + e_2 : \text{Int}^s \quad \Delta_1 \vdash_w e_1 \mapsto e_1' \quad \Delta_2 \vdash_w e_2 \mapsto e_2'} \\
(W\text{-LIFT}) & \quad \frac{\Delta \mid S \Gamma \vdash_w e : \text{Int}^s \mapsto e' : \tau' \quad \Delta' \mid \Delta \vdash_w v : \text{Int}^s}{\Delta', \Delta \mid S \Gamma \vdash_w \text{lift} e : \text{Int}^s \mapsto v : \text{Int}} \\
(W\text{-SINT}) & \quad \frac{\emptyset \mid \text{Id} \Gamma \vdash_w n^s : \text{Int}^s}{\bullet : \hat{n}} \\
(W\text{-S}+) & \quad \frac{\Delta_1 \mid S_1 \Gamma \vdash_w e_1 : \text{Int}^s \mapsto e_1' : \tau_1 \quad \Delta_2 \mid S_2 \Gamma \vdash_w e_2 : \text{Int}^s \mapsto e_2' : \tau_2'}{\Delta, \Delta_1, \Delta_2 \mid S_2 \Gamma \vdash_w e_1 + \hat{e}_2 : \text{Int}^s \mapsto \bullet : t} \\
(W\text{-DLAM}) & \quad \frac{\Delta \mid \Gamma, x : \tau_2 \mapsto x' : \tau_2' \vdash_w e : \tau_1 \mapsto e' : \tau_1'}{\Delta', \Delta \mid S \Gamma \vdash_w \lambda x.e : \tau_2 \mapsto \tau_1 \mapsto \lambda \lambda x'.e' : \Sigma \tau_2' \mapsto \tau_1'} \\
(W\text{-DAPP}) & \quad \frac{\Delta_1 \mid S_1 \Gamma \vdash_w e_1 : \tau_2 \mapsto \tau_1 \mapsto e_1' : \tau_1' \quad \Delta_2 \mid S_2 \Gamma \vdash_w e_2 : \tau_2 \mapsto \tau_2' \mapsto e_2' : \tau_2'}{U \Delta_2 \Delta_1, U \Delta_2 \mid U S_2 \Gamma \vdash_w e_1 @ \hat{e}_2 : \tau_1 \mapsto e_1' \hat{e}_2' : U t} \\
(W\text{-POLY}) & \quad \frac{\Delta \mid \text{Id} \Gamma \vdash_w \text{poly} e : \text{poly} \tau \mapsto e' : \tau' \quad \Delta' \mid \emptyset \vdash_w v : \text{IsMG} (\text{Gen}_{\Gamma} (\Delta \mapsto \tau')) s}{\Delta' \mid S \Gamma \vdash_w \text{poly} e : \text{poly} \tau \mapsto v[\Delta h, e'] : \text{poly} s} \\
(W\text{-SPEC}) & \quad \frac{\Delta \mid S \Gamma \vdash_w \text{poly} \tau \mapsto e' : \tau_\sigma \quad \Delta' \mid \text{Id} \Gamma \vdash_w v : \text{IsMG} (U s) \tau'}{\Delta'' \mid U \Delta, \Delta' \vdash_w \text{spec} e : \tau \mapsto v[e'] : \tau'}
\end{align*}
\]

Figure 2.9: Type Specialisation Algorithm (first part).
2.3. Principal Type Specialization

\[
\frac{\Delta \mid S_1 \Gamma \vdash_w e_1 : \tau_1 \leadsto e'_1 : \tau'_1 \quad \cdots \quad \Delta_n \mid S_n S_{n-1} \ldots S_1 \Gamma \vdash_w e_n : \tau_n \leadsto e'_n : \tau'_n}{S_n \ldots S_2 \Delta_1, \ldots, \Delta_n \mid S_n \ldots S_1 \Gamma \vdash_w (e_1, \ldots, e_n)^p : (\tau_1, \ldots, \tau_n)^p \quad \leadsto (e'_1, \ldots, e'_n) : (S_n \ldots S_2 \tau'_1, S_n \ldots S_3 \tau'_2, \ldots, \tau'_n)}
\]

(W-DTUPLE)

\[
\frac{\Delta \mid S \Gamma \vdash_w e : (\tau_1, \ldots, \tau_n)^p \leadsto e' : \tau' \sim^U (t_1, \ldots, t_n)}{U \Delta \mid U S \Gamma \vdash_w \pi_{i,n}^p e : \tau_i \leadsto \pi_{i,n} e' : U t_i}
\]

(W-DPRJ)

\[
\frac{\Delta_2 \mid S_2 \Gamma \vdash_w e_2 : \tau_2 \leadsto e'_2 : \tau'_2 \quad \Delta_1 \mid S_1 (S_2 \Gamma, x : \tau_2 \leadsto x' : \tau'_2) \vdash_w e_1 : \tau_1 \leadsto e'_1 : \tau'_1}{S_1 \Delta_2, \Delta_1 \mid S_1 S_2 \Gamma \vdash_w \text{let}^p x = e_2 \text{ in } e_1 : \tau_1 \leadsto \text{let } x' = e'_2 \text{ in } e'_1 : \tau'_1}
\]

(W-DLET)

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 $\Delta \vdash_{w,sr} \tau \leadsto \tau'$, where $\tau$ is an inherited attribute (i.e. input to the algorithm), and $\Delta$ and $\tau'$ 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 $\Delta \mid \Gamma \vdash e : \tau \leadsto e' : \sigma$, then there exist $e'_p$ and $\sigma_p$ satisfying $\Gamma \vdash e : \tau \leadsto e'_p : \sigma_p$ such that for all $\Delta''$, $e''$, $\sigma''$ with $\Delta'' \mid \Gamma \vdash e : \tau \leadsto e'' : \sigma''$ 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 $\vdash_p$ 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:
\[
\begin{align*}
&c \sim^{\text{Id}} c \\
&\hat{n} \sim^{\text{Id}} \hat{n} \\
\text{Int} &\sim^{\text{Id}} \text{Int} \\
\alpha &\sim^{\text{Id}} \alpha \\
\alpha &\not\in \text{FV}(\sigma) \\
\alpha &\sim^{[\alpha/\sigma]} \sigma \\
\alpha &\not\in \text{FV}(\sigma) \\
\sigma &\sim^{[\alpha/\sigma]} \alpha \\
\tau_1' &\sim^T \tau_2' \\
T &\sim^U T' \\
\tau_1'' &\sim^U T'' \\
\tau_1' &\rightarrow \tau_2' \sim^{UT} \tau_1'' \rightarrow \tau_2'' \\
\tau_1'_{11} &\sim^T \tau_2'_{21} \\
T_1 \tau_1'_{12} &\sim^T T_2 \tau_1'_{22} \quad \ldots \quad T_1 \tau_1'_{l_n} &\sim^T T_n T_{n-1} \ldots T_1 \tau_1'_{2n} \\
(\tau_1', \ldots, \tau_l') &\sim^{T_l} (\tau_2', \ldots, \tau_{2n}') \\
\sigma &\sim^{U} \sigma' \\
\text{poly} &\sigma \sim^{U} \text{poly} \sigma' \\
\sigma[a/c] &\sim^{U} \sigma'[a'/c] \\
\forall \alpha. \sigma &\sim^{U} \forall \alpha'. \sigma' \quad (c \text{ fresh}) \\
\delta &\sim^{U} \delta' \\
\rho &\sim^{U} \rho' \\
\delta &\Rightarrow \rho \sim^{U} \delta' \Rightarrow \rho' \\
\tau &\sim^{U} \tau' \\
\text{IsInt} &\tau \sim^{U} \text{IsInt} \tau' \\
\tau &\sim^{T} \tau' \\
T \tau_1 &\sim^U T \tau'_1 \\
UT \tau_2 &\sim^V UT \tau'_2 \\
\tau_1 &:= \tau_1 + \tau_2 \sim^{VUT} \tau_1' := \tau_1' + \tau_2' \\
\sigma_1 &\sim^{T} \sigma_2 \\
T \sigma_1 &\sim^U T \sigma_2' \\
\text{IsMG} \sigma_1 \sigma_1' &\sim^{UT} \text{IsMG} \sigma_2 \sigma_2' \\
\end{align*}
\]

Figure 2.11: Rules for unification.
2.3. Principal Type Specialization

\[
\text{IsInt } t \vdash \text{Int}^s \leftrightarrow t \quad (t \text{ fresh})
\]

\[
\emptyset \vdash \text{Int}^p \leftrightarrow \text{Int}
\]

\[
\Delta_1 \vdash \text{Int}^s \tau_1 \leftrightarrow \tau'_1 \quad \Delta_2 \vdash \text{Int}^s \tau_2 \leftrightarrow \tau'_2
\]

\[
\Delta_1, \Delta_2 \vdash \text{Int}^s \tau_2 \\
\tau_2 \rightarrow^p \tau_1 \leftrightarrow \tau'_2 \rightarrow \tau'_1
\]

\[
\Delta_1, \ldots, \Delta_n \vdash \text{Int}^s (\tau_1, \ldots, \tau_n)^p \leftrightarrow (\tau'_1, \ldots, \tau'_n)
\]

\[
\Delta \vdash \text{Int}^s \tau \leftrightarrow \tau'
\]

\[
\text{IsMG } \sigma \vdash \text{poly } \tau \leftrightarrow \text{poly } s \quad (\sigma = \text{Gen}_g (\Delta \Rightarrow \tau') \text{ and } s \text{ fresh})
\]

Figure 2.12: Rules calculating principal source-residual relationship.

- Booleans as a base type (together with its primitives).
- Static sum-like types, with constructors 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 specialize 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:

\[
\begin{align*}
  e & ::= \text{ | } \text{dcl} \text{\* } e_p \\
  \text{dcl} & ::= \text{data } D^o = cs \\
  cs & ::= C_1^o \, \tau \, \| \, \ldots \, \| \, C_n^o \, \tau \\
  e_p & ::= \ldots \\
  & \quad \mid C^o \\
  & \quad \mid \text{case}^o \, e_p \, \text{of} \, [ \text{br} ]^* \\
  \text{br} & ::= C^o \, x \rightarrow e_p
\end{align*}
\]
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 constructors (the symbol \( \| \) is used to avoid confusion with the symbol \( \| \) used for choice).

Declarations of dynamic sum-types are only allowed at the beginning 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:

\[
\begin{align*}
    e' & ::= \ [ \text{ddcl}' \] * e'_p \\
    \text{ddcl}' & ::= \text{data } D = c's' \\
    c's' & ::= C^1_{c_p} \tau \| \ldots \| C^n_{c_p} \tau \\
    e'_p & ::= \ldots \\
    & \quad | C \\
    & \quad | \text{case } e'_p \text{ of } [ b'r' ]^* \\
    & \quad | \text{proto\text{case}}_{v'} e'_p \text{ with } v'_p \text{ of } [ b'r' ]^* \\
    b'r' & ::= C x \rightarrow e'_p \\
    v'_p & ::= \ldots | \{ C^k \}_{k \in I}
\end{align*}
\]

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 \( c's' \) denotes just an enumeration of constructors. The purpose of the construct \( \text{proto\text{case}}_v \) will be explained 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 \( C^k_D \), 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 referring to, 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 \( D^p \) is considered. For example, if we have the following sum-type declaration

\[
\text{data } D^p = C^p_1 \text{ Int}^s
\]

in the source code

\[
\text{let}^p d_1 = C^p_1 29^s \\
\text{in } \text{let}^p d_2 = C^p_1 71^s \\
\text{in } 4^p
\]

**Chapter 3. Principal Specialization of Dynamic Sum-Types**
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 $D^o$, where $C^1$ can be applied to arguments with types $29$ and $71$ 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

\[
\begin{align*}
\text{data } D_1 &= C_1^1 \forall 29 \\
\text{data } D_2 &= C_2^1 \forall 71 \\
\text{let } d_1 &= C_1^1 \bullet \\
\text{in let } d_2 &= C_2^1 \bullet \\
\text{in } &4
\end{align*}
\]

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 $\tau$, is an element of the language defined by the following grammar:

\[
\begin{align*}
\tau ::= \text{Int}^o \mid \text{Int}^s \mid (\tau_1, \ldots, \tau_n)^o \mid \tau \rightarrow^o \tau \mid \text{poly } \tau \mid D^o
\end{align*}
\]

where the type $(\tau_1, \ldots, \tau_n)^o$ 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 $\tau'$, is an element of the language given by the grammar

\[
\begin{align*}
\tau' ::= t \mid \text{Int} \mid \hat{n} \mid \tau' \rightarrow \tau' \mid (\tau', \ldots, \tau') \mid \text{poly } \sigma \mid D_n \\
\rho ::= \delta \Rightarrow \rho \mid \tau' \\
\sigma ::= s \mid \forall \lambda, \sigma \mid \forall t, \sigma \mid \rho \\
\delta ::= \text{IsInt } \tau' \mid \tau' := \tau' + \tau' \mid \text{IsMG } \sigma \sigma \mid \delta_d
\end{align*}
\]

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.
\[
\Delta \vdash \Gamma_r \vdash_{\text{rr}} e' : \tau' \\
(\Delta \vdash v_k : C^k \in \tau'_e ? \Delta_k, v_{m_k} : C^k \in \tau'_e \text{ HasC } C^k \tau'_e \text{ has } \tau'_e \text{ for } k \in D \land k \neq j) \\
\Delta \vdash v_{m_j} : \text{HasC } C^j \tau'_j, \Gamma_{cs} : \text{ConstrOf } \tau'_e \\
\Delta \vdash \Gamma_r \vdash_{\text{rr}} C_{v_{m_j}}^j e' : \tau'_e
\]

\[
\Delta \vdash \Gamma_r \vdash_{\text{rr}} e' : \tau'_e \\
(h_k : \Delta_k | \Delta_k \vdash_{\text{rr}} \lambda x'_k e'_k : \tau'_k \rightarrow \tau'_e)_{k \in B} \\
(\Delta \vdash v_{m_k} : C^k \in \tau'_e ? \Delta_k, v_k : C^k \in \tau'_e ? \Delta_k)_{k \in B} \\
\Delta \vdash v_{cs} : \text{ConstrOf } \tau'_e \\
\Delta \vdash \Gamma_r \vdash_{\text{rr}} \text{ protocase}_v e' \text{ with } v_{cs} \text{ of } \\
\left( C_{v_{m_k}}^k x'_k \rightarrow e'_k[v_k/h_k] \right)_{k \in B} : \tau'
\]

Figure 3.1: Residual type rules of case and constructor application

**Definition 3.6.** The new predicates, denoted in Definition 3.5 as \( \delta_d \), are defined as follows

\[
\delta_d ::= \text{ConstrsOf } \tau' \\
| \text{HasC } \tau' C^k \tau' \\
| C \in \tau'_e \delta
\]

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-type 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 \( \sigma \) and \( \sigma' \) in their contexts.

**Proposition 3.7.** If \( h : \Delta \vdash \Gamma_r \vdash_{\text{rr}} e' : \sigma \), and \( \Delta' \vdash v : \Delta \), then \( \Delta' \vdash \Delta' \vdash v/h : \sigma \).

**Theorem 3.8.** If \( h : \Delta \vdash \Gamma_r \vdash_{\text{rr}} e' : \sigma \), and \( C : (h : \Delta | \sigma) \geq (h' : \Delta' | \sigma') \), then \( h' : \Delta' \vdash \Gamma_r \vdash \text{C}[e'] : \sigma' \).

### 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 (\( \vdash \)).
3.4. Specializations Rules for Dynamic Sum-Types

\[
\begin{align*}
[HASC] & \quad \frac{\Delta \vdash D_n (C^k) \sim \tau'}{\Delta \vdash n : \text{HasC} \ D_n \ C^k \ \tau'} \\
[CONSTRS-OF] & \quad \frac{D_n = \{ C^k \}_{k \in I}}{\Delta \vdash \{ C^k \}_{k \in I} : \text{ConstrsOf} \ D_n} \\
[GUARD-TRUE] & \quad \frac{C^k \in D_n \quad \Delta \vdash v : \Delta'}{\Delta \vdash v : C^k \in D_n \ ? \ \Delta'} \\
[GUARD-FALSE] & \quad \frac{C^k \notin D_n}{\Delta \vdash \bullet : C^k \in D_n \ ? \ \Delta'} \\
[UNIFY-HASC] & \quad \frac{\Delta, h : \text{HasC} \ \tau' \ \ C^k \ \tau''_1, \ \Delta' \vdash \tau''_2 \sim \tau''_2}{\Delta, h : \text{HasC} \ \tau' \ C^k \ \tau''_1, \ \Delta' \vdash \ h : \text{HasC} \ \tau' \ C^k \ \tau''_2} \\
[HASC-GUARD] & \quad \frac{\Delta, \text{HasC} \ \tau' \ \ C^k \ \tau'', \ \Delta' \vdash v : \Delta''}{\Delta, \text{HasC} \ \tau' \ C^k \ \tau'', \ \Delta' \vdash v : C^k \in \tau' \ ? \ \Delta''} \\
[ENTL-GUARD] & \quad \frac{h : \Delta \vdash v' : \Delta' \quad \Delta'' \vdash v_{cs} : \text{ConstrsOf} \ \tau'}{\Delta'', h : C^k \in \tau' \ ? \ \Delta' \ \vdash \text{if} \ v \ C^k \in v_{cs} \ \text{then} \ v' \ \text{else} \ \bullet : C^k \in \tau' \ ? \ \Delta'} 
\end{align*}
\]

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 \(D^p\). 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 \( h : \Delta \vdash \text{if}_v C^k \in v_{cs} \, \text{then} \, v_2[v_1/h_1] \, \text{else} \, C^k \in \tau' ? \Delta_2 \)

3.4.2 Source-Residual Relationship

The source-residual relationship, expressed by the judgment \( \Delta \vdash_{sr} \tau \hookrightarrow \tau' \), ensures a relation between a source type and a residual one in order to achieve 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 \( C^k \)'s argument in the definition of the sum-type \( D^0 \).

The rule \( (sr-ddt) \), presented in Figure 3.3, establishes the conditions that have to hold in order to \( D^0 \) be related to a residual sum-type \( \tau' \). It establishes that for each residual constructor \( C^k \) belonging to the definition of \( \tau' \), its argument has the residual type \( \tau' \) captured by \( C^k \in \tau' ? \text{HasC} \, \tau' \, C^k \, \tau'_k \). Observe that we also need predicates restricting \( \tau'_k \). We are capturing the possibility 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

\[
\text{data } D^0 = \text{Left}^0 \, \text{Int}^s \mid \text{Right}^0 \, \text{Bool}^s
\]

and the judgement

\[
\vdash_F \lambda e.e : D^0 \rightarrow^p D^0
\]

\( \hookrightarrow \lambda h_1, h_2, h_3, h_4, h_5, \lambda e'.e' : \forall t_1, t_2, t_3, \lambda h_1 : \text{Right} \in t_1? \text{HasC} \, t_1 \, \text{Right} \, t_2,
\]

\( h_2 : \text{Right} \in t_1? \text{IsBool} \, t_2,
\)

\( h_3 : \text{Left} \in t_1? \text{HasC} \, t_1 \, \text{Left} \, t_3,
\)

\( h_4 : \text{Left} \in t_1? \text{IsInt} \, t_3,
\)

\( h_5 : \text{ConstrsOf} \, t_1 \Rightarrow t_1 \rightarrow t_1
\)

If we apply this term to \( (\text{Left}^0 \, 10^s) \), \( \hat{1}0 \) will be the residual type of \( \text{Left} \)'s argument. This last fact can be appreciated in the following judgement

\[
\vdash_F (\lambda e.e) @^p (\text{Left}^0 \, 10^s) : D^0 \rightarrow^p D^0
\]

\( \hookrightarrow \lambda h_1, h_2, h_3, h_4, h_5, h_6, \lambda e'.e' @ (\text{Left}_{h_6} \, \bullet) : \forall t_1, t_2, t_3, h_1 : \text{Right} \in t_1? \text{HasC} \, t_1 \, \text{Right} \, t_2,
\]

\( h_2 : \text{Right} \in t_1? \text{IsBool} \, t_2,
\)

\( h_3 : \text{Right} \in t_1? \text{HasC} \, t_1 \, \text{Right} \, t_3,
\)

\( h_4 : \text{Right} \in t_1? \text{IsBool} \, t_3,
\)

\( h_5 : \text{ConstrsOf} \, t_1,
\)

\( h_6 : \text{HasC} \, t_1 \, \text{Left} \, \hat{1}0 \Rightarrow t_1
\)
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 referring to.

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

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

**Proposition 3.12.** If $\Delta \vdash_{SR} \tau \leftrightarrow \sigma$ and $\Delta' \vdash \Delta$, then $\Delta' \vdash_{SR} \tau \leftrightarrow \sigma$.

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

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 $\tau'_k$ 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 $\eta$-expanded, obtaining functions whose bodies always have applied constructors. After all specialization and post-processing phases have been carried out, $\eta$-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 $\tau'_k$, 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 $\tau'_k$. 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 $\tau'_k$. 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 ● see entailment rule (Guard-False) leaving a meaningless residual branch. Nevertheless,
\[ \Delta \mid \Gamma \vdash e : D(C^j) \quad \leftrightarrow \quad e' : \tau'_j \]

\[ (\Delta \vdash_{sr} D(C^k) \quad \rightarrow \quad \tau'_k)_{k \in D \land k \neq j} \]

\[ (\Delta \vdash v_k : C^k \in \tau'_e \forall \Delta_k, v_{m_k} : C^k \in \tau'_e \forall C^k \in \tau'_e \forall \Delta_k)_{k \in D \land k \neq j} \]

\[ \Delta \vdash v_{m_j} : \text{HasC} \quad \tau'_e \quad C^j \quad \tau'_f, v_{e_k} : \text{ConstrsOf} \quad \tau'_e \]

\[ \Delta \mid \Gamma \vdash C^j_{v_{m_j}} e : D^0 \quad \rightarrow \quad C^j_{v_{m_j}} e' : \tau'_e \]

\[ \Delta \vdash_{sr} \tau \quad \rightarrow \quad \tau' \]

\[ \Delta \mid \Gamma \vdash e : D^0 \quad \rightarrow \quad e' : \tau'_e \]

\[ (h_k : \Delta_k \mid \Gamma \vdash \lambda x_k. e_k : D(C^k) \rightarrow D \tau \quad \rightarrow \quad \lambda x'_k. e'_k : \tau'_k \rightarrow \tau')_{k \in B} \]

\[ (\Delta \vdash v_{m_k} : C^k \in \tau'_e \forall \Delta_k, v_k : C^k \in \tau'_e \forall \Delta_k)_{k \in B} \]

\[ \Delta \vdash v_{e_k} : \text{ConstrsOf} \quad \tau'_e \]

\[ \Delta \mid \Gamma \vdash \text{case } e \text{ of } \]  

\[ (C^j \mid x_k \rightarrow e_k)_{k \in B} \]

\[ : \tau \]

\[ \quad \rightarrow \quad \text{protocase}_v e' \text{ with } v_{e_k} \text{ of } \]

\[ (C^j \mid x'_k \rightarrow e'_k)_{k \in B} \]

\[ : \tau' \]

\[ \text{ protocase}_v e' \text{ with } \{C^j\}_{j \in J \land j \neq \emptyset} \text{ of } \triangleright \text{ case } e' \text{ of } \]

\[ (C^j \mid x'_j \rightarrow e'_j)_{j \in B \cap J} \]

\[ \text{ protocase}_v e' \text{ with } \{\} \text{ of } \triangleright \text{ error } \text{ “There are no branches.”} \]

\[ (C^j \mid x'_k \rightarrow e'_k)_{k \in B} \]

\[ \text{ Figure 3.4: Specialization rules for case and constructor application} \]

\[ \text{ protocase}_v e' \text{ with } \{\} \text{ of } \triangleright \text{ error } \text{ “There are no branches.”} \]

\[ \text{ Figure 3.5: Reduction rule for a protocase}_v \]

the protocase 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 preserved in the transformation are those whose constructors belong to the definition 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 protocase are equal if and only if they can produce the same residual cases 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 \( \Delta \mid \Gamma \vdash e : \tau \quad \rightarrow \quad e' : \sigma \), and for all \( x : \tau_x \rightarrow x' : \tau'_x \in \Gamma \), \( \Delta \vdash_{sr} \tau_x \quad \rightarrow \quad \tau'_x \), then \( \Delta \vdash_{sr} \tau \quad \rightarrow \quad \sigma \).

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

**Theorem 3.15.** If \( \Delta \mid \Gamma \vdash e : \tau \quad \rightarrow \quad e' : \sigma \), then \( \Delta \mid \Gamma_{(\text{hr})} \vdash_{\text{hr}} e' : \sigma \).
\[
\text{protocase}_v e' \text{ with } \{C^k\}_{k \in I} \text{ of } (C_{v_{m_k}}^k x'_k \rightarrow e'_k)_{k \in B} = 
\text{protocase}_v e'' \text{ with } \{C^k\}_{k \in I} \text{ of } (C_{v_{m_k}}^k x''_k \rightarrow e''_k)_{k \in B}
\]

iff

\[
e' = e''
\]

\[
(v_{m_k} = v'_{m_k})_{k \in B \cap I}
\]

\[
(\lambda x'_k.e'_k = \lambda x''_k.e''_k)_{k \in B \cap I}
\]

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

\[
\text{protocase}_v e' \text{ with } h \text{ of } = \text{protocase}_v e'' \text{ with } h \text{ of } (C_{v_{m_k}}^k x'_k \rightarrow e'_k)_{k \in B}
\]

iff for all \(D_n\) with the form \(\{C^k\}_{k \in I}\)

\[
e'[D_n/h] = e''[D_n/h]
\]

\[
(v_{m_k}[D_n/h] = v'_{m_k}[D_n/h])_{k \in B \cap I}
\]

\[
(\lambda x'_k.e'_k[D_n/h] = \lambda x''_k.e''_k[D_n/h])_{k \in B \cap I}
\]

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

Additionally, we also extend the following properties

**Proposition 3.16.** If \(h : \Delta \mid \Gamma \vdash_p e : \tau \leftarrow e' : \tau' \text{ and } h' : \Delta' \vdash v : \Delta, \text{ then } h' : \Delta \mid \Gamma \vdash_p e : \tau \leftarrow e'[h/v] : \tau'.

**Proposition 3.17.** If \(\Delta \mid \Gamma \vdash_p e : \tau \leftarrow e' : \sigma \text{ then } S \Delta \mid S \Gamma \vdash_p e : \tau \leftarrow e' : S \sigma.

**Lemma 3.18.** If \(h : \Delta \mid \Gamma \vdash_p e : \tau \leftarrow e' : \sigma \text{ then } \text{EV}(e') \subseteq h

**Lemma 3.19.** If \(\Delta \mid \Gamma \vdash_p e : \tau \leftarrow e' : \sigma \text{ then there exist } \beta_j, \Delta_\sigma, \text{ and } \tau' \text{ 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 \(46\), and the other on \(t'\) and \(99\).

We have the sum-type declaration

\[
data D^p = \text{Left}^p \text{ Int}^8 \mid \text{Right}^p \text{ Int}^p
\]
in the source code

\[
\text{let } d_1 = \text{if } \text{True} \text{ then Left } 46 \text{ else Right } 40 \\
\text{in let } d_2 = \text{if } \text{False} \text{ then Right } 20 \text{ else Left } 99 \text{ in 4} \\
\text{: Int}\]

and one of its specializations is

\[
\lambda h_1, h_2, h_3, h_4, h_5, h_6. \text{let } d_1 = \text{if } \text{True} \text{ then Left}_{h_1} \bullet \text{ else Right}_{h_2} 40 \\
\text{in let } d_2 = \text{if } \text{False} \text{ then Right}_{h_4} 20 \text{ else Left}_{h_5} \bullet \\
\text{in 4}
\]

with residual type

\[
\forall t, t'. h_1 : \text{HasC } t \text{ Left } 46, \\
h_2 : \text{HasC } t \text{ Right } \text{Int}, \\
h_3 : \text{ConstrsOf } t, \\
h_4 : \text{HasC } t' \text{ Right } \text{Int}, \\
h_5 : \text{HasC } t' \text{ Left } 99, \\
h_6 : \text{ConstrsOf } t' \Rightarrow \text{Int}
\]

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

\[
\text{data } D_1 = \text{Left}_1 46 | \text{Right}_1 \text{Int} \\
\text{data } D_2 = \text{Left}_2 99 | \text{Right}_2 \text{Int}
\]

\[
\text{let } d_1 = \text{if } \text{True} \text{ then Left}_1 \bullet \text{ else Right}_1 40 \\
\text{in let } d_2 = \text{if } \text{False} \text{ then Right}_2 20 \text{ else Left}_2 \bullet \\
\text{in 4}
\]

**Example 3.21.** Here, we force information to flow from a constructor whose argument has residual type \( 75 \) 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

\[
\text{data } D^o = \text{Left}^o \text{Int}^s | \text{Right}^o \text{Int}^o
\]

in the source code

\[
\lambda b. \text{let } d_1 = \text{Left}^o 75^s \\
\text{in let } d_2 = \text{Left}^o 11^s \\
\text{in let } d_3 = \text{Left}^o b \\
\text{in let } u = \text{if } \text{True}^o \text{ then } d_3 \text{ else } d_1 \\
\text{in 4}^o \\
\text{: Int}^s \rightarrow^o \text{Int}^o
\]
and one of its specializations is

\[ \lambda h_1, h_2, h_3, h_4, h_5, h_6. \lambda b. \text{let } d_1 = \text{Left}_{h_4} \bullet \\
    \text{in let } d_2 = \text{Left}_{h_1} \bullet \\
    \text{in let } d_3 = \text{Left}_{h_4} b \\
    \text{in let } u = \text{if } \text{True} \text{ then } d_3 \text{ else } d_1 \\
    \text{in } 4 \]

: \forall t, t'. h_1 : \text{HasC } t \text{ Left } \tilde{1}, \\
    h_2 : \text{Right } \in t'? \text{HasC } t \text{ Right } \text{Int}, \\
    h_3 : \text{ConstrsOf } t, \\
    h_4 : \text{HasC } t' \text{ Left } \tilde{75}, \\
    h_5 : \text{Right } \in t'? \text{HasC } t' \text{ Right } \text{Int}, \\
    h_6 : \text{ConstrsOf } t' \Rightarrow \tilde{75} \Rightarrow \text{Int} \\

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

\[ \text{data } D_1 = \text{Left}_1 \tilde{75} \]
\[ \text{data } D_2 = \text{Left}_2 \tilde{1} \]

\[ \lambda b. \text{let } d_1 = \text{Left}_1 \bullet \\
    \text{in let } d_2 = \text{Left}_2 \bullet \\
    \text{in let } d_3 = \text{Left}_1 b \\
    \text{in let } u = \text{if } \text{True} \text{ then } d_3 \text{ else } d_1 \\
    \text{in } 4 \]

: \tilde{75} \Rightarrow \text{Int}

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

\[ \text{data } D^o = \text{Left}^o \text{ Int}^s | \text{Right}^o \text{ Int}^s \]

and the judgement

\[ \vdash_p \text{Left}^o \tilde{5}^s : D^o \]
\[ \Rightarrow \lambda h_1, h_2, h_3, h_4. \text{Left}_{h_1} \bullet : \forall t, t'. h_1 : \text{HasC } t \text{ Left } \tilde{57}, \\
    h_2 : \text{Right } \in t'? \text{HasC } t \text{ Right } t', \\
    h_3 : \text{Right } \in t'? \text{IsInt } t', \\
    h_4 : \text{ConstrsOf } t \Rightarrow t \]

If the construct \text{Right} is eventually applied to an argument, its type should unify to \text{t'} by application of the entailment rules (\text{HAS-C-GUARD}) and (\text{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

\[ \text{data } D^p = \text{Only}^p \textit{Int}^s \]

and the judgement

\[ \vdash_p \text{if}^p \textit{True} \text{ then } \text{Only}^p \ \tau^s \ \text{else } \text{Only}^p \ 89^s : D^p \]
\[ \iff \lambda h_1, h_2, h_3. \text{if } \textit{True} \text{ then } \text{Only}_{h_1} \bullet \ \text{else } \text{Only}_{h_2} \bullet : \forall t, h_1 : \text{HasC } t \ \text{Only} \ \hat{\tau}, \]
\[ h_2 : \text{HasC } t \ \text{Only} \ 89, \]
\[ h_3 : \text{ConstrsOf } t \]
\[ \Rightarrow t \]

The entailment rule (UNIFY-HASC) says that we need to prove \( \hat{\tau} \sim 89 \) 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 \( 101 \) to successfully solve predicates.

We have the sum-type declaration

\[ \text{data } D^p = \text{Only}^p \textit{Int}^s \]

and the judgement

\[ \vdash_p \lambda x. \text{let}^p f = \text{Only}^p \]
\[ \text{in } (f \ @^p 101^s, f \ @^p x)^p \]
\[ : \textit{Int}^s \rightarrow^p (D^p, D^p)^p \]
\[ \iff \lambda h_1, h_2, h_3, h_4, \lambda x. \text{let } f = \text{Only}_{h_3} \]
\[ \text{in } (f \ @\bullet, f \ @\bullet) \]
\[ : \forall t, t', h_1 : \text{IsInt } t', \]
\[ h_2 : \text{HasC } t \ \text{Only} \ 101, \]
\[ h_3 : \text{HasC } t \ \text{Only} \ t', \]
\[ h_4 : \text{ConstrsOf } t \Rightarrow t' \rightarrow (t, t) \]

**Example 3.25.** All constructors are \( \eta \)-expanded before the specialization and \( \eta \)-reduced after all post-processing phases. We just analyze and show source programs without \( \eta \)-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

\[ \text{data } D^p = \text{Only}^p \textit{Int}^s \]

and the judgement

\[ \vdash_p \text{Only}^p : \textit{Int}^s \rightarrow^p D^p \]
\[ \iff \lambda h_1, h_2, \text{Only}_{h_2} : \forall t, t', h_1 : \text{IsInt } t', \]
\[ h_2 : \text{HasC } t \ \text{Only} \ t', \]
\[ h_3 : \text{ConstrsOf } t \Rightarrow t' \rightarrow t \]
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

\[ \text{data } D^p = \text{Left}^p \text{ Int}^s \mid \text{Right}^p \text{ Bool}^s \]

in the source code

\[
\text{let } c = \text{case}^p (\text{Left}^p 33^s) \text{ of } \\
\quad \text{Left}^p n \to \text{lift } n \\
\quad \text{Right}^p b \to 7^p \\
\text{ in } 4^p \\
: \text{Int}
\]

and one of its specializations is

\[ \forall h_1, h_2, h_3, h_4. \]
\[ \text{let } c = \text{proto} \text{case}^v (\text{Left}_{h_1} \bullet) \text{ with } h_4 \text{ of } \\
\quad \text{Left}_{h_1} n \to 33 \\
\quad \text{Right}_{h_2} n \to 7 \\
\text{ in } 4 \\
\]

with residual type

\[ \forall t, t'. h_1 : \text{HasC } t \text{ Left 33, } \\
\quad h_2 : \text{Right } t? \text{ HasC } t \text{ Right } t', \\
\quad h_3 : \text{Right } t? \text{ IsBool } t', \\
\quad h_4 : \text{ConstrsOf } t \Rightarrow \text{Int} \]

Constraint solving will produce

\[ \text{data } D_1 = \text{Left}_1 33 \]
\[ \text{let } c = \text{case } (\text{Left}_1 \bullet) \text{ of } \\
\quad \text{Left}_1 n \to 33 \\
\text{ in } 4 \]

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

\[ \text{data } D^p = \text{Left}^p \text{ Int}^s \mid \text{Right}^p \text{ Bool}^s \]
and the judgement
\[ \vdash \lambda e.\text{case}^p e \text{ of} \]
\[ \left. \begin{array}{l} \text{Left}^p n \to \text{lift } n \\ \rightarrow \end{array} \right\} \]
\[ \Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7. \]
\[ \lambda e.\text{proto}\text{case}_v e \text{ with } h_7 \text{ of} \]
\[ \left. \begin{array}{l} \text{Left}_{h_5} n \to h_6 \\ \forall t_1, t_2, t_3, t_4, h_1 : \text{Left} \in t_1?\text{HasC } t_1 \text{ Left } t_2, \\ h_2 : \text{Left} \in t_1?\text{IsInt } t_2, \\ h_3 : \text{Right} \in t_1?\text{HasC } t_1 \text{ Right } t_3, \\ h_4 : \text{Right} \in t_1?\text{IsBool } t_3, \\ h_5 : \text{Left} \in t_1?\text{HasC } t_1 \text{ Left } t_4, \\ h_6 : \text{Left} \in t_1?\text{IsInt } t_4, \\ h_7 : \text{ConstrsOf } t_1 \Rightarrow t_1 \Rightarrow \text{Int} \end{array} \right\} \]

The constructor \text{Left} can have an argument of residual type \( \hat{2} \), for instance, by just applying the previous source term to \( (\text{Left}^p \ 2^e) \). In this case we can conclude by \((\text{HASC-GUARD})\) and \((\text{UNIFY-HASC})\) that \( t_2 \) and \( t_4 \) have to be unified to \( \hat{2} \). The entailment rules \((\text{HASC-GUARD})\) and \((\text{UNIFY-HASC})\) 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 \text{Left}'s argument, which is easily achieved, for example, applying the previous source abstraction to the source term \( (\text{Right}^p \ \text{True}^e) \).

**Example 3.28.** Branches are specialized assuming that constructor \( C^k \) has an argument of residual type \( \tau^l_k \). 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
\[ \text{data } D^p = \text{Left}^p \ \text{Int}^e | \text{Right}^p \ \text{Bool}^e \]

in the source code
\[ \lambda e.\text{case}^p e \text{ of} \]
\[ \left. \begin{array}{l} \text{Left}^p n \to \text{let } id = \lambda x.x \\ \in \text{let}^p \text{ force } = (id @^p n, id @^p 44^e)^p \\ \in \text{lift } n \\ \text{Right}^p b \to \text{if } b \text{ then } 10^p \text{ else } 70^p \\ \rightarrow \end{array} \right\} \]
\[ D^p \rightarrow^p \text{Int}^p \]

and one of its specializations is
\[ \Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9. \]
\[ \lambda e.\text{proto}\text{case}_v e \text{ with } h_{12} \text{ of} \]
\[ \left. \begin{array}{l} \text{Left}_{h_5} n \to \text{let force } = (id @n, id @\bullet) \\ \in h_6 \\ \text{Right}_{h_7} b \to \text{if } v \text{ h}_8 \text{ then } 10 \text{ else } 70 \end{array} \right\} \]
with residual type
\[
\forall t_1, t_2, t_3, t_4.
\begin{align*}
  h_1 &: \text{Left} \in t_1 \Rightarrow \text{HasC} t_1 \text{ Left} t_2, \\
  h_2 &: \text{Left} \in t_1 \Rightarrow \text{IsInt} t_2, \\
  h_3 &: \text{Right} \in t_1 \Rightarrow \text{HasC} t_1 \text{ Right} t_3, \\
  h_4 &: \text{Right} \in t_1 \Rightarrow \text{IsBool} t_3, \\
  h_5 &: \text{Left} \in t_1 \Rightarrow \text{HasC} t_1 \text{ Left} 44, \\
  h_6 &: \text{Left} \in t_1 \Rightarrow \text{IsInt} 44, \\
  h_7 &: \text{Right} \in t_1 \Rightarrow \text{HasC} t_1 \text{ Right} t_4, \\
  h_8 &: \text{Right} \in t_1 \Rightarrow \text{IsBool} t_4, \\
  h_9 &: \text{ConstrsOf} t_1 \Rightarrow t_1 \Rightarrow \text{Int}
\end{align*}
\]

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

If we apply the previous function to the code
\[
\vdash_p \text{if} \ ^D \text{False} \ ^D \text{then} \ ^D \text{Right} \ ^S \text{True} \ ^S \text{else} \ ^D \text{Left} \ ^D \ 44 : \ ^D \text{D}
\]

\[
\iff \\
\forall t, h_1 : \text{HasC} t \text{ Right} \ ^D \text{True}, h_2 : \text{HasC} t \text{ Left} 44, h_3 : \text{ConstrsOf} t \Rightarrow t
\]
specialization and constraint solving will be carried out without problems. However, if the code we apply it to is
\[
\vdash_p \text{Left} \ ^D \text{6} : \ ^D \text{D}
\]

\[
\iff \\
\forall t, h_1, h_2, h_3, h_4, \text{Left} h_1 \bullet \\
\forall t, t', h_1 : \text{HasC} t \text{ Left} 6, h_2 : \text{Right} \in t \Rightarrow \text{HasC} t \text{ Right} t', h_3 : \text{Right} \in t \Rightarrow \text{IsBool} t', \\
\forall t, \text{ConstrsOf} t \Rightarrow t
\]
specialization can proceed, but constraint solving will fail because it is impossible to unify 44 with 6 which is forced by the entailment rules \((\text{UNIFY-HASC})\) and \((\text{HASC-GUARD})\).

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

We have the following sum-types declarations
\[
data D_1 \ ^D = \text{Left} \ ^D \text{ Int} \ ^S | \text{Right} \ ^D \text{ Bool} \ ^S
\]
\[
data D_2 \ ^D = \text{Only} \ ^D \text{ Int} \ ^S
\]

used in the source code
\[
\begin{align*}
\text{let} \ ^D f_1 &= \chi e. \text{case} \ ^D e \text{ of} \\
& \quad \text{Left} \ ^D n \rightarrow \text{lift} \ n \ ^D \text{1} \\
& \quad \text{Right} \ ^D n \rightarrow \text{2} \\
\text{in let} \ ^D f_2 &= \chi e. \text{case} \ ^D e \text{ of} \\
& \quad \text{Only} \ ^D n \rightarrow \text{lift} \ n \\
\text{in let} \ ^D g &= f_1 \ ^D \text{1} \ (\text{Left} \ ^D 10 \ ^S) \\
\text{in 4} \\
& : \text{Int} \ ^D
\end{align*}
\]
and one of its specializations is

\[ \Lambda \ h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9, h_{10}, h_{11}, h_{12}, h_{13}. \]

let \( f_1 = \lambda e.\mathsf{protocase}_v e \) with \( h_5 \) of

\[ \begin{align*}
\text{Left}_{h_{11}} \ & n \rightarrow 10 + 1 \\
\text{Right}_{h_3} \ & n \rightarrow 2 
\end{align*} \]

in let \( f_2 = \lambda e.\mathsf{protocase}_v e \) with \( h_{10} \) of

\[ \begin{align*}
\text{Only}_{h_8} \ & n \rightarrow h_9 
\end{align*} \]

in let \( g = f_1@ (\text{Left}_{h_{11}} \bullet) \)

in 4

with residual type

\[ \forall t_1, t_2, t_3, t_4, t_5, t_6, t_7. \]

\[ \begin{align*}
h_1 & : \text{Right} \in t_1?\text{HasC} \ t_1 \ \text{Right} \ t_2, h_2 : \text{Right} \in t_1?\text{Bool} \ t_2, \\
h_3 & : \text{Right} \in t_1?\text{HasC} \ t_1 \ \text{Right} \ t_3, h_4 : \text{Right} \in t_1?\text{Bool} \ t_3, h_5 : \text{ConstrsOf} \ t_1, \\
h_6 & : \text{Only} \in t_4?\text{HasC} \ t_4 \ \text{Only} \ t_5, h_7 : \text{Only} \in t_4?\text{Int} \ t_5, \\
h_8 & : \text{Only} \in t_4?\text{HasC} \ t_4 \ \text{Only} \ t_6, h_9 : \text{Only} \in t_4?\text{Int} \ t_6, h_{10} : \text{ConstrsOf} \ t_4, \\
h_{11} & : \text{HasC} \ t_1 \ \text{Left} \ 10, h_{12} : \text{Right} \in t_1?\text{HasC} \ t_1 \ \text{Right} \ t_7, \\
h_{13} & : \text{Right} \in t_1?\text{Bool} \ t_7 \Rightarrow \text{Int} 
\end{align*} \]

Constraint solving will detect these facts and will proceed deleting the \text{Right} and \text{Only} branches, giving the following residual code

\[ \text{data } D_1 = \text{Left}_1 \ 10 \]

let \( f_1 = \lambda e.\text{case } e \) of

\[ \begin{align*}
\text{Left}_1 \ n \ & \rightarrow 10 + 1 
\end{align*} \]

in let \( f_2 = \lambda e.\text{error} \) "There are no branches."

in let \( g = f_1@ (\text{Left}_1 \bullet) \)

in 4

We have eliminated unneeded branches from \text{cases}, not including \text{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 $\vdash_s$, a syntax directed version of $\vdash_p$, 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 $\vdash_w$ and the proof that $\vdash_s$ is still equivalent to $\vdash_w$ (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 $[\text{dconstr}]$ and $[\text{dcase}]$ respect the equivalence between systems $\vdash_p$ and $\vdash_s$, 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

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

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

The correspondence of this notion of generalization with several applications of the rule $[\text{gen}]$ can be stated as the following property.

**Proposition 4.2.** If $\Delta' \mid \Gamma \vdash_p e : \tau \rightarrow e' : \Delta \Rightarrow \tau'$, then $\Delta' \mid \Gamma \vdash_p e : \tau \rightarrow e' : \text{Gen}_{\Gamma,\Delta'}(\Delta \Rightarrow \tau')$, and both derivations only differ in the application of rule $[\text{gen}]$. 

43
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 \(\Gamma\) and \(\tau'\):

1. if \(h' : \Delta' \vdash v : \Delta\) and \(C = \Gamma(v)\) then \(C : \text{Gen}_{\Gamma}(\Delta \Rightarrow \tau') \geq (h' : \Delta' \vdash \tau')\)

2. if \(h' : \Delta' \vdash v : \Delta\) and \(C = \Lambda h'.\Gamma(v)\) then \(C : \text{Gen}_{\Gamma}(\Delta \Rightarrow \tau') \geq \text{Gen}_{\Gamma}(\Delta' \Rightarrow \tau')\)

3. for all substitutions \(R\) and all contexts \(\Delta\),
\[
 [\Gamma] \text{Gen}_{\Gamma}(\Delta \Rightarrow \tau') \geq \text{Gen}_{\Gamma}(R \Delta \Rightarrow R \tau')
\]

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.

\[
\begin{align*}
\Delta_s & \vdash_{\text{SR}} e : D(C^j) \rightarrow e'_s : \tau'_j \\
(\Delta'_k & \vdash_{\text{SR}} D(C^k) \rightarrow \tau'_k)_{k \in D \land k \neq j} \\
(\Delta_s & \vdash v^k_s : C^k \in \tau'_e, \Delta'_k, v^m_s : C^k \in \tau'_e, \text{HasC} \tau'_e C^k \tau'_k)_{k \in D \land k \neq j} \\
\Delta_s & \vdash v^m_s : \text{HasC} \tau'_e C^j \tau'_j, \text{ConstrsOf} \tau'_e \\
\hline
\Delta_s & \mid \Gamma \vdash_{s} e : C^D \rightarrow C^D_{v^m} e'_s : \tau'_D \\
(\Delta_s & \vdash_{\text{SR}} \tau \rightarrow \tau'_e) \\
\Delta_s & \mid \Gamma \vdash_{s} e : D^{\tau} \rightarrow e'_s : \tau'_e \\
(\Delta_s & \vdash h^k_s : D^k \vdash \lambda x^k_s e^k_s : D(C^k) \rightarrow^\tau \rightarrow \lambda x^k_s e^k_s : \tau'_k, \text{HasC} \tau'_e C^k \tau'_k, v^k_s : C^k \in \tau'_e)_{k \in B} \\
\Delta_s & \vdash v^m_s : \text{HasC} \tau'_e C^j \tau'_j, \text{ConstrsOf} \tau'_e \\
\hline
\Delta_s & \mid \Gamma \vdash_{s} \text{case} e \text{ of} \\
& (C^D_{v^m} x^k_s \rightarrow e^k_s)_{k \in B} \\
& : \tau' \\
& \rightarrow \\
\text{protectCase,} e'_s \text{ with } v^s_{es} \text{ of} \\
& (C^D_{v^m} x^k_s \rightarrow e^k_{s}[v^s_k/h^k_s])_{k \in B} \\
& : \tau'_s
\end{align*}
\]

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 \(h : \Delta \mid \Gamma \vdash_{s} e : \tau \rightarrow e' : \tau'\) then \(S \Delta \mid S \Gamma \vdash_{s} e : \tau \rightarrow e' : S \tau'\)

**Proposition 4.4.** If \(h : \Delta \mid \Gamma \vdash_{s} e : \tau \rightarrow e' : \tau'\) and \(\Delta' \vdash v : \Delta\), then
\[
\Delta' \mid \Gamma \vdash_{s} e : \tau \rightarrow e'[h/v] : \tau'
\]

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 \(\Delta \mid \Gamma \vdash_{s} e : \tau \rightarrow e' : \tau'\) then \(\Delta \mid \Gamma \vdash_{p} e : \tau \rightarrow e' : \tau'\).
4.2. Extension of The Inference Algorithm, W

\[ \frac{\tau \sim^U \tau'}{\text{ConstrsOf } \tau \sim^U \text{ ConstrsOf } \tau'} \]

\[ \frac{\tau_1 \sim^U \tau'_1 \quad U \tau_2 \sim^V U \tau'_2}{\text{HasC } \tau_1 \quad C^k \tau_2 \sim^V U \text{ HasC } \tau'_1 \quad C^k \tau'_2} \]

\[ \frac{\tau \sim^U \tau' \quad U \delta \sim^V U \delta'}{C^k \in \tau?\delta \sim^V U \delta' \quad C^k \in \tau'?\delta'} \]

Figure 4.2: Rules for unification.

**Theorem 4.6.** If \( h : \Delta \mid \Gamma \vdash_p e : \tau \iff e' : \sigma \), then there exist \( h'_s, \Delta'_s, e'_s, \tau'_s \), and \( C'_s \) such that

a) \( h'_s : \Delta'_s \mid \Gamma \vdash e : \tau \iff e'_s : \tau'_s \)

b) \( C'_s : \text{Gen}_P(\Delta'_s \Rightarrow \tau'_s) \geq (h : \Delta \mid \sigma) \)

c) \( C'_s[\Delta h'_s, e'_s] = e' \)

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 \( \vdash_p \) 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 \( \Delta' \mid \Delta \vdash_w \delta \) then \( \Delta', \Delta \vdash \delta \).
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 \( D^o \) see Figure 4.3.

The following propositions are also extended to relate the algorithm \( \vdash_{w,sr} \) with the specification of the \( \vdash_{sr} \) but considering dynamic sum-types.

Proposition 4.10. If \( \Delta \vdash_{w,sr} \tau \iff \tau' \) then \( \Delta \vdash_{sr} \tau \iff \tau' \).

Proposition 4.11. If \( \Delta \vdash_{sr} \tau \iff \sigma \) then \( \Delta'_w \vdash_{w,sr} \tau \iff \tau'_w \) with all the residual variables fresh, and there exists \( C'_w \) such that \( C'_w : \text{Gen}_0(\Delta'_w \Rightarrow \tau'_w) \geq (\Delta \mid \sigma) \).

This last property establishes that the residual type produced by \( \vdash_{w,sr} \) 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 \( \vdash_{w,sr} \) to constraint the type of a lambda-bound variable in rule \( \langle w-dlam \rangle \).

4.2.3 An algorithm for type specialisation

The rules that extend the algorithm \( W \) are shown in Figure 4.4. The rule \( \langle w-dconstr \rangle \) has a similar structure to \( \langle s-dconstr \rangle \) but involves some unifiers and the entailment algorithm. On the other hand, \( \langle w-dcase \rangle \) 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

\[
A_n^m = \begin{cases} 
U_{m-1}^n T_{w_{m-1}} \cdots U_{n-1}^n T_{w_n}, & \text{if } n \leq m \\
\text{Id}, & \text{otherwise}
\end{cases}
\]

in \( \langle w-dcase \rangle \) to express complex compositions of unifiers. Moreover, the hypothesis of the \( \langle w-dcase \rangle \) 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, 1994a], 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 \( h : \Delta \mid S \Gamma \vdash_w e : \tau \iff e' : \tau' \) then \( EV(e') \subseteq h \)

Theorem 4.13. If \( \Delta \mid S \Gamma \vdash_w e : \tau \iff e' : \tau' \) then \( \Delta \mid S \Gamma \vdash_s e : \tau \iff e' : \tau' \).
\[
\Delta_{w_j} \mid T_{w_j}, \Gamma \vdash_w e : D(C^j) \iff e' : \tau'_{w_j}
\]

(\Delta_{w_k} \vdash_{SR} D(C^k) \iff \tau'_{w_k} )_{k \in D \land k \neq j}

\Delta_{w} \mid \Delta_{w_j} \vdash_{w} \begin{array}{c}
(m^{w_j} : C^j \in t?\Delta_{w_k})_{k \in D \land k \neq j}, \\
(m^{w_{vn_k}} : C^k \in t?\text{HasC} t \ C^k \ \tau'_{w_k})_{k \in D \land k \neq j},
\end{array}

h^{w_{vn_k}} : \text{HasC} t \ C^j \ \tau'_{w_j}, h^{w_{vn_k}} : \text{ConstrsOf} t

\Delta_{w}, \Delta_{w_j} \mid T_{w_j}, \Gamma \vdash_w C^j_{\tau'_{w_j}} e : D^\tau \iff C^j_{\tau'_{w_j}} e' : t

\text{(W-DCONSTR)}

\Delta_{w_0} \vdash_{wSR} \begin{array}{c}
\tau \iff \tau'_{w_0}
\end{array}

\Delta_{w_0} \mid T_{w_0}, \Gamma \vdash_w e : D^\tau \iff e' : \tau'_{w_0}

\begin{array}{c}
(m^{w_{k}} : \Delta_{w_k} \vdash_{w} (A^1_{w_k}) \Gamma) \\
\vdash_{w} \lambda x'_{k} e'_{k} : D(C^k) \rightarrow D^\tau \iff \lambda x'_{k} e'_{w_k} : \tau'_{w_k} \rightarrow \tau'_{w_k}
\end{array}

\begin{array}{c}
\tau'_{w_{k}} \sim_{U_1^{B}} \tau'_{w_{k}}
\end{array}

\begin{array}{c}
(t^{w_{k}} \sim_{U_1^{B}} T_{w_k} U_{w_{k-1}}^{-B} \tau'_{w_{k-1}})_{k = 2..B}
\end{array}

\Delta_{w} \mid A_{1}^{B} \Delta_{w_0}, A_{2}^{B} U_{0}^{1} \Delta_{w_0} \vdash_{w}

\begin{array}{c}
(m^{w_{vn_k}} : C^k \in A_{1}^{B} \tau_{w_{k}} \text{?HasC} (A_{1}^{B} \tau'_{w_{k}}) \ C^k \ (A_{k+1}^{B} U_{k-1}^{B} \tau'_{w_{k}}))_{k = 1...B},
\end{array}

\begin{array}{c}
(h^{w_{vn_k}} : C^k \in A_{1}^{B} \tau'_{w_{k}} \text{?A}_{k+1}^{B} U_{k-1}^{B} \Delta_{w_k})_{k = 1...B},
\end{array}

h^{w_{vn_k}} : \text{ConstrsOf} A_{1}^{B} \tau_{w_{k}}

\text{(W-DCASE)}

\Delta_{w}, A_{1}^{B} \Delta_{w_0}, A_{2}^{B} U_{0}^{1} \Delta_{w_0} \mid A_{1}^{B} T_{w_0}, \Gamma

\vdash_{w} \text{case } e \text{ of } \begin{array}{c}
(C^k_{\tau_{w_k}} x_k \rightarrow e_k)_{k = 1...B}
\end{array}

\begin{array}{c}
: \tau
\end{array}

\begin{array}{c}
\rightarrow \text{proto case } e' \text{ with } h^{w}_{w_{vn_k}} \text{ of } \begin{array}{c}
(C^k_{h^{w_{vn_k}}_{w_k}} x'_{k} \rightarrow e'_{w_k}[h^{w_{vn_k}}_{w_k} / h^{w}_{w_{vn_k}}][k = 1...B]
\end{array}
\begin{array}{c}
: U_{B-1}^{\tau_{w_k}}
\end{array}

\end{array}

Figure 4.4: Extension of Type Specialization Algorithm.
Theorem 4.14. If \( h : \Delta \vdash S \Gamma \vdash_e : \tau \rightarrow e' : \tau' \), then \( h' : \Delta' \vdash S \Gamma \vdash_e : \tau \rightarrow e'_w : \tau'_w \) 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 maintain 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

\[
\text{data } D_A^O = \text{Only}_A^O (\text{Int}^s, D_B^O)^D \\
\text{data } D_B^O = \text{Only}_B^O (\text{Bool}^s, D_C^O)^D \\
\text{data } D_C^O = \text{Only}_C^O \text{ Int}^s
\]

in the source code

\[
\text{let } d_1 = \text{Only}_A^O (14^s, \text{Only}_B^O (\text{False}^s, \text{Only}_C^O 59^s)^D)^D \\
\text{in let } d_2 = \text{Only}_B^O (\text{True}^s, \text{Only}_C^O 77^s)^D \\
\text{in let } d_3 = \text{Only}_C^O 22^s \\
\text{in } 4^D
\]

is Int

its principal specialization is

\[
\lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9, h_{10}, h_{11}, h_{12}. \text{let } d_1 = \text{Only}^A_{h_1} (\bullet, \text{Only}^B_{h_3} (\bullet, \text{Only}^C_{h_1} \bullet)) \\
\text{in let } d_2 = \text{Only}^B_{h_7} (\bullet, \text{Only}^C_{h_9} \bullet) \\
\text{in let } d_3 = \text{Only}^C_{h_{11}} \bullet \\
\text{in } 4
\]
with residual type

\[\forall t_1, t_2, t_3, t_4, t_5, t_6,\]
\[h_1 : \text{HasC } t_1 \text{ Only}_A (\hat{1}_4, t_2), h_2 : \text{ConstrsOf } t_1,\]
\[h_3 : \text{HasC } t_2 \text{ Only}_B (\text{False}, t_3), h_4 : \text{ConstrsOf } t_2,\]
\[h_5 : \text{HasC } t_3 \text{ Only}_C \hat{5}, h_6 : \text{ConstrsOf } t_3,\]
\[h_7 : \text{HasC } t_4 \text{ Only}_B (\text{True}, t_5), h_8 : \text{ConstrsOf } t_4,\]
\[h_9 : \text{HasC } t_5 \text{ Only}_C \hat{7}, h_{10} : \text{ConstrsOf } t_5,\]
\[h_{11} : \text{HasC } t_6 \text{ Only}_C \hat{2}, h_{12} : \text{ConstrsOf } t_6 \Rightarrow \text{Int}\]

Constraint solving produces the following declarations

\[
\text{data } D_1 = \text{Only}_1^A(\hat{1}_4, D_2) \\
\text{data } D_2 = \text{Only}_2^B(\text{False}, D_3) \\
\text{data } D_3 = \text{Only}_3^C \hat{5} \\
\text{data } D_4 = \text{Only}_4^B(\text{True}, D_5) \\
\text{data } D_5 = \text{Only}_5^C \hat{7} \\
\text{data } D_6 = \text{Only}_6^C \hat{2}
\]

and the residual code

\[
\text{let } d_1 = \text{Only}_1^A(\bullet, \text{Only}_2^B(\bullet, \text{Only}_3^C \bullet)) \\
\text{in let } d_2 = \text{Only}_4^B(\bullet, \text{Only}_5^C \bullet) \\
\text{in let } d_3 = \text{Only}_6^C \bullet \\
\text{in 4}
\]

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 role of the \texttt{if}\_{\texttt{then}}\_{\texttt{else}} predicate is to defer the decision of which residual type has the Only’s argument until the value of \texttt{b} is known.

We have the dynamic sum-type declaration

\[
data D^D = \text{Only}^D \text{ Int}^S
\]

used in the source code

\[
\lambda^pb.\text{Only}^D (\text{if}^S b \text{ then } 3^S \text{ else } 5^S) : \text{Bool}^S \rightarrow^D D^D
\]

and its principal specialization is

\[
\Lambda h_1, h_2, h_3, h_4, h_5, \lambda b.\text{Only}_{h_1}(\text{if}^v h_2 \text{ then } \bullet \text{ else } \bullet)
\]
with the residual type
\[ \forall t_1, t_2, t_3. \]
\[ h_1 : \text{HasC } t_2 \text{ Only } t_3, \]
\[ h_2 : \text{IsBool } t_1. \]
\[ h_3 : t_3 := \text{if } t_1 \text{ then } 3 \text{ else } 5, \]
\[ h_4 : \text{IsInt } t_3, \]
\[ h_5 : \text{ConstrsOf } t_2 \Rightarrow t_1 \Rightarrow t_2 \]

**Example 4.17.** It is possible to make a polyvariant expression from a constructor if 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 \( D^p \) will be considered.

We have the dynamic sum-type declaration

\[
\text{data } D^p = \text{Only}^p \text{ Int}^s
\]

and the judgement

\[
\vdash_\tau \text{let } f = \text{poly } \text{Only}^p \text{ in } 4^p : \text{Int}^p
\]
\[
\Leftarrow \Lambda h_1. \text{let } f = h_1[\Lambda h, h', h''.\text{Only}_h] \text{ in } 4
\]
\[
: \forall s. \text{IsMG} \ (\forall t, t', h : \text{IsInt } t, h' : \text{HasC } t' \text{ Only } t, h'' : \text{ConstrsOf } t' \Rightarrow t \Rightarrow t') s \Rightarrow \text{Int}
\]

Constraint solving produces

\[
\text{let } f = \Lambda h, h', h''.\text{Only}_h \text{ in } 4
\]

where there is no declaration of residual sum-types; this happens because constructor \( \text{Only} \) 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

\[
\text{data } D^p = \text{Only}^p \text{ Int}^s
\]

used in the source code

\[
\text{let}^p f = \text{poly } \text{Only}^p
\]
\[
\text{in } \text{let}^p d_1 = \text{spec } f @^p 49^s
\]
\[
\text{in } \text{let}^p d_2 = \text{spec } f @^p 62^s
\]
\[
: \text{Int}^p
\]

and its principal specialization is

\[
\Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9.
\]
\[
\text{let } f = h_1[\Lambda h, h', h''.\text{Only}_h]
\]
\[
\text{in } \text{let} d_1 = h_5[f]@\bullet
\]
\[
\text{in } \text{let} d_2 = h_9[f]@\bullet
\]
\[
: \text{Int}^p
\]
with the residual type

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

Constraint solving produces the sum-type declarations

\[
\text{data } D_1 = \text{Only}_1 \hat{62} \\
\text{data } D_2 = \text{Only}_2 \hat{49}
\]

and the residual code

\[
\text{let } f = \lambda h, h', h''.\text{Only}_{h'} \\
\text{in let } d_1 = f((49, 2, \{\text{Only}_1\}))@\bullet \\
\text{in let } d_2 = f((62, 1, \{\text{Only}_2\}))@\bullet \\
\text{in } 4
\]

The first evidence given to \( f \) proves a predicate \text{IsInt} while the second one proves a predicate \text{HasC} indicating the residual instance of \( D^p \) 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 \text{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

\[
\text{data } D^p = \text{Left}^p \text{ Int}^s \mid \text{Right}^p \text{ Int}^s
\]
in the source code

\[
\begin{align*}
\text{let } & f = \text{poly } \text{Left}^p \\
\text{in let } & d_1 = \text{if }^p \text{ True then spec } f @^p 85^s \text{ else Right}^p 15^s \\
\text{in let } & d_2 = \text{spec } f @^p 92^s \\
\text{in let } & c_1 = \text{case}^p d_1 \text{ of } \\
& \quad \text{Left}^p n \rightarrow \text{lift } n \\
& \quad \text{Right}^p n \rightarrow \text{lift } n \\
\text{in let } & c_2 = \text{case}^p d_2 \text{ of } \\
& \quad \text{Left}^p n \rightarrow \text{lift } n \\
& \quad \text{Right}^p n \rightarrow \text{lift } n \\
\text{in 4}^p
\end{align*}
\]

and its principal specialization is

\[
\begin{align*}
\lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9, h_{10}, h_{11}, h_{12}, h_{13}, h_{14}, h_{15}, h_{16}, h_{17}, h_{18}, h_{19}, h_{20}.
\text{let } & f = h_1 [\lambda h_a, h_b, h_c, h_d, h_e. \text{Left}_{h_a}] \\
\text{in let } & d_1 = \text{if } \text{ True then } h_4 [f] @^p \bullet \text{ else Right}_{h_5} \bullet \\
\text{in let } & d_2 = h_{14} [f] @^p \bullet \\
\text{in let } & c_1 = \text{protocase}^p d_1 \text{ with } h_8 \text{ of } \\
& \quad \text{Left}_{h_1}, n \rightarrow h_{16} \\
& \quad \text{Right}_{h_5}, n \rightarrow 15 \\
\text{in let } & c_2 = \text{protocase}^p d_2 \text{ with } h_{13} \text{ of } \\
& \quad \text{Left}_{h_{11}}, n \rightarrow h_{18} \\
& \quad \text{Right}_{h_{19}}, n \rightarrow h_{20} \\
\text{in 4}
\end{align*}
\]

with the residual type

\[
\forall s, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9.
\text{h}_1 : \text{IsMG } (\forall t_1, t_2, t_3, \\
& \quad \text{h}_a : \text{IsInt } t_1, \\
& \quad \text{h}_b : \text{HasC } t_2 \text{ Left } t_1, \\
& \quad \text{h}_c : \text{Right } \in t_2? \text{HasC } t_2 \text{ Right } t_3, \\
& \quad \text{h}_d : \text{Right } \in t_2? \text{HasC } t_2 \text{ Right } \text{IsInt } t_3, \\
& \quad \text{h}_e : \text{ConstrsOf } t_2 \Rightarrow t_1 \Rightarrow t_2) s, \\
\text{h}_2 : \text{Left } \in t_1? \text{HasC } t_1 \text{ Left } t_2, h_3 : \text{Left } \in t_1? \text{IsInt } t_2, \\
\text{h}_4 : \text{IsMG } s (85 \rightarrow t_1), \\
\text{h}_5 : \text{HasC } t_1 \text{ Right } 15^s, \\
\text{h}_6 : \text{Left } \in t_4? \text{HasC } t_1 \text{ Left } t_3, h_7 : \text{Left } \in t_4? \text{IsInt } t_3, \\
\text{h}_8 : \text{ConstrsOf } t_1, \\
\text{h}_9 : \text{Left } \in t_4? \text{HasC } t_4 \text{ Left } t_5, h_{10} : \text{Left } \in t_4? \text{IsInt } t_5, \\
\text{h}_{11} : \text{Right } \in t_4? \text{HasC } t_4 \text{ Right } t_6, h_{12} : \text{Right } \in t_4? \text{IsInt } t_6, \\
\text{h}_{13} : \text{ConstrsOf } t_4, \\
\text{h}_{14} : \text{IsMG } s (92 \rightarrow t_4), \\
\text{h}_{15} : \text{Left } \in t_1? \text{HasC } t_1 \text{ Left } t_7, h_{16} : \text{Left } \in t_1? \text{IsInt } t_7, \\
\text{h}_{17} : \text{Left } \in t_4? \text{HasC } t_4 \text{ Left } t_8, h_{18} : \text{Left } \in t_4? \text{IsInt } t_8, \\
\text{h}_{19} : \text{Right } \in t_4? \text{HasC } t_4 \text{ Right } t_9, h_{20} : \text{Right } \in t_4? \text{IsInt } t_9 \Rightarrow \text{Int}
\end{align*}
\]
4.3. Examples

Constraint solving produces the sum-type declarations

\[
\text{data } D_1 = \text{Left}_1 \ 92 \\
\text{data } D_2 = \text{Left}_2 \ 85 \ | \ \text{Right}_2 \ 15
\]

and the residual code

\[
\text{let } f = \lambda h_a, h_b, h_c, h_d, h_e, \text{Left}_h \\
\text{in let } d_1 = \text{if } \text{True} \text{ then } f((85, 2, 2, 15, \{ \text{Left}, \text{Right} \})) @ \bullet \text{ else } \text{Right}_2 \bullet \\
\text{in let } d_2 = f((92, 1, \bullet, \bullet, \{ \text{Left} \})) @ \bullet \\
\text{in let } c_1 = \text{case } d_1 \text{ of} \\
\ \ \ \ \text{Left}_2 \ n \rightarrow 85 \\
\ \ \ \ \text{Right}_2 \ n \rightarrow 15 \\
\text{in let } c_2 = \text{case } d_2 \text{ of} \\
\ \ \ \ \text{Left}_1 \ n \rightarrow 92 \\
\ \ \ \ \text{in } 4
\]

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

**Example 4.20.** This is an interesting example showing how the unification produced by construct \( \text{if}^P \) is responsible for forcing the constructor \text{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

\[
\text{data } D_1^o = \text{Only}^o \ D_2^o \\
\text{data } D_2^o = \text{Just}^o \ \text{poly } (\text{Int}^s \rightarrow^o \text{Int}^s \rightarrow^o \text{Int}^s)
\]

in the source code

\[
\text{let}^o \ dummy = \text{if}^o \ \text{True} \\
\ \ \ \ \text{then } \text{Only}^o \ (\text{Just}^o \ (\text{poly } (\lambda^o x y. y))) \\
\ \ \ \ \text{else } \text{Only}^o \ (\text{Just}^o \ (\text{poly } (\lambda^o x y. x))) \\
\ \ \ \ \text{in } 4^o \\
\text{: Int}^o
\]

and its principal specialization is

\[
\lambda h_1, h_2, h_3, h_4, h_5, h_6. \\
\text{let } dummy = \text{if } \text{True} \\
\ \ \ \ \text{then } \text{Only}_{h_2} \ (\text{Just}_{h_4} \ (h_1[\lambda h_d, h_e, \lambda x'y', y'])) \\
\ \ \ \ \text{else } \text{Only}_{h_2} \ (\text{Just}_{h_4} \ (h_6[\lambda h_f, h_g, \lambda x'y', x'])) \\
\ \ \ \ \text{in } 4
\]
with residual type

\[ \forall s, t, t'. \]
\[ h_1 : \text{IsMG} \ (\forall t', t', h_d : \text{IsInt} t, h_e : \text{IsInt} t' \Rightarrow t \rightarrow t' \rightarrow t') s, \]
\[ h_2 : \text{HasC} t_1 \ \text{Only} t_2, \]
\[ h_3 : \text{ConstrsOf} t_1, \]
\[ h_4 : \text{HasC} t_2 \ \text{Just} (\text{poly} s), \]
\[ h_5 : \text{ConstrsOf} t_2, \]
\[ h_6 : \text{IsMG} \ (\forall t, t', h_f : \text{IsInt} t, h_g : \text{IsInt} t' \Rightarrow t \rightarrow t' \rightarrow t) s \Rightarrow \text{Int} \]

Constraint solving produces

\[
\text{data } D_1 = \text{Only}_1 \ D_2 \\
\text{data } D_2 = \text{Just}_2 \ \text{poly} \ (\forall t, h : \text{IsInt} t \Rightarrow t \rightarrow t \rightarrow t) \\
\]
\[
\text{let } \text{dummy} = \begin{cases} 
\text{if } \text{True} & \text{then } \text{Only}_1 (\text{Just}_2 (\Lambda h. \lambda x'y'. y')) \\
& \text{else } \text{Only}_1 (\text{Just}_2 (\Lambda h. \lambda x'y'. x')) \\
\end{cases} \\
\text{in } 4 \\
\]

Observe the use of \text{poly} in the declaration of the residual sum-type \(D_2\).

\[\text{EXAMPLE 4.21.}\] 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 \(D^s\). Observe that the residual of \(f\) is the constant

- 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

\[\text{data } D^s = \text{Only}^s \ \text{Int}^s\]

in the source code

\[
\text{let}^s f = \lambda b. \text{if}^s b \ \text{then} \ \text{Only}^s 33^s \ \text{else} \ \text{Only}^s 72^s \\
\text{in let}^s p = (f \ @^s \text{True}^s, f \ @^s \text{False}^s)^s \\
\text{in } 4^s \\
\text{: Int}^s \\
\]

its principal specialization is

\[\Lambda h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9. \\
\text{let } f = \bullet \\
\text{in let } p = (h_5 @ v f @ v \bullet, h_9 @ v f @ v \bullet) \\
\text{in } 4 \]

\[\text{Chapter 4. The Algorithm and the Proof, Extended}\]
with the residual type

\[ \forall t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9. \]
\[
\text{IsFunS clos}(\Delta h_a, h_b, h_c, h_d, h_e, h_f, h_g, h_h, h_i, h_j, h_k, h_l, h_m).
\]
\[
\lambda f'. \lambda h. \text{if } \text{true} \text{ then } \text{Only}_{h_a} \text{ else } \text{Only}_{h_k} \text{ end}.
\]
\[
: \forall t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9.
\]
\[
h_a : \text{IsBool} \ t_1,
\]
\[
h_b : t_2 := \text{if } t_1 \text{ then } t_3 \text{ else } t_4,
\]
\[
h_c : \text{Only} \in t_2? \text{HasC} \ t_2 \text{ Only} \ t_5,
\]
\[
h_d : \text{Only} \in t_2? \text{IsInt} \ t_5,
\]
\[
h_e : \text{ConstrsOf} \ t_2,
\]
\[
h_f : t_1? \text{IsInt} \ t_6,
\]
\[
h_g : t_1? \text{HasC} \ t_7 \text{ Only} \ t_6,
\]
\[
h_h : t_1? \text{ConstrsOf} \ t_7,
\]
\[
h_i : t_1?(t_6 \rightarrow t_7) \sim (33 \rightarrow t_3),
\]
\[
h_j :!t_1? \text{IsInt} \ t_8,
\]
\[
h_k :!t_1? \text{HasC} \ t_9 \text{ Only} \ t_8,
\]
\[
h_l :!t_1? \text{ConstrsOf} \ t_9,
\]
\[
h_m :!t_1?(t_8 \rightarrow t_9) \sim (72 \rightarrow t_4) \Rightarrow t_1 \rightarrow t_2)
\]

Constraint solving produces the sum-type declarations

\[
\text{data} \ D_1 = \text{Only}_1 \ 33
\]
\[
\text{data} \ D_2 = \text{Only}_2 \ 72
\]

used in

\[
\text{let } f = \bullet
\]
\[
\text{in let } p = (\text{Only}_1 \ \bullet, \ \text{Only}_2 \ \bullet)
\]
\[
\text{in } 4
\]
\[
: \text{Int}
\]

Note how the static applications were replaced by their results.

\textbf{Example 4.22.} We have a recursive static function that generates as many applications of the constructor \textit{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.

\[
data D^0 = \text{Only}^0 \text{ Int}^s
\]

\[
\text{let}^D f = \text{fix}^s (\lambda^s g.\lambda^n.\text{if}^s n ==^s 0^s \\
\text{then } 4^D \\
\text{else let}^D d = \text{Only}^D n \\
\text{in } g \, @^s (n -^s 1^s))
\]

\[
\text{in } f \, @^s 5^s
\]

Constraint solving produces

\[
data D_1 = \text{Only}_1 \hat{1}
\]
\[
data D_2 = \text{Only}_2 \hat{2}
\]
\[
data D_3 = \text{Only}_3 \hat{3}
\]
\[
data D_4 = \text{Only}_4 \hat{4}
\]
\[
data D_5 = \text{Only}_5 \hat{5}
\]

\[
\text{let } f = \bullet \\
\text{in let } d = \text{Only}_5 \bullet \\
\text{in let } d = \text{Only}_4 \bullet \\
\text{in let } d = \text{Only}_3 \bullet \\
\text{in let } d = \text{Only}_2 \bullet \\
\text{in let } d = \text{Only}_1 \bullet \\
\text{in } 4
\]
Chapter 5

Extension to the Prototype

“En cambio ahora creo que lo importante es el primer borrador; lo demás es cuestión 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, referential 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 contract 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, 1994a] 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 cases 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 cases 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.
Chapter 7

Simplification and Constraint Solving

The algorithm presented in Chapter 4 and extended in Chapter 2 to produce principal specialisation introduces potentially many predicates, several of which are redundant or expressible in simpler forms. In order to reduce a predicate assignment to another simpler, two phases are defined in [Martínez López and Hughes, 2004]. The first one, called simplification, is responsible for taking a predicate assignment and solves those variables whose solution are unique. This phase is the base for the second one, called constraint solving, which purpose is to take the decisions that were deferred during specialisation when possible (observe that in general, some decisions depend on contextual information that may still not be present).

With this separation, the specialisation can be regarded as a static analysis of the program, performed locally and collecting the restrictions that specify the properties of the final residual program; the constraint solving phase can be viewed as the implementation of the actual calculation of the residual.

Our extension to deal with dynamic sum-types also introduces redundant predicates. For instance, a guarded predicate whose condition is true. Additionally, our algorithm defers the declaration of residual sum-types to be done when all the predicates HasC are collected. Thus, an extension of the simplification and constraint solving phases are need.

We begin, in Section 7.1.1, 7.1.2, 7.1.3, and 7.1.4, by describing the process of simplification as given in [Martínez López and Hughes, 2004]. In Section 7.1.5 we present our extension to this phase. Constraint solving is explained in Section 7.2.1, 7.2.2, and 7.2.3, and our extension is discussed in Section 7.2.4.

7.1 Simplification

7.1.1 Motivation

The algorithm calculating the principal specialisation of an expression introduces several predicates to express the dependencies of subexpressions to static data. But, as this algorithm operates locally, often redundant predicates are introduced. With the goal of reducing the number of predicates, both because of legibility and to lower the computational effort of subsequent phases, we introduce a process of simplification of predicates.
For example, with the algorithm W presented in Figures 2.9 and 2.10, the specialisation of the term

\[ \lambda^x.\text{lift} \left( (x +^s 1^s) +^s (x +^s 1^s) \right) : \text{Int}^s \rightarrow^p \text{Int}^p \]

is the following residual term and type:

\[ \Lambda h_t \ h_a \ h_b \ h_c, \lambda x. h_c : \forall t \ t' \ t''. \ \text{IsInt} \ t, \]
\[ t' := t + \hat{1}, \]
\[ t'' := t + \hat{1}, \]
\[ t''' := t'' + t' \Rightarrow t \rightarrow \text{Int} \]

where the redundancy of predicates can be observed.

By the use of a simplification process, this residual can be converted into this other one:

\[ \Lambda h, h', \lambda x. h' : \forall t, t', t''', t' := t + \hat{1}, t''' := t' + t' \Rightarrow t \rightarrow \text{Int} \]

which, in some sense, is ‘simpler’ than the original, but equivalent.

### 7.1.2 Specification

In order to establish formally the notion of simplification, we will recall the properties we expect of a simplification relation. We will also use a special notation.

**Notation 7.1.** In a simplification we use conversions of the form \((\Lambda h.[])(v))\) and compositions of these. To simplify the reading, we will use a particular notation for this restricted form of conversions (we call them replacements): \(h \leftarrow v\) will be denoting the previous conversion, and the composition of replacements will be written \(h \leftarrow v \cdot C\) to denote \((\Lambda h.C)((v))\) (the operator \(\cdot\) will associate to the right) In this way, \(h_1 \leftarrow v_1 \ldots h_n \leftarrow v_n\) will denote the conversion \((\Lambda h_1 \ldots h_n.[])((v_1 \ldots v_n))\).

The following property of the operator \((\cdot)\) will be very useful:

**Lemma 7.2.** A conversion \(h \leftarrow h\) is neutral for \((\cdot)\) (observe that \([\cdot]\) is a particular case of this.) That is, for every conversion \(C\) and evidence variable \(h\), it holds that \(h \leftarrow h \cdot C = C = C \cdot h \leftarrow h\).

**Proposition 7.3.** \((\Lambda h.e')(\langle h \rangle) = e'\)

Now we are ready to define simplification:

**Definition 7.4.** A relation \(S; C \mid h : \Delta \geq h' : \Delta'\) is a simplification for \(\Delta\) if \(C = h \leftarrow v\) and the following conditions hold:

(i) \(h' : \Delta' \vdash v : S\Delta\)

(ii) \(S\Delta \vdash \Delta'\)
The conditions establish that the predicate assignments are equivalent with respect to entailment (under $S$); as we intend to use this process to replace one predicate assignment with another, it is a natural condition to ask. The condition about the form of conversion $C$ expresses that it can be used to transform an expression assuming predicates in $\Delta$ into another one assuming predicates in $\Delta'$.

Observe that with this definition, $\text{Id;}[]|\Delta \ni \Delta'$ is a valid simplification for $\Delta$. However, we expect that any interesting simplification will be able to do more work, as the following example shows.

**Example 7.5.** Given

$$\Delta_1 = h_1 : \text{IsInt } \hat{9}, h_2 : \text{IsInt } t'''', h_3 : t := \hat{1} + \hat{2}, h_4 : t' := t + \hat{3}, h_5 : t'''' := t'' + t'$$

$$\Delta_2 = h_5 : t'''' := t'' + 6$$

we would like our implementation of $\ni$ to satisfy

$$S; C | \Delta_1 \ni \Delta_2$$

where $S = \text{Id;}[t/\hat{3}][t'/\hat{6}]$ and $C = h_1 \leftarrow 9 \cdot h_2 \leftarrow h_5 \cdot h_3 \leftarrow 3 \cdot h_4 \leftarrow 6$; the reasons for that are:

- $h_1 : \text{IsInt } \hat{9}$ can be trivially simplified by $(\text{IsInt})$, and 9 is its evidence.
- $h_2 : \text{IsInt } t'''$ is entailed by the fifth predicate.
- $h_3 : t := \hat{1} + \hat{2}$ can be simplified calculating the result of $1 + 2$ and generating the substitution that changes $t$ for $\hat{3}$ in the fourth predicate.
- $h_4 : t' := t + \hat{3}$, can be simplified in a similar way, once the value of $t$ is known (from the previous predicate).

To conclude this subsection, we present a closure property of the simplification relation with respect to substitutions. It states that if two predicate assignments are related, the instances of them will be related (provided the substitutions are ‘well behaved’).

**Definition 7.6.** Two substitutions $S$ and $T$ are said to be compatible with respect to a type $\tau$, denoted $S \sim_{\tau} T$, if $TS\tau = ST\tau$. This notion extends naturally to type schemes $\sigma$, predicates $\delta$, and predicate assignments $\Delta$.

**Lemma 7.7.** Let $T; C | \Delta \ni \Delta'$ be a simplification for $\Delta$. If $S$ and $T$ are compatible under $\Delta$, i.e. $S \sim_{\Delta} T$, then $T; C | S\Delta \ni S\Delta'$ is a simplification for $S\Delta$.

This property is important to ensure that sequential steps of an algorithm give a sound solution. This is presented in Section 7.2, where we will combine simplification with constraint solving.


\[
\frac{\Delta \vdash v_\delta \delta}{\Id; h_\delta \leftarrow v_\delta \mid h : \Delta, h_\delta : \delta \geq h : \Delta}
\]

\[
\frac{S; C \mid h : \Delta \geq h' : \Delta' \quad S'; C' \mid h' : \Delta' \geq h'' : \Delta''}{S'S; C' \circ C \mid h : \Delta \geq h'' : \Delta''}
\]

\[
\frac{S; C \mid h_1 : \Delta_1 \geq h_2 : \Delta_2}{S; C \mid h_1 : \Delta_1, h' : \Delta' \geq h_2 : \Delta_2, h' : S \Delta'}
\]

\[
\frac{S; h_1, h_2 \leftarrow v_1, v_2 \mid h_1 : \Delta_1, h_2 : \Delta_2 \geq h'_1, h'_2 : \Delta'_2}{S; h_2, h_1 \leftarrow v_2, v_1[h_2/v_2] \mid h_2 : \Delta_2, h_1 : \Delta_1 \geq h'_2, h'_1 : \Delta'_1}
\]

Figure 7.1: Structural rules for simplification

7.1.3 Implementing a Simplification

Our next step is to define a set of rules implementing a simplification relation.

We start with structural rules, which should be present in any good simplification; they are presented in Figure 7.1. Rule (SimEnt) allows the elimination of redundant predicates; this includes both predicates that are deducible from others, but also those that are true by their form. For example, predicates of the form $\text{IsInt} \, \hat{n}$ for known $\hat{n}$, or predicates $\text{IsMG} \, \sigma \, \sigma'$ for which it can be shown that $C : \sigma \geq \sigma'$. The second rule, (SimTrans), provides transitivity, giving us a way to compose simplifications. The third rule, (SimCtx), expresses how to simplify only part of an assignment; it is important to note the use of the substitution $S$ on the right hand side in order to cancel variables that may have been simplified. Finally, the last rule, (SimPerm), establishes that predicate assignments can be treated as if they had no order, closing the relation with respect to permutations.

The last two rules are complementary, and usually used together. In order to express this, we define a derived rule, (SimCHAM), which allows the application of simplification in any context, following the ideas of the Chemical Abstract Machine [Berry and Boudol, 1990].

\[
\frac{\Delta'_1 \approx \Delta_1 \quad S; C \mid \Delta_1 \geq \Delta_2 \quad \Delta_2 \approx \Delta'_2}{S; C \approx \mid \Delta'_1, \Delta \geq \Delta'_2, S \Delta}
\]

In this last rule ((SimCHAM)), the equivalence $\approx$ is defined as the least congruence on predicate assignments containing $\Delta, \delta, \delta', \Delta' \approx \Delta, \delta', \delta, \Delta'$, allowing assignments to be considered as lists without order for the application of the simplification rules. It is important to note that the order of predicates can be changed only when they are still labelled with evidence variables in a predicate assignment ($h : \delta$); after they are introduced in a type with the (QTN) rule of qualified types theory, the link from the variables to their predicates is only given by the order in which they appear in the expressions ($\Delta h_\vdash \Delta$ in terms and $\delta \Rightarrow \Delta$ in types).

We have to prove that the given structural rules (on Figure 7.1) indeed define a simplification relation according to Definition 7.4.
7.1. Simplification

\[
\begin{align*}
\text{SimOp}_{\text{pes}}: & \quad \forall t, \hat{\eta} \quad t \sim^S \hat{\eta} \\
\text{SimMG}_{\text{t}}: & \quad \forall C : \sigma_2 \geq \sigma_1 \\
\text{Id}; h_2 \leftarrow h_1 \circ C \mid h_1 : \text{IsMG} \quad \sigma_1 s, h_2 : \text{IsMG} \quad \sigma_2 \quad s \sqsupseteq h_1 : \text{IsMG} \quad \sigma_1 s
\end{align*}
\]

Figure 7.2: Language-dependent simplification rules

**Theorem 7.8.** The rules \(\text{SimEnt}_{\text{t}}, \text{SimTrans}_{\text{t}}, \text{SimCtx}_{\text{t}},\) and \(\text{SimPerm}_{\text{t}}\) (of Figure 7.1) define a simplification relation, and the derived rule \(\text{SimCHAM}_{\text{t}}\) is consistent with it.

As a second step in implementing a simplification, we complete the relation defined by the structural rules with those given in Figure 7.2, dealing with some constructs of our language. Rule \(\text{SimOp}_{\text{pes}}\) internalizes the computation of binary operators, when all the operands are known. A similar rule will exist for unary operators as well. The rule \(\text{SimMG}_{\text{t}}\) eliminates redundant uses of predicate IsMG, when two upper bounds of the same variable are comparable. A similar rule for lower bounds would not make sense in this system: as lower bounds are produced by the rules \(\text{SPEC}_{\text{t}}\), they have types instead of schemes; in addition, evidence elimination will need to use all the lower bounds (see Section ??). Regarding predicates as IsInt \(\hat{\eta}\) or IsMG \(\sigma, \sigma'\) when both \(\sigma\) and \(\sigma'\) are not scheme variables, they can be simplified using rule \(\text{SimEnt}_{\text{t}}\), as the entailment relation can deal with them.

Again we have to show that these rules define a simplification relation.

**Theorem 7.9.** A system defining a simplification relation, extended with rules \(\text{SimOp}_{\text{pes}}\) and \(\text{SimMG}_{\text{t}}\) still defines a simplification relation.

Although the rules presented here as an implementation may seem restricted, its goal is to simplify exactly the predicates generated by the specialisation algorithm (not including a rule for lower bounds is an example of this tailoring). When designing a system as this one, the trade off between generality and specificity has to be taken into account - in one end, a very general but useless simplification, and in the other one, a non-tractable or unsolvable simplification would be obtained.

Extensions to the system presented here are possible, and in the case of extending the language of predicates, necessary.

### 7.1.4 Simplification during specialisation

In order to use simplification during the specialisation phase, we need to add a rule to the system \(P\); this rule can be used in any place, but in practice is only needed before the use of a (POLY), or at the end of the derivation.

\[
\text{(SIMP) } \quad \Gamma \mid \Delta \mid h : \Delta \vdash e : \tau \rightarrow e' : \sigma \quad S \mid h : \Delta \triangleright h' : \Delta' \\
\text{h'} : \Delta' \mid S \Gamma \mid e : \tau \rightarrow C[e'] : S \sigma
\]

We can prove that the new rule is consistent with the rest of the system
THEOREM 7.10. If \( h : \Delta \vdash \Gamma \vdash e : \tau \rightarrow e' : \sigma \) and \( S; C \vdash h : \Delta \rightarrow h' : \Delta' \) then \( h' : \Delta' \vdash \Gamma \vdash e : \tau \rightarrow C[e'] : S\sigma \).

As we have seen, the relation of simplification is not necessarily functional, and then there is the possibility to choose among different assignments to replace the current one. In practice, we use a function simplify such that simplify\((h, \Delta)\) returns a triple \( \langle v : \Delta', S, C \rangle \) such that \( S; C \vdash h : \Delta \rightarrow v : \Delta' \). This follows Mark Jones’ work [Jones, 1994a].

Continuing with this idea, we also extend the specialisation algorithm with the following rule:

\[
\frac{h : \Delta \vdash \Gamma \vdash e : \tau \rightarrow e' : \tau'}{h'' : \text{IsMG} \sigma s \mid TS\Gamma \vdash e : \text{poly} \tau \rightarrow e'' : \text{poly} s}
\]

where:

\[
e'' = h''[\Delta h'.C[e']] \\
(h' : \Delta', T, C) = \text{simplify}(h : \Delta) \\
\sigma = TS\Gamma\emptyset(T\Delta' \Rightarrow T\tau') \\
s and h'' \text{ fresh}
\]

With this version of the algorithm, predicate assignments are simplified before their introduction in the type schemes of \text{poly} s. It is important to note, however, that not all predicates can be completely simplified before being introduced in the type schemes: predicates with free variables will remain unsolved, and will not be simplified until constraint solving (although the free variables can take their final value much earlier).

With these rules we have completed our goal of incorporating the simplification to the specialisation process. Additional features and more possible rules will be discussed as new constructs are added to the specialiser.

7.1.5 Extension to Simplification

The rules to simplify the predicates introduced in Section 3.2 are given in Figure 7.3. The rules (shasc) and (scts) extracts the lower index and the constructors used in a residual sum-type declaration. The simplification of guards are given by the rules (sg-true) and (sg-false). This rules require, for technical reasons, a new entailment rule to be added see Figure 7.4. Lastly, the rule (su-hc) forces the arguments’ residual types of two predicates HasC with the same constructors to be the same.

As before, we have to show that these new rules define a simplification relation:

THEOREM 7.11. A system defining a simplification relation, extended with rules (shasc), (scts), (sg-true), (sg-false), and (su-hc) still defines a simplification relation.

The Lemma 7.2, Proposition 7.3, Lemma 7.7, Theorem 7.8, Theorem 7.9, and Theorem 7.10 hold by our extension. They are proved by definitions and properties that were properly extended in Sections 3 and 4 see [Martínez López, 2004] for details in the proofs.
\[
\frac{C^k \in D_n}{\text{Id;}[] \mid h : C^k \in D_n? \Delta \triangleright h : \Delta}
\]

\[
\frac{D_n(C^k) \sim^S \tau'}{S; h \leftarrow n \mid h : \text{HasC } D_n \quad C^k \tau' \triangleright \emptyset}
\]

\[
\frac{D_n = \{ C^k \}_{k \in I}}{\text{Id;} h \leftarrow \{ C^k \}_{k \in I} \mid h : \text{ConstrsOf } D_n \triangleright \emptyset}
\]

\[
\frac{C^k \notin D_n}{\text{Id;} h \leftarrow \bullet \mid h : C^k \in D_n? \Delta \triangleright \emptyset}
\]

\[
\frac{\tau_1'' \sim^S \tau_2''}{S; h_2 \leftarrow h_1 \mid h_1 : \text{HasC } \tau' \quad C^k \quad \tau_1'', \\
\quad h_2 : \text{HasC } \tau' \quad C^k \quad \tau_2'', \\
\quad \triangleright h_1 : \text{HasC } S \tau' \quad C^k \quad S \tau_1''}
\]

**Figure 7.3:** Sum-Types dependent simplification rules

\[
\frac{C^k \in D_n \quad \Delta, \Delta' \vdash \Delta''}{\Delta, C^k \in D_n? \Delta' \vdash \Delta''}
\]

**Figure 7.4:** Additional entailment rule (part I)

\[
\frac{\text{Id;}[] \mid h_1 : \text{HasC } \tau' \quad C^k \quad \tau'', \\
\quad h_2 : C^k \in \tau'? \Delta, \\
\quad \triangleright h_1 : \text{HasC } \tau' \quad C^k \quad \tau'', \quad h_2 : \Delta}{\text{Id;} h \leftarrow v \mid h : \Delta \triangleright h' : \Delta'}
\]

**Figure 7.5:** Additional sum-types simplification rules

\[
\frac{\Delta, \text{HasC } \tau' \quad C^k \quad \tau'', \Delta' \vdash \Delta''}{\Delta, \text{HasC } \tau' \quad C^k \quad \tau'', C^k \in \tau'? \Delta' \vdash \Delta''}
\]

**Figure 7.6:** Additional entailment rule (part II)
Discussion

Although the simplification rules presented previously are enough to simplify predicates related to sum-types, it is possible to perform simplification of predicates inside of guarded predicates. However, we need to guarantee that the substitutions produced do not alter variables that can 'escape' a given guard, because if that guard is going to take a false value, the predicate will simply disappear, and the value assigned to the variable will be unsound. On the other hand, we can also detect before hand that a guard would be true and thus simplifying it.

The rules to perform those kind of simplification are given in Figure 7.5. We need to add another entailment rule to technically prove that (SHC-G) and (SENTL-G) are simplification rules see Figure 7.6. The decision to include or not these rules produces an eager or lazy simplification phase. In other words, a phase that can simplify or not inside of guarded predicates even though values of guards are not yet known.

Again, we have to prove that these two extra rules define a simplification relation:

**Theorem 7.12.** A system defining a simplification relation, extended with rules (SHC-G) and (SENTL-G) still defines a simplification relation.

### 7.2 Constraint Solving

#### 7.2.1 Motivation

In this section, we present the constraint solving phase, a process for deciding the final values of scheme and type variables that cannot be decided by simplification, and thus cannot be performed arbitrarily during specialisation (in the general case, because global information is needed). In [Martínez López, 2004], this phase is separated in two parts: a specification part, where a description of the problem is constructed, and an implementation part, where a solution for the constructed description is found.

Constraint solving is clearly used in the presence of poly and spec annotations, because the specialiser does not decide the final form of polyvariant expressions, but abstracts it with evidence variables (used in every definition point poly and use point spec until all the information can be gathered. A complete description of the procedure to decide the values of scheme variables can be found in [Martínez López, 2004].

The predicates generated by our extension allow the implementation of different heuristics for constraint solving. Some heuristics can produce specific (monomorphic) residual sum-types definitions, while other heuristics can produce general (polymorphic) residual ones. We have implemented the heuristic to produce monomorphic definitions, explained and formalized with detail here, and we are also going to explain how other heuristics may work.

The constraint solving phase is responsible for introducing definitions of residual sum-types based on the predicates found in residual types. It will detect all the predicates of the form HasC $t \cdot C \cdot \tau'$, and will assign to $t$ a value $D_n$ with those constructors appearing in the predicates corresponding to that residual type.

We can see that this approach treats type specialisation as a static program analysis where each part of the program is analyzed locally, and then allows the application of
resolution techniques for the generated constraints. In the field of type specialisation, this approach provides a language allowing to express problems and to look for solutions in a uniform way.

As we have done with simplification, we will first specify the idea of constraint solving, and then we will implement an heuristic for deciding type variables that represent residual sum-types.

### 7.2.2 Specifying Solutions

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

**Definition 7.13.** [Solving] A *solution* from a predicate assignment \(\Delta_1\) to \(\Delta_2\), requiring the predicates of \(\Delta'\) is a relation

\[
S, T; C \mid \Delta_1 + \Delta' \Rightarrow V \Delta_2
\]

where \(S\) and \(T\) are substitutions, \(C\) a conversion and \(V\) a set of type variables, such that

(i) \(T; C \mid S\Delta_1, \Delta' \Rightarrow \Delta_2\)

(ii) \(\text{dom}(S) \cap (V \cup FTV(\Delta')) = \emptyset\)

We will say that \(S\) is the *solution* of the solving, and that \(V\) restricts the application of \(S\).

While solving may appear similar to simplifying at a first glance its consequences are stronger. The substitution \(S\), the solution may decide the values of some scheme variables and it is not required that the new (solved) predicate assignment entails the original one (in contrast to simplifying, where both predicate assignment are equivalent in some sense).

### 7.2.3 Solving and Specialisation

We now study how solving can be performed during specialisation, by incorporating it to system \(P\)

\[
\frac{\Delta_1 \mid \Gamma \vdash e : \tau \leftrightarrow e' : \sigma \quad S, T; C \mid \Delta_1 + \Delta' \Rightarrow_{FTV(\Gamma, \sigma)} \Delta_2}{\Delta_2 \mid T\Gamma \vdash e : \tau \leftrightarrow C[e'] : T\sigma} \quad \text{(SOLV)}
\]

In contrast with the case of simplification, some cautions have to be taken to avoid unsound results: if a variable is decided when some of the information affecting its set of possible values is missing, which can happen if a scheme variable occurs anywhere in the residual type or in the type assignment \(\Gamma\), then it must not be solved. This situation is captured in the rule by the use of the set \(FTV(\Gamma, \sigma)\) in the solving premise (and there used for condition (ii) of Definition 7.13).

As with simplification, the rule \((\text{SOLV})\) is sound respect to specialisation.
Theorem 7.14. Given $\Delta_1 \mid \Gamma \vdash_e \tau \iff e' : \sigma$, and if $S, T; C \mid \Delta_1 + \Delta' \triangleright_{FTV(\Gamma, \sigma)} \Delta_2$ then, it is also the case that

$$\Delta_2 \mid T \Sigma \vdash_e \tau \rightarrow C[e'] : T \sigma.$$  

In order to perform the constraint solving, we proceed incrementally. We justify the incremental nature with the following lemma.

Lemma 7.15. Composition of solutions is a solution.

That is, if $S_2 \sim_{S_1, \Delta_1, \Delta'} T_1, S_1, T_1; C_1 \mid \Delta_1 + \Delta' \triangleright_V \Delta_2$, and $S_2, T_2; C_2 \mid \Delta_2 + \Delta'' \triangleright_V \Delta_3$ then

$$S_2 S_1, T_2 T_1; C_2 \circ C_1 \mid \Delta_1 + (S_2 \Delta', \Delta'') \triangleright_V \Delta_3.$$  

The Lemma 7.15, and Theorem 7.14 holds by our extension since their proofs use definitions and properties already extended in previous chapters see [Martínez López, 2004] for details.

In presence of dynamic recursion, we need a more powerful method. A discussion about this is given in Chapter 9 in [Martínez López, 2004].

7.2.4 Extending the Algorithm for Constraint Solving

We have already defined the notion of resolution. An algorithmic implementation of a heuristic to find, in those cases when it is possible, a resolution for all the predicates in a specialisation judgement was given in [Martínez López, 2004]. However, that algorithm does not find values for those types variables that represent sum-types.

In this section we give an algorithmic implementation of a heuristic to introduce sum-type declaration in such a way that the predicates HasC and $C_k \in t? \Delta$ can be solved. The goal of the presentation is to establish the form in which sum-type declaration are introduced. However, we are not going to give a detailed executable implementation. We present several functions expressed in a pseudo-functional code as in [Martínez López, 2004].

The constraint solving is defined by the function stepSolve, that, given a predicate assignment and a type variable used in a set of HasC predicates to solve, finds a solution for it.

Function stepSolve will be defined in terms of an auxiliary function that introduces monomorphic sum-types polymorphic sum-types can be obtaining by only changing how this function works.

MonomorphicST : it receives a list of predicates of the form $(\text{HasC } t \ C^k \ \tau_k')_{k \in I}$, and introduces a fresh sum-type $D_n$ where each constructor $C^k_n$ is applied to elements of residual type $\tau_k'$.

stepSolve : it is the main step of our constraint solving heuristic. It takes a predicate assignment of the form

$$\Delta = \{h_k : \text{HasC } t \ C^k \ \tau_k'\}_{k \in I}, \Delta_t$$
7.2. Constraint Solving

and a variable $t$ with $t \notin FTV(\Delta_t, \{\tau'_k\}_{k \in I})$, it returns a substitution $S$, a conversion $C$, a predicate assignment $\Delta'$ such that the following resolution holds $S, T; C \mid \Delta + \Delta' \triangleright_V \emptyset$ for any $V$ not containing $t$. It is implemented as follows:

$$\text{stepSolve } t (\{h_k : \text{HasC } t \ C^k \tau'_k\}_{k \in I}, \Delta_t) =$$

$\text{let } D_n = \text{MonomorphicST } \{h_k : \text{HasC } t \ C^k \tau'_k\}_{k \in I}$$

$\text{in } ([D_n/t], \text{Id}, h_k \leftarrow n, \emptyset, \Delta_t)$

**Theorem 7.16.** The heuristic presented is correct wrt. the definition of the constraint solving relation. That is:

1. MonomorphicST finds a solution for $t$, respecting the predicates HasC.

2. If $(S, T, C, \Delta_f, \Delta') = \text{stepSolve } t \ \Delta$ and $s \notin V$ then

$$S, T; C \mid \Delta + \Delta_f \triangleright_V \Delta'.$$

It is important to remark that the algorithm solve, obtained by the repeated composition of solveStep with itself, it is not defined for every predicate assignment. For instance, to solve a scheme variables is necessary to have an assign with upper and lower bound for such variable see [Martínez López, 2004] for details. Our requirement, on the other hand, is to have predicates HasC in the context. Other constructions in the language can establish other requirements for the predicate assignment.
Chapter 8

An Interpreter With Error Handling

# ???

## To do:

* ???

### 8.1 Running Example

Our work is a step forwards towards practical usability of principal type specialisation. In that spirit, we are going to show the specialisation of a simple interpreter for numerical expression with error handle of division by zero.

The interpreter is shown in Figure ???. It is inspired in the interpreter shown [?] to motivate the introduction of a monad to handle errors. Observe that errors are handled dynamically. This decision was based on the fact that exceptions and error-handle mechanism are triggered in runtime.

**Example 8.1.** The following expression completes the definitions given in Figure 8.1.

\[
\text{ueval } @^s (\text{Lam}^s \ 'f',^s \\
\text{(App}^s (\text{Var}^s \ 'f',^s) \\
\text{(App}^s (\text{Var}^s \ 'f',^s) \\
\text{(Const}^s 0^s))))
\]

The specialisation using Hughes’ formulation is

\[
\text{Fun } (\lambda v. \text{case } v \text{ of Fun } f \rightarrow \\
\text{f}@\text{(case } v \text{ of Fun } f \rightarrow \\
\text{f}@\text{(Num 0)))} : \text{Value}
\]

Observe how each function requires a Fun tag, each application requires a case, and each number requires a Num tag.

Since we are compiling by specialisation, the way the residual program is produced indicates that the resulting compilation is for untyped lambda-calculus that is, there are type checks at run-time, and thus the possibility to produce errors during program execution. To illustrate this point, let us consider the code

\[
\text{ueval } @^s (\text{App}^s (\text{Const}^s 2^s) (\text{Const}^s 3^s))
\]

to complete the definitions given in Figure 8.1. The specialisation using Hughes’ formulation is
data $\text{Exp}^s = \text{Var} \cdot \text{Char}^s$

| $\text{Cte} \cdot \text{Int}^s$ |
| $\text{Add} \cdot \text{Exp}^s \cdot \text{Exp}^s$ |
| $\text{Div} \cdot \text{Exp}^s \cdot \text{Exp}^s$ |

data $\text{MaybeInt}^p = \text{Raise} \cdot \text{String}^p \mid \text{Value} \cdot \text{Int}^s$

let $\text{bind} = \lambda^s x. \lambda^s v. \lambda^s \text{env}. \lambda^s y. \text{if}^s x == y \text{ then } v \text{ else } \text{ env} @^s y$

in

let $\text{preeval} =$

fix $^s (\lambda^s \text{eval} \cdot \lambda^s \text{expr} \cdot \lambda^s \text{env}.$

\text{case} $^s \text{expr} \text{ of}$

Var $x$ -> env @^s x

Cte $n$ -> Value^p n

Add $e_1 \ e_2$ -> case^p (eval @^s e_1 @^s env) of

Raise $e$ -> Raise^p $e$

Value $n$ -> case^p (eval @^s e_2 @^s env) of

Raise $e$ -> Raise^p $e$

Value $m$ -> Value^p ($n +^s m$)

Div $e_1 \ e_2$ -> case^p (eval @^s e_1 @^s env) of

Raise $e$ -> Raise^p $e$

Value $n$ -> case^p (eval @^s e_2 @^s env) of

Raise $e$ -> Raise^p $e$

Value $m$ -> if^p (lift $m$ ==^p 0^p) then Raise^p "Div. by zero" else Value^p ($n \div^s m$)

in

let $\text{ueval} =$ preeval @^s (Div^p (Cte^p 1^p))(Add^p (Var^p 0^s)(Cte^p 2^s)))

in ⟨...⟩

Figure 8.1: An evaluator for untyped lambda-calculus
8.1. Running Example

\textbf{case (Num 2) of}
\begin{align*}
\text{Fun } f & \rightarrow f@{\text{(Num 3)}}
\end{align*}

Observe that the \textbf{case} is evaluated in run-time, and when this happens an error will be produced since the number 2 is not a function.
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 masas y que ésta sólo podrá avanzar más rápido si la alentamos con nuestro ejemplo."

La Educación Directa, El Socialismo y el Hombre en Cuba (seminario Marcha, 12 de Marzo de 1965)

Ernesto Che Guevara - La Revolución, Escritos Esenciales
Selección y Prólogo de Marcos Mayer


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 \( h : \Delta | \Gamma_r \vdash e' : \sigma \), and \( \Delta' \vdash v : \Delta \), then \( \Delta' | \Gamma_r \vdash e'[v/h] : \sigma \).

Proof. By induction on the RT derivation.

Extending Proposition 6.11 from [Martínez López, 2004].

Case (rt-dconstr): We know that

\[
\begin{align*}
\Delta | \Gamma_r & \vdash e' : \tau_j' \\
\Delta \vdash v_k : C^k \in \tau_e? \Delta_k, v_m : C^k \in \tau_e?' HasC\tau_e' C^k \tau_k')_{k \in D \land k \neq j} \\
\Delta \vdash v_m : HasC\tau_e' C^j \tau_j', v_e : \text{ConstrsOf } \tau_e' \\
\Delta | \Gamma_r & \vdash C^{j}_{v_m} e' : \tau_e
\end{align*}
\]

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

\[
\Delta' | \Gamma_r \vdash e'[v/h] : \tau_j'
\]  \hspace{1cm} (A.1)

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

\[
\begin{align*}
\Delta' & \vdash (v_k[v/h] : C^k \in \tau_e? \Delta_k, v_m[v/h] : C^k \in \tau_e?' HasC\tau_e' C^k \tau_k')_{k \in D \land k \neq j} \quad (A.2) \\
\Delta' & \vdash v_m[v/h] : HasC\tau_e' C^j \tau_j', v_e[v/h] : \text{ConstrsOf } \tau_e'
\end{align*}
\]

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

\[
\Delta' | \Gamma_r \vdash C^{j}_{v_m}[v/h] e'[v/h] : \tau_e
\]

The result follows from \( C^{j}_{v_m}[v/h] e'[v/h] = (C^{j}_{v_m} e')[v/h] \).
Case (rt-dcase): We know that

\[ \Delta \mid \Gamma_r \vdash_{tr} e' : \tau_e' \]

\[ (h_k : \Delta_k \mid \Gamma_r \vdash_{tr} \lambda x_k' \cdot e_k' : \tau_k' \rightarrow \tau')_{k \in B} \]

\[ (\Delta \vdash v_{m_k} : C^k \in \tau_e'?\text{HasC}\ C^k \tau_k', v_k : C^k \in \tau_e'?\Delta_k)_{k \in B} \]

\[ \Delta \vdash v_{cs} : \text{ConstrsOf}\ \tau_e' \]

\[ \Delta \mid \Gamma_r \vdash_{tr} \text{proto}case_v e' \text{ with } v_{cs} \text{ of } (C^k_{v_{m_k}} x_k' \rightarrow e_k'[v_k/h_k])_{k \in B} \]

\[ : \tau' \]

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

\[ \Delta' \vdash (v_{m_k}[v/h] : C^k \in \tau_e'?\text{HasC}\ C^k \tau_k', v_k[v/h] : C^k \in \tau_e'?\Delta_k)_{k \in B} \]

\[ \Delta' \vdash v_{cs}[v/h] : \text{ConstrsOf}\ \tau_e' \quad (A.3) \]

By IH, we have

\[ \Delta' \mid \Gamma_r \vdash_{tr} e'[v/h] : \tau_e' \quad (A.4) \]

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

\[ \Delta' \mid \Gamma_r \vdash_{tr} \text{proto}case_v e'[v/h] \text{ with } v_{cs}[v/h] \text{ of } : \tau' \]

\[ C^k_{v_{m_k}}[v/h] x_k' \rightarrow e_k'[v_k[v/h]/h_k] \]

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

\[ \text{proto}case_v e'[v/h] \text{ with } v_{cs}[v/h] \text{ of } = \]

\[ (\text{proto}case_v e' \text{ with } v_{cs}, \text{ of } (C^k_{v_{m_k}} x_k' \rightarrow e_k'[v_k/h_k])_{k \in B})_{[v/h]} \]

\[ \text{A.2 Proof of theorem 3.8 from section 3.3} \]

**Theorem 3.8.** If \( h : \Delta \mid \Gamma_r \vdash_{tr} e' : \sigma \), and \( C : (h : \Delta \mid \sigma) \geq (h' : \Delta' \mid \sigma') \), then \( h' : \Delta' \mid \Gamma_r \vdash_{tr} C[e'] : \sigma' \).

**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), let_v, and Proposition 3.7.

\[ \text{A.3 Proof of lemma 3.9 from section 3.4} \]

**Lemma 3.9.** If \( h_1 : \Delta_1 \vdash v_2 : \Delta_2 \) and \( h : \Delta \vdash v_1 : C^k \in \tau_e'?\Delta_1, v_{cs} : \text{ConstrsOf}\ \tau' \) then \( h : \Delta \vdash \text{if}_v C^k \in v_{cs} \text{ then } v_2[v_1/h_1] \text{ else } C^k \in \tau_e'?\Delta_2 \).
Proof: By hypothesis we have that

\[ h_1 : \Delta_1 \vdash v_2 : \Delta_2 \quad (A.5) \]

and

\[ h : \Delta \vdash v_{cs} : \text{ConstrsOf } \tau' \quad (A.6) \]

Now, applying (entl-guard) to A.5 and A.6, we obtain

\[ h : \Delta, h_1 : C^k \in \tau''? \Delta_1 \vdash \text{if } v C^k \in v_{cs} \text{ then } v_2 \text{ else } \bullet : C^k \in \tau''? \Delta_2 \quad (A.7) \]

It is also trivially true that

\[ h : \Delta \vdash h : \Delta, v_1 : C^k \in \tau''? \Delta_1 \quad (A.8) \]

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 \( \Delta \vdash_{SR} \tau \rightarrow \sigma \) then \( S \Delta \vdash_{SR} \tau \rightarrow S \sigma \).

**Proof:** By induction on the SR derivation.

Extending Proposition 6.13 from [Martínez López, 2004].

**Case (sr-ddt):** We know that

\[
\begin{align*}
(\Delta_k \vdash_{SR} D(C^k) & \rightarrow \tau'_k)_{k \in D} \\
(\Delta \vdash C^k \in \tau''? \Delta_k, C^k \in \tau''? \text{HasC } \tau' \ C^k \ \tau'_k)_{k \in D} \\
\Delta \vdash \text{ConstrsOf } \tau' \\
\hline
\Delta \vdash_{SR} D' \rightarrow \tau' 
\end{align*}
\]

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

\[ (S \Delta_k \vdash_{SR} D(C^k) \rightarrow S \tau'_k)_{k \in D} \quad (A.9) \]

and

\[ (S \Delta \vdash S(C^k \in \tau''? \Delta_k), S(C^k \in \tau''? \text{HasC } \tau' \ C^k \ \tau'_k))_{k \in D} \]

\[ S \Delta \vdash S(\text{ConstrsOf } \tau') \]

These last two entailments are equivalent to

\[ (S \Delta \vdash C^k \in S \tau''? S \Delta_k, C^k \in S \tau''? \text{HasC } (S \tau') \ C^k \ (S \tau'_k))_{k \in D} \quad (A.10) \]

Finally, applying (sr-ddt) with A.9 and A.10, we obtain

\[ S \Delta \vdash_{SR} D' \rightarrow S \tau' \]
A.5 Proof of proposition 3.12 from section 3.4

**Proposition 3.12.** If $\Delta \vdash_{sr} \tau \rightarrow \sigma$ and $\Delta' \vdash \Delta$, then $\Delta' \vdash_{sr} \tau \rightarrow \sigma$.

**Proof:** By induction on the SR derivation.
Extending Proposition 6.14 from [Martínez López, 2004].

**Case (sr-ddt):** We know that

$$
\begin{align*}
(\Delta_k \vdash_{sr} D(C^k) & \rightarrow \tau'_k)_{k \in D} \\
(\Delta \vdash C^k \in \tau'? \Delta_k, C^k \in \tau'? \text{HasC } \tau' \ C^k \tau'_k)_{k \in D} \\
\Delta \vdash \text{ConstrsOf } \tau'
\end{align*}
$$

and we obtain

$$
(\Delta' \vdash C^k \in \tau'? \Delta_k, C^k \in \tau'? \text{HasC } \tau' \ C^k \tau'_k)_{k \in D} \tag{A.11}
$$

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

$$
(\Delta \vdash C^k \in \tau'? \Delta_k, C^k \in \tau'? \text{HasC } \tau' \ C^k \tau'_k)_{k \in D} \\
\Delta \vdash \text{ConstrsOf } \tau'
$$

where $\Delta_k$ satisfies

$$
(\Delta_k \vdash_{sr} D(C^k) \rightarrow \tau'_k)_{k \in D} \tag{A.12}
$$

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

$$
\Delta' \vdash_{sr} D^o \rightarrow \tau'
$$

A.6 Proof of theorem 3.13 from section 3.4

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

**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 $\Delta \mid \Gamma \vdash_{\text{P}} e : \tau \rightarrow e' : \sigma$, and for all $x : \tau_x \rightarrow x' : \tau'_x \in \Gamma$, $\Delta \vdash_{sr} \tau_x \rightarrow \tau'_x$, then $\Delta \vdash_{sr} \tau \rightarrow \sigma$.

**Proof:** By induction on the P derivation.
Extending Theorem 6.19 from [Martínez López, 2004].
Case (dConstr): We know that
\[
\Delta \mid \Gamma \vdash \ e : D(C^j) \leadsto e' : \tau'_j
\]
\[
(\Delta_k \vdash_{\text{SR}} D(C^k) \leadsto \tau'_k)_{k \in D \land k \neq j}
\]
\[
(\Delta \vdash v_k : C^k \in \tau'_e \Delta_k, \nu_{m_k} : C^k \in \tau'_e \text{Has} C \tau'_e C^k \tau'_k)_{k \in D \land k \neq j}
\]
\[
\Delta \vdash v_{m_j} : \text{Has} C \tau'_e C^j \tau'_j, \nu_{c} : \text{ConstrOf} \tau'_e
\]
\[
\Delta \mid \Gamma \vdash_{\text{D}} C^j : D^\circ \leadsto C^j_{\text{b}_{m_j}} e' : \tau'_e
\]
By IH we know that
\[
\Delta \vdash_{\text{SR}} D(C^j) \leadsto \tau'_j
\]
and by hypothesis we also know that
\[
(\Delta_k \vdash_{\text{SR}} D(C^k) \leadsto \tau'_k)_{k \in D \land k \neq j}
\]
So, defining \( \Delta_j = \Delta \), we have that
\[
(\Delta_k \vdash_{\text{SR}} D(C^k) \leadsto \tau'_k)_{k \in D}
\]
(A.13)
Now, we have to prove that
\[
(\Delta \vdash C^k \in \tau'_e \Delta_k, C^k \in \tau'_e \text{Has} C \tau'_e C^k \tau'_k)_{k \in D}
\]
\[
\Delta \vdash \text{ConstrOf} \tau'_e
\]
(A.14)
By (Univ), it is enough to prove
\[
(\Delta \vdash C^k \in \tau'_e \Delta_k, C^k \in \tau'_e \text{Has} C \tau'_e C^k \tau'_k)_{k \in D \land k \neq j}
\]
(A.15)
\[
\Delta \vdash C^j \in \tau'_e \Delta, C^j \in \tau'_e \text{Has} C \tau'_e C^j \tau'_j
\]
(A.16)
\[
\Delta \vdash \text{ConstrOf} \tau'_e
\]
(A.17)
Item A.15 holds trivially by hypothesis.
For item A.16 we know that \( \Delta \vdash \text{Has} C \tau'_e C^j \tau'_j \) and so
\[
\Delta \vdash \Delta, \text{Has} C \tau'_e C^j \tau'_j
\]
(A.18)
\[
\Delta, \text{Has} C \tau'_e C^j \tau'_j \vdash \Delta
\]
(A.19)
\[
\Delta, \text{Has} C \tau'_e C^j \tau'_j \vdash \text{Has} C \tau'_e C^j \tau'_j
\]
(A.20)
are trivially true. Applying (HasC-guard) to A.19 and A.20, we obtain
\[
\Delta, \text{Has} C \tau'_e C^j \tau'_j \vdash C^j \in \tau'_e \Delta
\]
(A.21)
\[
\Delta, \text{Has} C \tau'_e C^j \tau'_j \vdash C^j \in \tau'_e \text{Has} C \tau'_e C^j \tau'_j
\]
(A.22)
Then, applying (Univ) to A.21 and A.22, we have
\[
\Delta, \text{Has} C \tau'_e C^j \tau'_j \vdash C^j \in \tau'_e \Delta, C^j \in \tau'_e \text{Has} C \tau'_e C^j \tau'_j
\]
(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 $\Delta \vdash \Gamma \vdash e : \tau \rightarrow e' : \sigma$, then $\Delta \mid_{(\text{RR})} \vdash_{\text{RR}} e' : \sigma$.

**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 $h : \Delta \mid \Gamma \vdash e : \tau \rightarrow e' : \tau'$ and $h' : \Delta' \vdash v : \Delta$, then $h' : \Delta \mid \Gamma \vdash e \rightarrow e'[h/v] : \tau'$

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

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

\[
\begin{align*}
\Delta \mid \Gamma \vdash e : D(C^j) & \rightarrow e' : \tau'_j \\
(\Delta_k \vdash_{\text{SR}} D(C^k) & \rightarrow \tau'_{k})_{k \in D \land k \neq j} \\
(\Delta \vdash v_k : C^k & \in \tau'_e) \Delta_k, v_m_k : C^k \in \tau'_e \text{HasC } \tau'_e \ C^k \ 	au'_k)_{k \in D \land k \neq j} \\
\Delta \vdash v_m_j : \text{HasC } \tau'_e \ C^j \ 	au'_j, v_{cs} : \text{ConstrsOf } \tau'_e \\
\end{align*}
\]

By IH, we have

\[
\Delta' \mid \Gamma \vdash e : D(C^j) \rightarrow e'[v/h] : \tau'_e \quad (A.24)
\]

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

\[
\begin{align*}
(\Delta' \vdash v_k[v/h] : C^k & \in \tau'_e) \Delta_k, v_m_k[v/h] : C^k \in \tau'_e \text{HasC } \tau'_e \ C^k \ 	au'_k)_{k \in D \land k \neq j} \\
\Delta' \vdash v_m_j[v/h] : \text{HasC } \tau'_e \ C^j \ 	au'_j \\
\Delta' \vdash v_{cs}[v/h] : \text{ConstrsOf } \tau'_e \\
\end{align*}
\]

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

\[
\Delta' \mid \Gamma \vdash C^j \ e : D^o \rightarrow C^j_{v_m_j[v/h]} e'[v/h] : \tau'_e \quad (A.26)
\]

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

\[
\Delta \vdash_{SR} \tau \rightarrow \tau' \\
\Delta \mid \Gamma \vdash_p e : D^p \rightarrow e' : \tau_e' \\
(h_k : \Delta_k \mid \Gamma \vdash_p \lambda x_k, e_k : D(C^k) \rightarrow^D \tau \rightarrow \lambda x_k', e_k' : \tau_k' \rightarrow \tau')_{k \in B} \\
(\Delta \vdash v_{m_k} : C^k \in \tau_e') \text{HasC } \tau_e' \space \tau'_k, v_k : C^k \in \tau_e' \Delta_k \mid \Gamma \vdash v_{e_k} : \text{ConstrsOf } \tau_e' \\
\Delta \mid \Gamma \vdash_p \text{ case } e \text{ of} \\
(C_k^D \space x_k \rightarrow e_k)_{k \in B} : \tau \\
\rightarrow \\
\text{ protcase}_v e' \text{ with } v_{e_k} \text{ of} \\
(C_{v_{m_k}}^k \space x'_k \rightarrow e'_k[v_k/h_k])_{k \in B} : \tau'.
\]

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

\[
\Delta' \vdash_{SR} \tau \rightarrow \tau' \\
\Delta' \mid \Gamma \vdash_p e : D^p \rightarrow e'[v/h] : \tau'_e \\
(\Delta' \vdash v_{m_k}[v/h] : C^k \in \tau'_e \text{ HasC } \tau'_e \space \tau'_k, v_k[v/h] : C^k \in \tau'_e \Delta_k)_{k \in B} \\
\Delta' \vdash v_{e_k}[v/h] : \text{ConstrsOf } \tau'_e \\
\Delta' \mid \Gamma \vdash_p \text{ case } e'[v/h] \text{ of} \\
(C_{v_{m_k}[v/h]}^k \space x'_k \rightarrow e'_k[v_k/h_k])_{k \in B} : \tau' \\
\text{ protcase}_v e'[v/h] \text{ with } v_{e_k}[v/h] \text{ of}
\]

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

\[
\text{ protcase}_v e'[v/h] \text{ with } v_{e_k}[v/h] \text{ of} \\
(C_{v_{m_k}[v/h]}^k \space x'_k \rightarrow e'_k[v_k/h_k])_{k \in B} : \tau' \\
\text{ protcase}_v e'[v/h] \text{ with } v_{e_k}[v/h] \text{ of}
\]

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

A.10 Proof of proposition 3.17 from section 3.4

Proposition 3.17. If \( \Delta \mid \Gamma \vdash_p e : \tau \rightarrow e' : \sigma \) then \( S \Delta \mid S \Gamma \vdash_p e : \tau \rightarrow e' : S \sigma \).

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 $h : \Delta \vdash_\varphi e : \sigma \rightsquigarrow e' : \sigma$ then $\text{EV}(e') \subseteq h$

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

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

\[
\begin{align*}
\Delta &\vdash_\varphi e : D(C^j) \rightsquigarrow e' : \tau'_j \\
(\Delta_k &\vdash_\varphi D(C^k) \rightarrow \tau'_{k})_{k \in D \land k \neq j} \\
(\Delta &\vdash v_k : C^k \in \tau'_{e}^j \Delta_k, v_m_j : C^k \in \tau'_{e}^j \text{HasC } \tau'_e C^k \tau'_{e} v_{cs} : \text{ConstrOf } \tau'_e \\
\Delta &\vdash v_m_j : \text{HasC } \tau'_e C^j \tau'_{e} v_{cs} : \text{ConstrOf } \tau'_e \\
\Delta &\vdash_\varphi \text{ConstrOf } \tau'_e
\end{align*}
\]

Let us take $h$ as evidence of $\Delta$. Applying IH we have that $\text{EV}(e') \subseteq h$ and we also know, by entailment, that $\text{EV}(v_m_j) \subseteq h$, so the result follows since $\text{EV}(C_{v_m_j}^j e') = \text{EV}(v_m_j) \cup \text{EV}(e')$.

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

\[
\begin{align*}
\Delta &\vdash_\varphi \tau \rightarrow \tau' \\
\Delta &\vdash_\varphi e : D^o \rightarrow e' : \tau'_e \\
(h_k &\vdash \Delta_k \vdash_\varphi C^k \vdash_\varphi X^x x_k, e_k : D(C^k) \rightarrow D^o \rightarrow \lambda x^x', e'_k : \tau'_k \rightarrow \tau')_{k \in B} \\
(\Delta &\vdash v_m_j : C^k \in \tau'_{e}^j \text{HasC } \tau'_e C^k \tau'_{e} v_k : C^k \in \tau'_{e}^j \Delta_k)_{k \in B} \\
\Delta &\vdash v_{cs} : \text{ConstrOf } \tau'_e \\
\Delta &\vdash_\varphi \text{ConstrOf } \tau'_e
\end{align*}
\]

Let us take $h$ as evidence of $\Delta$. The evidence variables of the residual term produced are

\[
\text{EV}(e') \cup \text{EV}(v_{cs}) \cup (\text{EV}(v_m_j))_{k \in B} \cup (\text{EV}(e'_k[v_k/h_k]))_{k \in B} \tag{A.27}
\]

By IH on all judgements from system P in the hypothesis of the rule we obtain that $\text{EV}(e') \subseteq h$ and $(\text{EV}(e'_k) \subseteq h_k)_{k \in B}$. We have that $(\text{EV}(e'_k[v_k/h_k])) = \text{EV}(v_k))_{k \in B}$, by substitution. Because of entailments of predicates in the hypothesis of the rule, we have that $(\text{EV}(v_k) \subseteq h)_{k \in B}$, $\text{EV}(v_{cs}) \subseteq h$ and $(\text{EV}(v_m_j) \subseteq h)_{k \in B}$. 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 $\Delta \mid \Gamma \vdash_{p} e : \tau \rightarrow e' : \sigma$ then there exist $\beta_{j}, \Delta_{\sigma}$, and $\tau'$ 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 $\Delta' \mid \Gamma \vdash_{p} e : \tau \rightarrow e' : \Delta \Rightarrow \tau'$, then $\Delta' \mid \Gamma \vdash_{p} e : \tau \rightarrow e' : \text{Gen}_{\tau_{\sigma}}(\Delta \Rightarrow \tau')$, 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 $h : \Delta \mid \Gamma \vdash_{s} e : \tau \rightarrow e' : \tau'$ then $h : S \Delta \mid S \Gamma \vdash_{s} e : \tau \rightarrow e' : S \tau'$

*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 $h : \Delta \mid \Gamma \vdash_{s} e : \tau \rightarrow e' : \tau'$ and $\Delta' \vdash v : \Delta$, then

\[\Delta' \mid \Gamma \vdash_{s} e : \tau \rightarrow e'[h/v] : \tau'.\]

*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 $\Delta \mid \Gamma \vdash_{s} e : \tau \rightarrow e' : \tau'$ then $\Delta \mid \Gamma \vdash_{p} e : \tau \rightarrow e' : \tau'$.

*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 \( h : \Delta | \Gamma \vdash e : \tau \rightarrow e' : \sigma, \) then there exist \( h'_s, \Delta'_s, e'_s, \tau'_s, \) and \( C'_s \) such that

a) \( h'_s : \Delta'_s | \Gamma \vdash e : \tau \rightarrow e'_s : \tau'_s \)

b) \( C'_s : \text{Gen}_\Gamma (\Delta'_s \Rightarrow \tau'_s) \geq (h : \Delta | \sigma) \)

c) \( C'_s[h'_s,e'_s] = e' \)

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

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

\[
\begin{align*}
\Delta | \Gamma \vdash e : D(C^j) & \rightarrow e' : \tau'_j \\
(\Delta_k \vdash_{sr} D(C^k) & \rightarrow \tau'_k)_{k \in D \& k \neq j} \\
(\Delta \vdash v_k : C^k \in \tau'_e, v_{m_k} : C^k \in \tau'_e \Rightarrow \text{HasC} \tau'_e C^k \tau'_e \tau'_k)_{k \in D \& k \neq j} \\
\Delta \vdash v_{m_i} : \text{HasC} \tau'_e C^j \tau'_j, v_{cs} : \text{ConstrOf} \tau'_e
\end{align*}
\]

By IH, we have that there exist \( h'_e, \Delta'_e, e'_e, \tau'_e, \) and \( C'_e \) such that

\[
\begin{align*}
\Delta'_e | \Gamma \vdash e : D(C^j) & \rightarrow e'_e : \tau'_j \\
C'_e : \text{Gen}_\Gamma (\Delta'_e \Rightarrow \tau'_e) & \geq (\Delta \vdash \tau'_j) \\
C'_e[h'_e,e'_e] & = e'
\end{align*}
\]

By Definition 2.20 and A.29, we also know that there exist a substitution \( S'_e \) and evidence \( v'_e \) such that

\[
\begin{align*}
\tau'_j & = S'_e \tau'_j \\
\Delta & \vdash v'_e : S'_e \Delta'_e \\
C'_e & = \text{let}_{v} x = \emptyset \text{ in } x(v'_e)
\end{align*}
\]

By hypothesis, we also have that

\[
\begin{align*}
(\Delta_k \vdash_{sr} D(C^k) & \rightarrow \tau'_k)_{k \in D \& k \neq j} \\
(\Delta \vdash v_k : C^k \in \tau'_e, v_{m_k} : C^k \in \tau'_e \Rightarrow \text{HasC} \tau'_e C^k \tau'_e \tau'_k)_{k \in D \& k \neq j} \\
\Delta \vdash v_{cs} : \text{ConstrOf} \tau'_e
\end{align*}
\]

Let us take \( h'_e = (h, h_{v_{m_j}, e'_e}), \Delta'_e = (\Delta, \text{HasC} \tau'_e C^j (S'_e \tau'_j), S'_e \Delta'_e), \tau'_e = \tau'_e \) and \( C'_e = \text{let}_{v} x = \emptyset \text{ in } x((h, h_{v_{m_j}, v'_e})), \) so we need to prove that

a) \( \Delta'_e | \Gamma \vdash_{D_o} C'_j e : D_o \rightarrow C'_{h_{v_{m_j}}} e'_s : \tau'_e \)
\textbf{b)} $C'_s : \text{Gen}_r (\Delta'_s \Rightarrow \tau'_e) \geq (\Delta \mid \tau_e)$ \\
\textbf{c)} $C'_s[A h'_s, C^j_{h_{vm_j}} \ e'_s] = C^j_{h_{vm_j}} \ e'$ \\

For item \textbf{a)} we apply Proposition 4.3 with $S^e$ and A.28, obtaining \\

\[ S^e \Delta^s \mid S^e \Gamma \vdash_e e : D(C^j) \implies e'_s : S^e \tau'_j. \] 

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

\[ S^e \Delta^s \mid \Gamma \vdash_e e : D(C^j) \implies e'_s : \tau'_j. \tag{A.37} \]

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

\[ \Delta'_s \mid \Gamma \vdash e : D(C^j) \implies e'_s : \tau'_j. \tag{A.38} \]

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

\[ \Delta'_s \models h_{vm_j} : \text{HasC} \ \tau'_e \ C^j \ \tau'_j. \tag{A.39} \]

Applying \texttt{(Trans)} to $\Delta'_s \models \Delta$ and A.35 and A.36, we have that \\

\[ (\Delta'_s \mid v_k : C^k \in \tau'_e \Delta_k, \ v_{km_k} : C^k \in \tau'_e \text{HasC} \ \tau'_e \ C^k \ \tau'_k)_{k \in D \wedge k \neq j} \\tag{A.40} \]

\[ \Delta'_s \models v_{ka} : \text{ConstrsOf} \ \tau'_j. \tag{A.41} \]

Finally, applying \texttt{(s-dconstr)} with A.38, A.34, A.40, A.39 and A.41, we obtain the desired property.

For item \textbf{b)} we take the substitution $\text{Id}$ and evidence $v = (h, v_{m_j}, v^s)$. 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 \textbf{A.42} holds trivially by definition of $\text{Id}$.

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

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

\[ C'_s[A h, h_{vm_j}, h^s_e, C^j_{h_{vm_j}} \ e'_s] = (\Delta h, h_{vm_j}, h^s_e, C^j_{h_{vm_j}} \ e'_s)((h, v_{m_j}, v^s_e)) \]

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

\[ C'_s[A h, h_{vm_j}, h^s_e, C^j_{h_{vm_j}} \ e'_s] = C^j_{h_{vm_j}} \ e'_s[v^s_e/h^s_e] \]

The result follows since $e'_s[v^s_e/h^s_e] = e'$ by A.30.
**Case (DCASE):** We know that

\[
\Delta \vdash_{SR} \tau \rightarrow \tau' \\
\Delta \mid \Gamma \vdash_{P} e : D^p \rightarrow e' : \tau'_e \\
(h_k : \Delta_k \mid \Gamma \vdash_{P} x_k.e_k : D(C^k) \rightarrow^p \tau \rightarrow \lambda x'_k.e'_k : \tau'_k \rightarrow \tau')_{k \in B} \\
(\Delta \vdash v_{ms} : C^k \in C \Rightarrow C^k \rightarrow \tau'_e \text{ has } \tau'_e \text{ ConstrOf } \tau'_e) \\
\Delta \vdash v_{cs} : \text{ConstrOf } \tau'_e
\]

\[
\Delta \mid \Gamma \vdash_{P} \text{ case } e \text{ of } \\
(C^k_d x_k \rightarrow e_k)_{k \in B} \mid \tau \\
\rightarrow \\
\text{protocase}_v e' \text{ with } v_{cs} \text{ of } \\
(C^k_{v_{ms}} x'_k \rightarrow e'_k[v_k/h_k])_{k \in B} \mid \tau'
\]

By IH on the second hypothesis of the rule, we have that there exist \( h'_{v_{cs}}', \Delta'_{v_{cs}}', \tau'_{v_{cs}}' \) and \( C_{v_{cs}}' \) such that

\[
\begin{align*}
& h'_{v_{cs}} : \Delta'_{v_{cs}}' \mid \Gamma \vdash_{S} e : D^p \rightarrow e' : \tau'_{v_{cs}}' \\
& C_{v_{cs}}' : \text{Gen}_F(\Delta'_{v_{cs}}' \Rightarrow \tau'_{v_{cs}}' \rightarrow \tau_e') \\
& C_{v_{cs}}'[h'_{v_{cs}}', e'_k] = e'
\end{align*}
\]

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 \( h^k, \Delta^k, \tau^k, \tau^s, \text{ and } C_k \) such that

\[
\begin{align*}
& h^k : \Delta^k \mid \Gamma \vdash_{S} x^k.e^k : D(C^k) \rightarrow^p \tau \rightarrow \lambda x_{k^*}.e_{k^*} : \tau_{k^*} \rightarrow \tau_s \\
& C_k : \text{Gen}_F(\Delta^k \Rightarrow \tau^k \rightarrow \tau_s) \\
& C_k[h^k, \lambda x_{k^*}.e_{k^*}] = \lambda x_{k^*}.e_{k^*}
\end{align*}
\]

By Definition 2.20 applied to A.45, we know that there exist a substitution \( S_{v_{cs}} \) and evidence \( v_{cs} \) such that

\[
\begin{align*}
& \tau'_e = S_{v_{cs}} \tau'_{v_{cs}}' \\
& h : \Delta \vdash v_{cs} : \Delta_{v_{cs}}' \\
& C_{v_{cs}} = \text{let}_v x = [] \text{ in } x((v_{cs}))
\end{align*}
\]

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

\[
\begin{align*}
& \tau'_k \rightarrow \tau' = S_k (\tau'_{k^*} \rightarrow \tau'_{k^*}) \\
& h_k : \Delta_k \vdash v'_k' : S_k \Delta_k \\
& C_k = \text{let}_v x = [] \text{ in } x((v'_k))
\end{align*}
\]
Let us take \( h_1^s = (h, h^s_k, \ldots, h^s_{v_{cs}}), h^s_{v_{cs}} \) and

\[
\Delta'_s = (\Delta, \Delta^s_{v_{cs}}, (C^k \in S^e_k \gamma^e_k. S^e_k \gamma^e_k. S^e_k \Delta^s_{v_{cs}})_{k \in B}, \text{ConstrsOf } S^e_k \gamma^e_k) \quad (A.54)
\]

So, we need to prove that

\[\Delta'_s \vdash \exists_\exists e \text{ case } e \text{ of } \quad (C^k \in S^e_k \gamma^e_k. S^e_k \Delta^s_{v_{cs}}) \quad (A.55)\]

Additionally, by applying Proposition 4.3 to A.44 with \( S^e_k \), A.50, Proposition 4.4 with \( \Delta'_s \vdash S^e_k \Delta^s_{v_{cs}} \) and the fact that \( \text{dom}(S^e_k) \cap \text{FTV}(\Gamma) = \emptyset \), we know that

\[\Delta'_s \vdash \exists_\exists e : D^o \quad (A.56)\]

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

\[\Delta'_s \vdash \exists_\exists \exists \lambda \gamma_k. \exists_\exists e : D(C^k) \rightarrow^p \tau' \quad (A.57)\]

Besides, by definition of \( \Delta'_s \), A.50 and A.52, it is very easy to verify that

\[\Delta'_s \vdash h^s_{v_{cs}} : C^k \in \tau'_e \gamma^e_k \quad (A.58)\]

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

\[v = (h, v_{c^s}, (v_{m_k}, \text{if } C^k \in v_{c^s} \text{ then } v'_k[v_k/h_k] \text{ else } \text{true})_{k \in B}, v_{c^s})\]

We have to prove that

\[\tau' = \text{Id } \tau' \quad (A.59)\]

Item A.59) holds trivially by definition of Id.
For item A.60, it is enough to prove that (by virtue of A.54)

\[ h : \Delta \vdash \Delta \]  \hspace{1cm} (A.61)
\[ h : \Delta \vdash \Delta_{e'} \]  \hspace{1cm} (A.62)
\[ (h : \Delta \vdash C^k \in S_{e'} \tau_{e'}^j \cdot \text{HasC} (S_{e'} \tau_{e'}^j \cdot) C^k (S_k \tau_{k^*})_{k \in B} \]  \hspace{1cm} (A.63)
\[ (h : \Delta \vdash C^k \in S_{e'} \tau_{e'}^j \cdot S_k \Delta_k)_{k \in B} \]  \hspace{1cm} (A.64)
\[ h : \Delta \vdash \text{ConstrOf} (S_{e'} \tau_{e'}^j) \]  \hspace{1cm} (A.65)

Item A.61 holds trivially with evidence \( h \).
Item A.62 holds by A.51 with evidence \( v_{e'} \).
Item A.63 holds by A.50, A.52 and hypothesis with evidence \( v_{m'} \).
For item A.64, it is very easy to verify by hypothesis that

\[ \Delta \vdash v_k : C^k \in \tau_{e'}^j \cdot \Delta _{k}, v_{c's} : \text{ConstrOf} \tau_{e'}^j \]

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

\[ \Delta \vdash \text{if}_v C^k \in v_{c's} \rightarrow v'_k[v_k/h_k] \text{ else } v'_k[v_k/h_k] \rightarrow \text{constrOf } v_{c's} \]

Item A.65 holds by A.50 and hypothesis with evidence \( v_{c's} \).
Finally, by Definition 2.20 we know that

\[ C'_{s} = \text{let}_v x = [] \]
\[ \text{in } (h, v_{e'}, (v_{m'}, \text{if}_v C^k \in v_{c's} \rightarrow v'_k[v_k/h_k] \text{ else } v'_k[v_k/h_k])_{k \in B}, v_{c's}) \]

For item c), by definition of \( C'_{s} \) and \( \Delta'_{s} \), we have that

\[
\begin{align*}
\text{protocase}_v e'_s \text{ with } h_{v_{c's}} \text{ of } C^k_{h_{v_{m'}}, h_{v_{c's}}}, x'_k \rightarrow e'_k[h_{v_{c's}}/h_{v_{m'}}])_{k \in B} & \\
\text{[h, v_{e'}, (v_{m'}, \text{if}_v C^k \in v_{c's} \rightarrow v'_k[v_k/h_k] \text{ else } v'_k[v_k/h_k])_{k \in B}, v_{c's}] & \\
\text{h, v_{e'}, (h_{v_{m'}}, h_{v_{c's}})_{k \in B}, h_{v_{c's}}] & }
\end{align*}
\]

Since \( EV(e'_s) \subseteq h_{v_{e'}}, EV(v_{c's}) \subseteq h, EV(v'_k) \subseteq h_k, EV(v'_k[v_k/h_k]) \subseteq h \) in addition to \( EV(e'_k[h_{v_{c's}}/h_{v_{m'}}]) \subseteq h_{v_{c's}} \) and \( h_{v_{c's}} \neq h \neq h_k \neq h_k \neq h_{v_{c's}}, \) so A.66 can be rewritten as

\[
\begin{align*}
\text{protocase}_v e'_k[v'_k/h_{v_{c's}}] \text{ with } h_{v_{c's}}, [v_{c's}/h_{v_{c's}}] \text{ of } C^k_{h_{v_{m'}}, h_{v_{c's}}}, x'_k \rightarrow e'_k[v_k/h_k] \text{ if } C^k \in v_{c's} \rightarrow v'_k[v_k/h_k] \text{ else } v'_k[v_k/h_k])_{k \in B} & \\
\end{align*}
\]

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

\[
\begin{align*}
\text{protocase}_v e' \text{ with } v_{c's} \text{ of } & \\
(C^k_{v_{m'}}, x'_k \rightarrow e'_k[v_k/h_k])_{k \in B} & \\
\end{align*}
\]

So, we need to prove that

\[
\begin{align*}
\text{protocase}_v e' \text{ with } v_{c's} \text{ of } & \\
(C^k_{v_{m'}}, x'_k \rightarrow e'_k[v_k/h_k])_{k \in B} & \\
\end{align*}
\]

We have to consider two cases accordingly to equality of \text{protocase}_v.
Case \(v_{cs} = \{C^k\}_{k \in I}\) By Definition 3.6, it is enough to prove that

\[
(\lambda x'_k, e'_k [v_k / h_k] = \\
\lambda x'_{k*}, e'_{k*} [\textbf{if}_v C^k \in v_{cs} \textbf{ then } v'_k \textbf{ else } /h_k][v_k / h_k])_{k \in B \cap I}
\]

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

\[
(\lambda x'_k, e'_k [v_k / h_k] = \lambda x'_{k*}, e'_{k*} [v'_k / h_k][v_k / h_k])_{k \in B \cap I}
\]  \hspace{1cm} (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

\[
(\lambda x'_k, e'_k [v_k / h_k][D_n / h'] = \\
\lambda x'_{k*}, e'_{k*} [\textbf{if}_v C^k \in h' \textbf{ then } v'_k \textbf{ else } /h_k][v_k / h_k][D_n / h'])_{k \in B \cap I}
\]

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

\[
(\lambda x'_k, e'_k [v_k / h_k][D_n / h'] = \\
\lambda x'_{k*}, e'_{k*} [\textbf{if}_v C^k \in D_n \textbf{ then } v'_k \textbf{ else } /h_k][v_k / h_k][D_n / h'])_{k \in B \cap I}
\]

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 \(\Delta' \mid \Delta \vdash_w \delta\) then \(\Delta', \Delta \vdash \delta\).

**Proof:** By definition of \(\Delta' \mid \Delta \vdash_w \delta\) and \((\text{Fr})\).
A.21 Proof of proposition 4.10 from section 4.2

Proposition 4.10. If $\Delta \vdash_{w-sr} \tau \iff \tau'$ then $\Delta \vdash_{sr} \tau \iff \tau'$.

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 $\Delta \vdash_{sr} \tau \iff \sigma$ then $\Delta' \vdash_{w-sr} \tau \iff \tau'$ with all the residual variables fresh, and there exists $C'_w$ such that $C'_w : \text{Gen}_0(\Delta'_w \Rightarrow \tau'_w) \geq (\Delta | \sigma)$.

Proof: By induction on the SR derivation.
Extending Proposition 7.12 from [Martínez López, 2004].

Case (sr-ddt): We know that

$$\frac{(\Delta k \vdash_{sr} D(C^k) \iff \tau'_{k})_{k \in D}}{(\Delta \vdash C^k \in \tau'\Delta_k, C^k \in \tau'\text{HasC } \ C^k \ \tau_k)_{k \in D}}$$

$$\frac{(\Delta \vdash \text{ConstrsOf } \tau')}{\Delta \vdash D^o \iff \tau'}$$

By IH we know that there exist $\Delta_{w_k}, \tau_{w_k}$ and $C_{w_k}$ conversions such that

$$(C_{w_k} : \text{Gen}_0(\Delta_{w_k} \Rightarrow \tau_{w_k}) \geq (\Delta_k | \tau_k'))_{k \in D}$$

which means that there exist substitutions $(S_k)_{k \in D}$ and evidence $(v_k)_{k \in D}$ such that

$$(\tau'_k = S_k \tau'_{w_k})_{k \in D}$$

$$\frac{(h_k : \Delta_k \vdash v_k : S_k \Delta_{w_k})_{k \in D}}{(A.69)}$$

$$\frac{(A.70)}{v^sr_k : C^k \in \tau'\Delta_k)_{k \in D}}$$

$$(A.71)$$

$$\frac{(A.72)}{\Delta \vdash v^sr_m : C^k \in \tau'\text{HasC } \ C^k \ \tau_k')_{k \in D}}$$

We define $\tau'_w = t$, for $t$ fresh,

$$\Delta'_w = ((C^k \in t?\Delta_{w_k}, C^k \in t?\text{HasC } t \ C^k \ \tau'_w)_{k \in D}, \text{ConstrsOf } t)$$

(A.73)
and the substitution

\[
S(t') = \begin{cases} 
\tau' & \text{if } t' = t \\
S_k(t') & \text{if } t' \in \text{dom}(S_k)
\end{cases}
\]

Let us also define evidence

\[
v = ((\textbf{if } v. C^k \in \upsilon^s_{cs} \textbf{ then } v_k[v_k^s/h_k] \textbf{ else } \bullet, \upsilon^s_{ms})_{k \in D}, \upsilon^s_{cs})
\]

Now, we have to prove that

(a) \( \tau' = S t \)

(b) \( \Delta \vdash v : (S (C^k \in t?\Delta_w), S (C^k \in t?\text{HasC } t \ C^k \tau'_w))_{k \in D}, S (\text{ConstrsOf } t) \)

in order to establish the existence of the desired conversion.

Item (a) holds trivially by definition of \( S \).

For item (b), distribute \( S \) and by its definition together with A.69, we obtain that

\[
\Delta \vdash C^k \in \tau'?S_k \Delta_w, C^k \in \tau'?\text{HasC } \tau' \ C^k \tau'_w, \text{ConstrsOf } \tau'
\]

By (Univ), it is enough to prove that

\[
\Delta \vdash C^k \in \tau'?S_k \Delta_w \quad (A.74)
\]

\[
\Delta \vdash C^k \in \tau'?\text{HasC } \tau' \ C^k \tau'_w \quad (A.75)
\]

\[
\Delta \vdash \text{ConstrsOf } \tau' \quad (A.76)
\]

Item (A.74) holds by applying Lemma 3.9 to A.70, A.71 and A.73, obtaining

\[
(\Delta \vdash \textbf{if } v. C^k \in \upsilon^s_{cs} \textbf{ then } v_k[v_k^s/h_k] \textbf{ else } \bullet : C^k \in \tau'?S_k \Delta_w)_{k \in D}
\]

Item (A.75) holds trivially by A.72 with evidence \( v^s_{ms} \).

Item (A.76) holds trivially by A.73 with evidence \( v^s_{cs} \).

### A.23 Proof of lemma 4.12 from section 4.2

**Lemma 4.12.** If \( h : \Delta \vdash_w e : \tau \rightarrow e' : \tau' \) 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 $\Delta \mid S \vdash_w e : \tau \Rightarrow e' : \tau'$ then $\Delta \mid S \vdash_e e : \tau \Rightarrow e' : \tau'$.

**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 $h : \Delta \mid S \vdash_w e : \tau \Rightarrow e' : \tau'$, then $h'_w : \Delta'_w \mid T'_w \vdash_w e : \tau \Rightarrow e'_w : \tau'_w$ 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

$$
\Delta_s \mid S \vdash_s e : D(C^j) \Rightarrow e'_s : \tau'_j^s
$$

$$(\Delta_s \vdash_s D(C^k) \Rightarrow \tau'_k^s)_{k \in D \land k \neq j}$$

$$(\Delta_s \vdash v^s_k : C^k \in \tau'_e \wedge \Delta^s_k, v^s_{m_j} : C^k \in \tau'_e_j \wedge \text{HasC} \tau'_e \wedge C^k \tau'_{k_j}^s, \text{ConstrOf} \tau'_e, \tau'_j^s, \text{ConstrOf} \tau'_e, \tau'_j^s)_{k \in D \land k \neq j}$$


$$
\Delta_s \mid S \vdash_s C_j^o e : D^o \Rightarrow C_j^o v^s_{m_j} e'_s : \tau'_j^s
$$

By IH on the first hypothesis of the rule, we have that

$$h_{w_j} : \Delta_{w_j} \mid T_{w_j} \vdash_w e : D(C^j) \Rightarrow e'_{w_j} : \tau'_{w_j}$$

and there exist a substitution $R^{th}$ and evidence $v^{th}$ such that
\begin{align*}
S &\approx R^{IH} T_{w_j} \\
\tau'_{w_j} &= R^{IH} \tau^t_{w_j} \\
h_k : \Delta^s_k \vdash v_{IH} : R^{IH} \Delta_{w_j} \\
e' &= e'_{w_j}[v_{IH}/h_{w_j}] \\
\end{align*}

By Proposition 4.11 on the second hypothesis of the rule, we also know that there exist conversions \( \{ C_{w_k} \}_{k \in D \land k \neq j} \) such that

\[
(C_{w_k} : \text{Gen}(\Delta_{w_k} \Rightarrow \tau^t_{w_k}) \geq (\Delta^s_k \mid \tau^t_{k^*})_{k \in D \land k \neq j})
\]  

(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

\[
\begin{align*}
(\tau^t_{k^*} &= S_k \tau^t_{w_k})_{k \in D \land k \neq j} \\
(h^t_k : \Delta^s_k \vdash v_k : S_k \Delta_{w_k})_{k \in D \land k \neq j}
\end{align*}
\]  

(A.82)

(A.83)

Without losing generality, we assume that the sets \( \text{dom}(R^{IH}), \text{dom}(S_k)_{k \in D \land k \neq j} \) and \( \{ t \} \) are disjoint (if that is not the case, \( \alpha \)-conversions can be performed).

By application of \( \vdash_w \), we obtain \( \Delta_w \) such that

\[
\Delta_w = (((C_k \in t?\Delta_{w_k}, C_k \in t?\text{HasC} t C^k \tau^t_{w_k})_{k \in D \land k \neq j}, \\
\text{HasC} t C^j \tau^t_{w_j}, \\
\text{ConstrsOf}(t))
\]  

(A.84)

Let us take \( h'_w = ((h^w_{v_k}, h^w_{v_{m_k}})_{k \in D \land k \neq j}, h^w_{v_{m_j}}, h^w_{v_w}, h_w), \Delta'_w = (\Delta_w, \Delta_{w_j}), T'_w = T_{w_j}, \tau'_w = t, \)

\[
R(t') = \begin{cases} 
\tau^t_{v_k} & \text{if } t' = t \\
S_k(t') & \text{if } t' \in \text{dom}(S_k) \\
R^{IH}(t') & \text{otherwise}
\end{cases}
\]

and \( v'_w = (\text{if } C_k \in v^t_{v_k} \text{ then } v_k[v^t_k/h^t_k] \text{ else } \bullet, v^t_{m_k})_{k \in D \land k \neq j}, v^t_{m_j}, v^t_{v_k}, v_{IH}). \) Now, we have to prove that

a) \( S \approx RT_{w_j} \)

b) \( \tau^t_{e} = R t \)

c) \( \Delta^s \vdash v'_w : R(\Delta_w, \Delta_{w_j}) \)

d) \( C^j_{v_{m_j}} e' = (C^j_{v_{m_j}} e'_{w_j})[v'_w/h_w, h_{w_j}] \)

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 \( \text{dom}(S_k) \) can appear in \( T_{w_j} \).
Item b) holds trivially by definition of \(R\).

For item c), by A.84 we have to prove

\[
\Delta_s \vdash R \left( [C^k \in t? \Delta_{w_{k,j}}, C^k \in t? \text{HasC} \ t \ C^k \ \tau'_{w_{k,j}}, \ \text{HasC} \ t \ C^j \ \tau'_{w_{k,j}}, \ \text{ConstrsOf} \ t, \ \Delta_{w_{j}}] \right)
\]

This last entailment is equivalent to

\[
\Delta_s \vdash (C^k \in \tau'_{e_s} \ ? S_k \Delta_{w_{k,j}}, C^k \in \tau'_{e_s} \ ? \text{HasC} \ \tau'_{e_s} \ C^k \ (S_k \tau'_{w_{k,j}}))_{k \in D \land k \neq j},
\]

\[
\text{HasC} \ (R) \ C^j \ (R\tau'_{w_{j}}),
\]

\[
\text{ConstrsOf} \ (R),
\]

\[
R\Delta_{w_{j}}
\]

which can be rewritten as follows

\[
\Delta_s \vdash (C^k \in \tau'_{e_s} \ ? S_k \Delta_{w_{k,j}}, C^k \in \tau'_{e_s} \ ? \text{HasC} \ \tau'_{e_s} \ C^k \ (S_k \tau'_{w_{k,j}}))_{k \in D \land k \neq j},
\]

\[
\text{HasC} \ \tau'_{e_s} \ C^j \ (R^{IH} \tau'_{w_{j}}),
\]

\[
\text{ConstrsOf} \ \tau'_{e_s},
\]

\[
R^{IH} \Delta_{w_{j}}
\]

By A.78 and A.82, we have that

\[
\Delta_s \vdash (C^k \in \tau'_{e_s} \ ? S_k \Delta_{w_{k,j}}, C^k \in \tau'_{e_s} \ ? \text{HasC} \ \tau'_{e_s} \ C^k \ \tau'_{k*})_{k \in D \land k \neq j},
\]

\[
\text{HasC} \ \tau'_{e_s} \ C^j \ \tau'_{j*},
\]

\[
\text{ConstrsOf} \ \tau'_{e_s},
\]

\[
R^{IH} \Delta_{w_{j}}
\]

In order to prove this entailment, it is enough to prove

\[
(D_s \vdash C^k \in \tau'_{e_s} \ ? S_k \Delta_{w_{k,j}})_{k \in D \land k \neq j} \quad \text{(A.85)}
\]

\[
(D_s \vdash C^k \in \tau'_{e_s} \ ? \text{HasC} \ \tau'_{e_s} \ C^k \ \tau'_{k*})_{k \in D \land k \neq j} \quad \text{(A.86)}
\]

\[
\Delta_s \vdash \text{HasC} \ \tau'_{e_s} \ C^j \ \tau'_{j*} \quad \text{(A.87)}
\]

\[
\Delta_s \vdash \text{ConstrsOf} \ \tau'_{e_s} \quad \text{(A.88)}
\]

\[
\Delta_s \vdash R^{IH} \Delta_{w_{j}} \quad \text{(A.89)}
\]

A.85) can be proved using the third and fourth hypothesis, that is

\[
(D_s \vdash v^s_t : C^k \in \tau'_{e_s} \ ? \Delta_{w_{k,j}})_{k \in D \land k \neq j} \quad \text{(A.90)}
\]

\[
\Delta_s \vdash v^s_{e_s} : \text{ConstrsOf} \ \tau'_{e_s} \quad \text{(A.91)}
\]

Thus we can apply Lemma 3.9 with A.83, A.90 and A.91, proving

\[
(D_s \vdash \text{if}^s \ C^k \in v^s_{e_s} \ \text{then} \ v_k[v^s_t/h^s_{k}] \ \text{else} \ C^k \in \tau'_{e_s} \ ? S_k \Delta_{w_{k,j}})_{k \in D \land k \neq j} \quad \text{A.86}
\]

holds by third hypothesis with evidence \(v^s_{m_{i,k}}\).

A.87) holds by fourth hypothesis with evidence \(v^s_{m_{i,j}}\).

A.88) holds by fourth hypothesis with evidence \(v^s_{e_s}\).
A.89) holds by A.79 with evidence \(v_{IH}\).

Item d)

\[
C_{v_{1m_j}}^{j} e' = (C_{h_{w_{m_j}}}^{j} e_{w_j}' [v'_{w}/h_w, h_{w_j}])
\]

\[
= (C_{h_{w_{m_j}}}^{j} e_{w_j}' [(\text{if } v \in u_{cs}^{k} \text{ then } v_k [v^s_k / h_j] \text{ else } \bullet, u_m^s, u_{cs}^s, v^s_{IH} / h_w, h_{w_j}])
\]

\[
= (C_{h_{w_{m_j}}}^{j} e_{w_j}' [(\text{if } v \in u_{cs}^{k} \text{ then } v_k [v^s_k / h_j] \text{ else } \bullet, u_m^s, u_{cs}^s, v^s_{IH} / h_w, h_{w_j}]) \quad (A.92)
\]

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

\[
C_{v_{1m_j}}^{j} e' = (C_{h_{w_{m_j}}}^{j} e_{w_j}' [v^s_{m_j}, v_{IH} / h_{w_{m_j}}, h_{w_j}])
\]

which is the same that

\[
C_{v_{1m_j}}^{j} e' = C_{h_{w_{m_j}}}^{j} [v^s_{m_j} / h_{w_{m_j}}] e_{w_j}' [v_{IH} / h_{w_j}]
\]

Applying substitution and A.80, we obtain

\[
C_{v_{1m_j}}^{j} e' = C_{v_{1m_j}}^{j} e'
\]
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):

\[
\Delta_s \vdash_S \tau \leftarrow \tau_s'
\]

\[
\Delta_s \mid S \Gamma \vdash_e : D^\rho \leftarrow e_1' : \tau_{e_1}'
\]

\[
h^s_1 : \Delta_1^s \mid S \Gamma \vdash x_1.e_1 : D(C^1) \rightarrow D^\tau \leftarrow \lambda x_1'.e_1' : \tau_{e_1}' \rightarrow \tau_s'
\]

\[
\Delta_s \vdash v^s_{m_1} : C^1 \in \tau_{e_1}'; \text{HasC} \tau_{e_1}' \text{ C}^1 \tau_{e_1}', v_1^s : C^1 \in \tau_{e_1}'; \Delta_1^s,
\]

\[
v_{m_1}^s : \text{ConstrsOf} \tau_{e_1}'
\]

\[
\Delta_s \mid S \Gamma \vdash_S \text{case } e \text{ of } C_1^D \ x_1 \rightarrow e_1
\]

\[
: \tau
\]

\[
\rightarrow \text{proc case } e_1' \text{ with } v^s_{e_1} \text{ of } C_{v^s_{m_1}}^1 \ x'_1 \rightarrow e'_1[v^s_{1}/h^s_1]
\]

\[
: \tau_s'
\]
By the outer IH, we obtain a substitution $R_e$ and evidences $v_e$ such that
\[
\begin{align*}
h_{w_e} : \Delta_{w_e} & \mid T_{w_e} \Gamma \vdash_w e : D^q & \iff & e'_{w_e} : \tau'_{w_e} \tag{A.93} \\
S & \approx R_e T_{w_e} & \tag{A.94} \\
\tau'_s & = R_e \tau'_{w_e} \tag{A.95} \\
\Delta_s \vdash v_e : R_e \Delta_{w_e} & \tag{A.96} \\
e'_s & = e'_{w_e}[v_e/h_{w_e}] \tag{A.97}
\end{align*}
\]

By the third hypothesis of the rule and A.94, we have that
\[
\begin{align*}
h^*_1 : \Delta^*_1 & \mid R_e (T_{w_e} \Gamma) \vdash \lambda^0 x_1.e_1 : D(C^1) \rightarrow^p \tau \iff \lambda x'_1.e'_1 : \tau'_1 \rightarrow \tau'_s \tag{A.98}
\end{align*}
\]
Applying the outer IH on A.98, we obtain a substitution $R_1$ and evidence $v_1$ such that
\[
\begin{align*}
h^w_{w_1} : \Delta_{w_1} & \mid T_{w_1} (T_{w_e} \Gamma) \vdash_w \lambda^0 x_1.e_1 : D(C^1) \rightarrow^p \tau \iff \lambda x'_1.e'_1 : \tau'_1 \rightarrow \tau'_{w_1} \tag{A.99} \\
R_e & \approx R_1 T_{w_1} \tag{A.100} \\
\tau'_s & = R_1 \tau'_{w_1} \tag{A.101} \\
\tau'_s & = R_1 \tau'_{w_1} \tag{A.102} \\
h^*_1 : \Delta^*_1 & \vdash v_1 : R_1 \Delta_{w_1} \tag{A.103} \\
\lambda x'_1.e'_1 = (\lambda x'_1.e'_1)[v_1/h^w_{w_1}] \tag{A.104}
\end{align*}
\]
By Proposition 4.11, there exist a substitution $R_{sr}$ and evidence $v_{sr}$ such that
\[
\begin{align*}
\Delta_{w_0} & \vdash_{W-SR} \tau \iff \tau'_{w_0} \tag{A.105} \\
\tau'_s & = R_{sr} \tau'_{w_0} \tag{A.106} \\
h^*_s : \Delta_s & \vdash v_{sr} : R_{sr} \Delta_{w_0} \tag{A.107}
\end{align*}
\]
Every type variable appearing in the judgement from system W-SR in the hypothesis of the rule is fresh. So, we know that
\[
\text{dom}(R_{sr}) \bigcap \text{FTV}(R_1) = \emptyset \tag{A.108}
\]
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
\[
(R_{sr} R_1) \tau'_{w_1} = (R_{sr} R_1) \tau'_{w_0} \tag{A.110}
\]
Now, from this last equation we know that exists an unifier $U^1_0$ and a substitution $R^t_1$ such that
\[
\begin{align*}
R_{sr} R_1 & = R^t_1 U^1_0 \tag{A.111} \\
\tau'_{w_1} & \neg \equiv U^1_0 \tau'_{w_0} \tag{A.112}
\end{align*}
\]
Applying \((\text{w-dcase})\) rule to A.105, A.93, A.99, A.112, and the predicates \(\Delta_w\) (returned by \(\vdash_w\)) such that

\[
\Delta_w = (C^1 \in \text{U}^1_0 \ T_{w_1} \ \tau^{w_1}_{w}, \ U^1_0 \ \Delta_{w_1}),
\]

\[
C^1 \in \text{U}^1_0 \ T_{w_1} \ \tau^{w_1}_{w}, \ \text{HasC} \ (U^1_0 \ T_{w_1} \ \tau^{w_1}_{w}, \ C^1 \ (U^1_0 \ \tau^{w_1}_{w})),
\]

we obtain

\[
\Delta_w, U^1_0 \ T_{w_1} \ \Delta_{w_1}, U^1_0 \ \Delta_{w_0} \ | \ U^1_0 \ T_{w_1} \ T_{w_2} \ \Gamma
\]

\[
\vdash_w \ \text{case} \ \epsilon \ \text{of}
\]

\[
C^1 \ D^1 \ x_1 \rightarrow c_1
\]

\[
\implies
\]

\[
\text{protocase} \ \epsilon_1 \ \text{with} \ h^w \ \text{of}
\]

\[
C^1_{\text{w}_m} \ x_1 \rightarrow \epsilon_1 [h^w/v^1_1/h^1_1]
\]

Let us take \(R = R^1\) and

\[
v^1_w = (\text{if} \ v^1 \ C^1 \in v^1_{cs} \ \text{then} \ v_1[v^1_1/h^1_1] \ \text{else} \ \bullet, v^1_{m_1}, v^1_{cs}, v_e, v_{sr})
\]

Now, we have to prove that

a) \(S \approx R^1 U^1_0 T_{w_1} T_{w_2}\)
a') \(\tau^{w_1}_{w} = R^1 U^1_0 T_{w_1} T_{w_2} \ \tau^{w_1}_{w}\)
b) \(\tau^{w_1}_{w} = R^1 U^1_0 \ \tau^{w_1}_{w}\)
c) \(\Delta_s \vdash \epsilon^1 \ : \ R^1 (\Delta_w, U^1_0 T_{w_1} \ \Delta_{w_1}, U^1_0 \ \Delta_{w_0})\)
d) \(\text{protocase} \ \epsilon^1 \ \text{with} \ v^1_{cs} \ \text{of} = (\text{protocase} \ \epsilon \ \text{with} \ h^w \ \text{of}
\]

\[
C^1_{\text{w}_m} \ x_1 \rightarrow \epsilon_1 [v^1_1/h^1_1] \quad C^1_{\text{w}_m} \ x_1 \rightarrow \epsilon_1 [h^w/v^1_1/h^1_1]
\]

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

\[
\Delta_s \vdash R^1 (C^1 \in U^1_0 T_{w_1} \ \tau^{w_1}_{w}, \ U^1_0 \ \Delta_{w_1},
\]

\[
C^1 \in U^1_0 T_{w_1} \ \tau^{w_1}_{w}, \ \text{HasC} \ (U^1_0 \ T_{w_1} \ \tau^{w_1}_{w}, \ C^1 \ (U^1_0 \ \tau^{w_1}_{w})),
\]

ConstrsOf \(U^1_0 T_{w_1} \ \tau^{w_1}_{w},
\]

\[
U^1_0 T_{w_1} \ \Delta_{w_1},
\]

\[
U^1_0 \ \Delta_{w_0}
\]

which is equivalent to

\[
\Delta_s \vdash \ C^1 \in R^1 U^1_0 T_{w_1} \ \tau^{w_1}_{w}, \ R^1 U^1_0 \ \Delta_{w_1},
\]

\[
C^1 \in R^1 U^1_0 T_{w_1} \ \tau^{w_1}_{w}, \ \text{HasC} \ (R^1 U^1_0 \ T_{w_1} \ \tau^{w_1}_{w}, \ C^1 \ (R^1 U^1_0 \ \tau^{w_1}_{w})),
\]

ConstrsOf \(R^1 U^1_0 T_{w_1} \ \tau^{w_1}_{w},
\]

\[
R^1 U^1_0 T_{w_1} \ \Delta_{w_1},
\]

\[
R^1 U^1_0 \ \Delta_{w_0}
\]
Applying A.95, A.100, A.101, A.108, A.109, and A.111, we obtain

\[
\Delta_s \vdash C^1 \in \tau_{e_s}^t \land R_1 \Delta_{w_1},
\]
\[
C^1 \in \tau_{e_s}^t \land \text{HasC } \tau_{e_s}^t C^1 \tau_{e_s}^t,
\]
\[
\text{ConstrsOf } \tau_{e_s}^t,
\]
\[
R_e \Delta_{w_e},
\]
\[
R_{sr} \Delta_{w_0}
\]

In order to prove this entailment, it is enough to prove

\[
\Delta_s \vdash C^1 \in \tau_{e_s}^t \land R_1 \Delta_{w_1}
\]  \hspace{1cm} (A.115)
\[
\Delta_s \vdash C^1 \in \tau_{e_s}^t \land \text{HasC } \tau_{e_s}^t C^1 \tau_{e_s}^t
\]  \hspace{1cm} (A.116)
\[
\Delta_s \vdash \text{ConstrsOf } \tau_{e_s}^t
\]  \hspace{1cm} (A.117)
\[
\Delta_s \vdash R_e \Delta_{w_e}
\]  \hspace{1cm} (A.118)
\[
\Delta_s \vdash R_{sr} \Delta_{w_0}
\]  \hspace{1cm} (A.119)

Item **A.115**. By hypothesis, it is very easy to verify that

\[
\Delta_s \vdash v_1^s : C^1 \in \tau_{e_s}^t \land \Delta_v^s, v_{cs}^s : \text{ConstrsOf } \tau_{e_s}^t
\]  \hspace{1cm} (A.120)

Applying Lemma 3.9 with A.103 and A.120, we obtain

\[
\Delta_s \vdash \textbf{if}_v C^1 \in v_{cs}^s \textbf{ then } v_1[v_1^s/h_1^s] \textbf{ else } C^1 \in \tau_{e_s}^t \land R_1 \Delta_{w_1}
\]

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

\[
\text{\textbf{protocase}}_v e_s^t \text{ with } v_{cs}^s \text{ of } = \begin{cases} 
\text{\textbf{protocase}}_v e_{w_v}^t \text{ with } h_{v_w}^w \text{ of } \\
C_{v_{m_1}^s}^{1} \ x_{1}^s \rightarrow e_{1}^t[v_1^s/h_1^s] \\
C_{h_{v_{m_1}^s}}^{1} \ x_{1}^t \rightarrow e_{1}'[w_1^w/v_1^w/h_1^w] \\
\text{[if}_v C^1 \in v_{cs}^s \text{ then } v_1[v_1^s/h_1^s] \text{ else } \bullet, v_{m_1}^s, v_{cs}^s, v_e^w, v_{sr}^w, h_{v_{m_1}^s}^w, h_{v_{c_s}^s}^w, h_{w_e}^w, h_{w_0}^w] 
\end{cases}
\]

Since \( h_{w_0}^w \) does not appear in the residual \( \text{\textbf{protocase}}_v \), the last equation is equivalent to

\[
\text{\textbf{protocase}}_v e_{w_v}^t \text{ with } h_{v_w}^w \text{ of } \\
C_{v_{m_1}^s}^{1} \ x_{1}^t \rightarrow e_{1}'[w_1^w/v_1^w/h_1^w] \\
\text{[if}_v C^1 \in v_{cs}^s \text{ then } v_1[v_1^s/h_1^s] \text{ else } \bullet, v_{m_1}^s, v_{cs}^s, v_e^w, h_{v_{m_1}^s}^w, h_{v_{c_s}^s}^w, h_{w}^w, h_{w}^w, h_{w_e}^w, h_{w_0}^w] 
\]

By Lemma 4.12 and substitutions, we know that \( \text{EV}(e_{w_v}^t) \subseteq h_{w_v}^w \) and \( \text{EV}(e_{w_v}^t[h_{v_{m_1}^s}/h_{w_{m_1}^w}]) \subseteq h_{v_{m_1}^s}^w \), where \( h_{w_v}^w \neq h_{w_{m_1}^w}^w \neq h_{v_{m_1}^s}^w \). The evidence variables \( h_{v_{m_1}^w} \) and \( h_{v_{c_s}^s} \) are fresh, so the last equation is equivalent to

\[
\text{\textbf{protocase}}_v e_{w_v}^t [v_e/h_{w_v}^w] \text{ with } h_{v_w}^w[v_{cs}^s/h_{v_{m_1}^w}^w] \text{ of } \\
C_{h_{v_{m_1}^w}}^{1}[v_{m_1}^w/h_{v_{m_1}^w}^w] \ x_{1}^t \rightarrow e_{1}'[v_{v_{1}^w}/h_{v_{1}^w}/h_{v_{1}^w}] \text{[if}_v C^1 \in v_{cs}^s \text{ then } v_1[v_1^s/h_1^s] \text{ else } \cdot} / h_{v_{1}^w}^w] 
\]
Applying substitutions and A.97, we obtain that

\[ \text{protocase}_{v} e'_{v} \text{ with } v_{cs}^{s} \text{ of} \]
\[ C_{v_{m1}}^{1} x'_{1} \rightarrow e'_{w_{1}}[h_{w_{1}}/h_{v_{1}}] \text{ if } C^{1} \in v_{cs}^{s} \text{ then } v_{1}[v_{s}^{s}/h_{s}] \text{ else } \bullet /h_{v_{1}}^{w} \]

Since \( EV(v_{1}) \subseteq h_{1}^{s} \), \( EV(v_{1}^{s}) \subseteq h_{s} \), \( EV(v_{cs}^{s}) \subseteq h_{s} \) and \( h_{1}^{s} \neq h_{s} \neq h_{v_{1}}^{w} \), we have that

\[ \text{protocase}_{v} e'_{v} \text{ with } v_{cs}^{s} \text{ of} = \text{protocase}_{v} e'_{v} \text{ with } v_{cs}^{s} \text{ of} \]

\[ C_{v_{m1}}^{1} x'_{1} \rightarrow e'_{w_{1}}[v_{s}^{s}/h_{s}] \text{ if } C^{1} \in cs^{s} \text{ then } v_{1} \text{ else } \bullet /h_{w_{1}}^{w}[v_{s}^{s}/h_{1}^{s}] \]

Summarizing, we have to prove that

\[ \text{protocase}_{v} e'_{v} \text{ with } v_{cs}^{s} \text{ of} = \text{protocase}_{v} e'_{v} \text{ with } v_{cs}^{s} \text{ of} \]

\[ C_{v_{m1}}^{1} x'_{1} \rightarrow e'_{w_{1}}[v_{s}^{s}/h_{s}] \text{ if } C^{1} \in cs^{s} \text{ then } v_{1} \text{ else } \bullet /h_{w_{1}}^{w}[v_{s}^{s}/h_{1}^{s}] \]

In order to prove this, we have to use the definitions appearing in Figures 3.6 and 3.7.

Case \( v_{cs}^{s} = \{ C^{k} \}_{k \in I} \): By Definition 3.6, we only need to consider when \( 1 \in I \). So, it is enough to prove that

\[ \lambda x'_{1} \cdot e'_{1}[v_{s}^{s}/h_{1}^{s}] = \lambda x'_{1} \cdot e'_{w_{1}}[v_{s}^{s}/h_{1}^{s}] \text{ if } C^{1} \in v_{cs}^{s} \text{ then } v_{1} \text{ else } \bullet /h_{w_{1}}^{w}[v_{s}^{s}/h_{1}^{s}] \quad (A.121) \]

By the form of \( v_{cs}^{s} \) and the reduction rule for \( \text{if}_{v} \) in Figure 2.4, this last equation is equivalent to

\[ \lambda x'_{1} \cdot e'_{1}[v_{s}^{s}/h_{1}^{s}] = \lambda x'_{1} \cdot e'_{w_{1}}[v_{s}^{s}/h_{w_{1}}^{w}][v_{s}^{s}/h_{1}^{s}] \]

The result follows from A.104.

Case \( v_{cs}^{s} = h' \): By Definition 3.7, it is enough to prove that

\[ \lambda x'_{1} \cdot e'_{1}[v_{s}^{s}/h_{1}^{s}][D_{n}/h'] = \lambda x'_{1} \cdot e'_{w_{1}}[v_{s}^{s}/h_{1}^{s}] \text{ if } C^{1} \in h' \text{ then } v_{1} \text{ else } \bullet /h_{w_{1}}^{w}[v_{s}^{s}/h_{1}^{s}][D_{n}/h'] \quad (A.122) \]

for any \( D_{n} \) of the form \( \{ C^{k} \}_{k \in I} \). By substitution, \( EV(v_{1}) \subseteq h_{1}^{s} \) and \( h' \neq h_{1}^{s} \), we have that this last equation is equivalent to

\[ \lambda x'_{1} \cdot e'_{1}[v_{s}^{s}/h_{1}^{s}][D_{n}/h'] = \lambda x'_{1} \cdot e'_{w_{1}}[v_{s}^{s}/h_{1}^{s}][D_{n}/h'] \text{ if } C^{1} \in D_{n} \text{ then } v_{1} \text{ else } \bullet /h_{w_{1}}^{w}[v_{s}^{s}/h_{1}^{s}][D_{n}/h'] \]

Finally, we only have to consider when \( 1 \in D_{n} \), proceeding in the same way as for \( v_{cs}^{s} = \{ C^{k} \}_{k \in I} \).

Case \((B + 1) - \text{branches})\): 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

$$
\Delta_s \mid S \Gamma \vdash_s \text{case } e \text{ of } \quad (C^D_k \ x_k \rightarrow e_k)_{k \in B}
$$

$$
: \tau
\hookrightarrow
$$

\text{proto case } e' \text{ with } v^s_w \text{ of } \quad (C^D_k \ x'_k \rightarrow e'_k [v^s_w / h^s_k])_{k \in B}

$$
: \tau
\hookrightarrow
$$

\text{proto case } e'_w \text{ with } h^w_w \text{ of } \quad (C^k_w \ x'_k \rightarrow e'_w [h^w_w / h^w_w])_{k \in B}

is related to

$$
\Delta_w^B, A^B_1 \Delta_w, A^B_2 U^1_0 \Delta_{w_0} \mid A^B_1 T_{w_0} \Gamma
$$

$$\vdash_w \text{ case } e \text{ of } \quad (C^D_k \ x_k \rightarrow e_k)_{k \in B}
$$

$$
: \tau
\hookrightarrow
$$

\text{proto case } e'_w \text{ with } h^w_w \text{ of } \quad (C^k_w \ x'_k \rightarrow e'_w [h^w_w / h^w_w])_{k \in B}

by means of a substitution $R^{IH_B}$ and evidence $v_{wIH_B}$ such that

$$
S \cong R^{IH_B} A^B_1 T_{w_0}
$$

(A.125)

$$
\tau'_{e_s} = R^{IH_B} A^B_1 T_{w_0} \tau'_{w_0}
$$

(A.126)

$$
\tau'_{s} = R^{IH_B} U_{B-1} \tau'_{w_B}
$$

(A.127)

$$
\Delta_s \vdash v_{IH_B} : R^{IH_B} (\Delta^B_w, A^B_1 \Delta_w, A^B_2 U^1_0 \Delta_{w_0})
$$

(A.128)

\text{proto case } e'_s \text{ with } v^s_w \text{ of } \quad (C^k_{w_{v_{w}} \Delta^B_w} \ x'_k \rightarrow e'_k [v^s_w / h^s_k])_{k \in B}

= (\text{proto case } e'_w \text{ with } h^w_w \text{ of } \quad (C^k_{w_{v_{w}} \Delta^B_w} \ x'_k \rightarrow e'_w [h^w_w / h^w_w])_{k \in B}

[A_wIH_B] [h^B_w, h_{w_0}, h_{w_0}]

(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

$$
\lambda x^B_{w_{B+1}}. e_{B+1} : D(C^B) \rightarrow^B \tau \hookrightarrow \lambda x'^B_{B+1} \cdot e'_{B+1} : \tau'_{B+1} \rightarrow \tau'_{s}
$$

(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

$$
\Delta_{w_{B+1}} \mid T_{w_{B+1}} A^B_1 T_{w_0} \Gamma \vdash_w \lambda x^B_{w_{B+1}}. e_{B+1} : D(C^B) \rightarrow^B \tau \hookrightarrow \lambda x'^B_{B+1} \cdot e'_{w_{B+1}} : \tau'_{w_{B+1}} \rightarrow \tau'_{w_{B+1}}
$$

(A.131)
\[ R^{IH_B} \approx R_{B+1} T_{B+1} \]  
(A.132)
\[ \tau^t_{B+1} = R_{B+1} \tau^t_{w_{B+1}} \]  
(A.133)
\[ \tau^t_s = R_{B+1} \tau^t_{w_{B+1}} \]  
(A.134)
\[ \Delta^s_{B+1} \vdash v_{B+1} : R_{B+1} \Delta_{w_{B+1}} \]  
(A.135)
\[ \lambda \epsilon^t_{B+1} \cdot e^t_{B+1}^s = (\lambda \epsilon^t_{B+1} \cdot e^t_{w_{B+1}}) [v_{B+1} / h_{w_{B+1}} ] \]  
(A.136)

Using A.127, A.132, and A.134, we obtain
\[ R_{B+1} \tau^t_{w_{B+1}} = R_{B+1} T_{B+1} U^B_{B-1} \tau^t_{w_{B}} \]  
(A.137)

Then, there exist an unifier \( U^{B+1}_B \) and a substitution \( R^{B+1}_{B+1} \) such that
\[ \tau^t_{w_{B+1}} \sim U^{B+1}_B T^{B+1}_{w_{B+1}} U^B_{B-1} \tau^t_{w_{B}} \]  
(A.138)
\[ R_{B+1} = R^{B+1}_{B+1} U^{B+1}_B \]  
(A.139)

We can apply \( \langle w, \text{dcase} \rangle \) rule with judgements \( \vdash w \) and unifiers appearing in hypothesis of A.124, A.131, and A.138, obtaining
\[ \Delta^w_{B+1} T_{B+1} A^B_{B+1} \Delta_{w_e}, U^B_{B+1} T_{B+1} A^B_{B+1} U^1_{B+1} \Delta_{w_0} | U^B_{B+1} T_{B+1} A^B_{B+1} T_{w_e} \Gamma \]
\( \vdash_w \text{case } e \text{ of } (C^0_k \ x_k \rightarrow e_k)_{k \in B+1} \)
\( \vdash \)
\( \langle \text{pocase}_v \ e^t_{w_e} \rangle \text{ with } h^w_{w_e} \text{ of } (C^k_{h^w_{w_e}} \ x^t_{k} \rightarrow e^t_{w_e} [h^w_{v_k} / h^w_{w_k} ] )_{k \in B+1} \)
\( \vdash U^{B+1}_B \tau^t_{w_{B+1}} \)  
(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 \( \langle \text{univ} \rangle \) three times to A.128. In other words, we have
\[ \Delta^s \vdash v^B_w : R^{IH_B} \Delta^B_w \]  
(A.141)
\[ \Delta^s \vdash v_{w_e} : R^{IH_B} A^B_1 \Delta_{w_e} \]  
(A.142)
\[ \Delta^s \vdash v_{sr} : R^{IH_B} A^B_2 U^1_{0} \Delta_{w_0} \]  
(A.143)

Let us take \( R = R^{B+1}_{B+1} \) and
\[ v^t_w = (v^B_w, \text{ if } C^{B+1}_e \in v^s_{es} \text{ then } v^s_{B+1} [v^s_{B+1} / h^s_{B+1}] \text{ else } v^s_{m_{B+1}}, v_{w_e}, v_{sr} ) \]

Now, we have to prove that
\( \text{a) } S \approx R^{B+1}_{B+1} U^B_{B+1} T_{B+1} A^B_1 T_{w_e} \)
\( \text{a') } \tau^t_{v_{es}} = R^{B+1}_{B+1} U^B_{B+1} T_{B+1} A^B_1 T_{w_e} \tau^t_{v_{es}} \)
\( \text{b) } \tau^t_s = R^{B+1}_{B+1} U^B_{B+1} \tau^t_{w_{B+1}} \)
\( \text{c) } \Delta^s \vdash v^t_w : R^{B+1}_{B+1} (\Delta_w, U^{B+1}_{B+1} T_{B+1} A^B_1 \Delta_{w_e}, U^{B+1}_{B+1} T_{B+1} A^B_2 U^1_{0} \Delta_{w_0}) \)

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
\[
\Delta_w = U_B^{B+1} T_{B+1} \Delta^B_w,
\]
\[
C^{B+1} + U_B^{B+1} T_{B+1} A_1^B \tau_{w_e} ? U_B^{B+1} \Delta_{w_{B+1}},
\]
\[
C^{B+1} + U_B^{B+1} T_{B+1} A_1^B \tau_{w_e} ? \text{HasC} \left( U_B^{B+1} T_{B+1} A_1^B \tau_{w_e} \right) C^{B+1} \left( U_B^{B+1} \tau^I_{w_{B+1}} \right)
\]

Applying (Univ) several times to item c) and distributing substitution inside predicates, it is enough to prove
\[
\Delta_s \vdash R'_{B+1} U_B^{B+1} T_{B+1} \Delta^B_w
\]
\[
\Delta_s \vdash C^{B+1} \in R'_{B+1} U_B^{B+1} T_{B+1} A_1^B \tau_{w_e} ? R'_{B+1} U_B^{B+1} \Delta_{w_{B+1}}
\]
\[
\Delta_s \vdash C^{B+1} \in U_B^{B+1} T_{B+1} A_1^B \tau_{w_e} ? \text{HasC} \left( U_B^{B+1} T_{B+1} A_1^B \tau_{w_e} \right) C^{B+1} \left( U_B^{B+1} \tau^I_{w_{B+1}} \right)
\]
\[
\Delta_s \vdash R'_{B+1} U_B^{B+1} T_{B+1} A_1^B \Delta_w
\]
\[
\Delta_s \vdash R'_{B+1} U_B^{B+1} T_{B+1} A_2^B U_0^1 \Delta_{w_0}
\]

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
\[
\Delta_s \vdash v_{B+1}^s : C^{B+1} \in \tau_{e_s}^I, \Delta_{B+1}^s, v_{cs}^s : \text{ConstrsOf} \tau_{e_s}^I
\]

By A.135, A.139, item a'), and application of Lemma 3.9 to A.135 and A.150, we obtain
\[
\Delta_s \vdash \text{if}_v C^{B+1} \in v_{cs}^s \text{ then } v_{B+1}^s[v_{B+1}^s/h_{B+1}^s] \text{ else } \bullet : C^{B+1} \in R'_{B+1} U_B^{B+1} T_{B+1} A_1^B \tau_{w_e} ? R'_{B+1} U_B^{B+1} \Delta_{w_{B+1}}
\]

Item A.147) holds by Item a'), A.133, and A.139, obtaining that
\[
\Delta_s \vdash C^{B+1} \in \tau_{e_s}^I, \text{HasC} \tau_{e_s}^I C^{B+1} \tau_{e_s}^I
\]

which it is true by hypothesis with evidence $v_{B+1}^s$.

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
\[
\text{protcase}_v, e_{w_e}^w \text{ with } h_{v_{w_e}}^w \text{ of }
\]
\[
\text{protcase}_v, e_{w_e}^w \text{ with } h_{v_{w_e}}^w \text{ of }
\]
Since $EV(v_{cs}^s) \subseteq h_s$, $EV(v_{B+1}^s[v_{B+1}^s/h_{B+1}^s]) \subseteq h_s$, $EV(v_{w_e}^w) \subseteq h_{w_e}$, in addition to $EV(v_{sr}^s) \subseteq h_{sr}$, and $h_s \neq h_{w_e} \neq h_{sr}$, we obtain that A.152 is equivalent to
\[
\text{protcase}_v, e_{w_e}^w \text{ with } h_{v_{w_e}}^w \text{ of }
\]
\[
\text{protcase}_v, e_{w_e}^w \text{ with } h_{v_{w_e}}^w \text{ of }
\]
and by definition of \( v_{IH} \), this can be rewritten as

\[
\text{protn-case}_v e'_w, \text{with } h^w_{v_{cs}} \text{ of } \quad (C^k_{v_{cs}} x'_{k} \rightarrow e'_w [h^w_{v_{cs}} / h^w_{v_{cs}^{k}}])_{k \in B+1})
\]

\[
[v^s_{IH} \text{ if } v C^{B+1} \text{ then } v_{B+1} / h^s_{v_{B+1}^{k} / h^s_{B+1}^{k}} \text{ else } \bullet, \quad v_{m_{B+1}^{s}} / h^w_{v_{B+1}^{s}} / h^w_{v_{m_{B+1}^{s}}}
\]

Because of evidence variables \( h^w_{v_{B+1}^{s}} \) and \( h^w_{v_{B+1}^{s}} \) appear only in the \( (B+1) \)th branch, A.154 is equivalent to

\[
\text{protn-case}_v e'_w, \text{with } h^w_{v_{cs}} \text{ of } \quad (C^k_{v_{cs}^{k}} x'_{k} \rightarrow e'_w [h^w_{v_{cs}^{k}} / h^w_{v_{cs}^{k}}])_{k \in B}) [v^s_{IH} / h^B_{v_{cs}^{k}} / h^w_{v_{cs}^{k}} / h^w_{v_{cs}^{k}}]
\]

\[
[v^s_{IH} \text{ if } v C^{B+1} \text{ then } v_{B+1} / h^s_{v_{B+1}^{s}} / h^s_{v_{B+1}^{s}} \text{ else } \bullet, \quad v_{m_{B+1}^{s}} / h^w_{v_{B+1}^{s}} / h^w_{v_{m_{B+1}^{s}}}
\]

To sum up, we have to prove that

\[
\text{protn-case}_v e'_s, \text{with } v^s_{cs} \text{ of } \quad (C^k_{v_{cs}^{k}} x'_{k} \rightarrow e'_s [v^s_{cs} / h^s_{k}])_{k \in B+1})
\]

\[
= (\text{protn-case}_v e'_w, \text{with } h^w_{v_{cs}} \text{ of } \quad (C^k_{v_{cs}^{k}} x'_{k} \rightarrow e'_w [h^w_{v_{cs}^{k}} / h^w_{v_{cs}^{k}}])_{k \in B}) [v^s_{IH} / h^B_{v_{cs}^{k}} / h^w_{v_{cs}^{k}} / h^w_{v_{cs}^{k}}]
\]

\[
[v^s_{IH} \text{ if } v C^{B+1} \text{ then } v_{B+1} / h^s_{v_{B+1}^{s}} / h^s_{v_{B+1}^{s}} \text{ else } \bullet, \quad v_{m_{B+1}^{s}} / h^w_{v_{B+1}^{s}} / h^w_{v_{m_{B+1}^{s}}}
\]

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 \( B+1, x'_{B+1}^{s}, e'_{B+1}^{s}, v_{B+1}, h^w_{v_{B+1}^{s}}, \) and A.136 instead of \( 1, x'_{1}^{s}, e'_{1}^{s}, v_{1}, h^w_{v_{1}^{s}}, \) and A.104.

- **Case \( v_{cs} = h^s \)** 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 \( h^w_{v_{B+1}^{s}} \) instead of \( 1, x'_{1}^{s}, e'_{1}^{s}, v_{1}, \) and \( h^w_{v_{1}^{s}}. \)

### A.26 Proof of theorem 7.11 from section 7.1

**Theorem 7.11.** A system defining a simplification relation, extended with rules (SHASC), (SCTS), (SG-TRUE), (SG-FALSE), and (SU-HC) still defines a simplification relation.

**Proof:** The proof is by induction on the derivation, taking one case for each rule:

- **(SHASC):** (i) It holds that \( \emptyset \vdash n : \text{HasC } D_n C^k S \tau' \) by (HASC), and also (ii) \( \text{HasC } D_n C^k S \tau' \vdash \emptyset. \)
• (scts): (i) It holds that \( \emptyset \vdash \{ C^k \}_{k \in I} : \text{ConstrsOf} \ D_n \) by (Constrs-of), and also (ii) \( \text{ConstrsOf} \ D_n \vdash \emptyset \).

• (sg-false): (i) It holds \( \emptyset \vdash C^k \in D_n \Rightarrow \Delta \) by applying (Guard-false) to \( C^k \notin D_n \), and also (ii) \( \bullet : C^k \in D_n \Rightarrow \Delta \vdash \emptyset \).

• (su-hc): It trivially holds that

\[
h_1 : \text{HasC } \tau' \ C^k \ S_1'' \vdash h_1 : \text{HasC } \tau' \ C^k \ S_1
\]

Additionally, we obtain that

\[
h_1 : \text{HasC } \tau' \ C^k \ S_1'' \vdash h_1 : \text{HasC } \tau' \ C^k \ S_2'' \quad \text{(A.156)}
\]

by applying (Unify-hasC) to \( h_1 : \text{HasC } \tau' \ C^k \ S_1'' \vdash S_1'' \sim S_2'' \). Thus, (i) holds by applying (Univ) to A.155 and A.156. The item (ii) holds trivially since

\[
\text{HasC } \tau' \ C^k \ S_1'' \vdash \text{HasC } \tau' \ C^k \ S_1' \quad \text{(A.155)}
\]

A.27 Proof of theorem 7.12 from section 7.1

**Theorem 7.12.** A system defining a simplification relation, extended with rules (shc-g) and (sntl-g) still defines a simplification relation.

**Proof:** The proof is by induction on the derivation, taking one case for each rule:

• (sg-true): (i) It holds \( h : \Delta \vdash h : C^k \in D_n \Rightarrow \Delta \) by applying (Guard-true) to \( C^k \in D_n \) and \( \Delta \vdash \Delta \), and also (ii) \( C^k \in D_n \Rightarrow \Delta \vdash \Delta \) by applying (Into-guard-true) to \( C^k \in D_n \) and \( \Delta \vdash \Delta \).

• (shc-g): (i) We know that

\[
h_1 : \text{HasC } \tau' \ C^k \ \tau'' , h_2 : \Delta \vdash h_2 : C^k \in \tau' \Rightarrow \Delta \quad \text{(A.157)}
\]

by applying (HasC-guard) to \( h_1 : \text{HasC } \tau' \ C^k \ \tau'' , h_2 : \Delta \vdash h_2 : \Delta \). Additionally, we trivially know that

\[
h_1 : \text{HasC } \tau' \ C^k \ \tau'' , h_2 : \Delta \vdash h_1 : \text{HasC } \tau' \ C^k \ \tau'' \quad \text{(A.158)}
\]

The result follows by applying (Univ) to A.157 and A.158. (ii) On the other hand, we know that

\[
h_1 : \text{HasC } \tau' \ C^k \ \tau'' , h_2 : C^k \in \tau' \Rightarrow \Delta \vdash h_1 : \text{HasC } \tau' \ C^k \ \tau'' \quad \text{(A.159)}
\]

Additionally, we have that

\[
h_1 : \text{HasC } \tau' \ C^k \ \tau'' , h_2 : C^k \in \tau' \Rightarrow \Delta \vdash h_2 : \Delta \quad \text{(A.160)}
\]

by applying (Elim-HasC-guard) to \( h_1 : \text{HasC } \tau' \ C^k \ \tau'' , h_2 : \Delta \vdash h_2 : \Delta \). The result follows by applying (Univ) to A.159 and A.160.
(\text{SENTL-G}): (i) We know that

\[ h' : C^k \in \tau' ? \Delta', h_c : \text{ConstrsOf } \tau' \vdash h_c : \text{ConstrsOf } \tau' \]  

\hspace{1cm} (A.161)

Additionally, we have that

\[ h' : C^k \in \tau' ? \Delta', h_c : \text{ConstrsOf } \tau' \vdash v_i : C^k \in \tau' ? \Delta \]  

\hspace{1cm} (A.162)

by applying \((\text{ENTL-GUARD})\) to \(h' : \Delta' \vdash v : \Delta\) which holds by IH applied to \(\text{Id}; h \leftarrow v \mid h : \Delta \triangleright h' : \Delta'\) and \(h_c : \text{ConstrsOf } \tau' \vdash h_c : \text{ConstrsOf } \tau'\). The result follows by applying \((\text{Univ})\) to A.161 and A.162. On the other hand, (ii) we know that

\[ C^k \in \tau' ? \Delta, \text{ConstrsOf } \tau' \vdash \text{ConstrsOf } \tau' \]  

\hspace{1cm} (A.163)

and we also have that

\[ C^k \in \tau' ? \Delta, \text{ConstrsOf } \tau' \vdash C^k \in \tau' ? \Delta' \]  

\hspace{1cm} (A.164)

by applying \((\text{ENTL-GUARD})\) to \(\Delta \vdash \Delta'\) which holds by IH applied to \(\text{Id}; h \leftarrow v \mid h : \Delta \triangleright h' : \Delta'\) and \(\text{ConstrsOf } \tau' \vdash \text{ConstrsOf } \tau'\). The result follows by applying \((\text{Univ})\) to A.163 and A.164.

\section{A.28 Proof of theorem 7.16 from section 7.2}

\textbf{Theorem 7.16.} The heuristic presented is correct wrt. the definition of the constraint solving relation. That is:

1. MonomorphicST finds a solution for \(t\), respecting the predicates HasC.

2. If \((S,T,C,\Delta_f,\Delta') = \text{stepSolve } t \Delta \text{ and } s \notin V\) then

\[ S, T; C \mid \Delta + \Delta_f \triangleright_V \Delta'. \]

\textbf{Proof:}

1. A new sum-type is introduced. It has a constructor for each predicate HasC found in the predicate assignment. If several predicates HasC appear for the same constructor, then the argument’s residual type in the declaration can be any of those argument’s residual type in such predicates— that is because of all of them would be eventually unified by the rule \((su-HC)\). Thus, a valid sum-type definition is obtained.

2. It holds by applying the simplification rule \((hasc)\) several times.
Theorem 2.28, 45, 50
Theorem 4.13, 45, 48
Theorem 4.14, 45, 48
Theorem 4.5, 45, 47
Theorem 4.6, 45, 47