Polymorphic Dynamic Typing

Aspects of proof theory and inference

Master's Thesis

Jakob Rehof

DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø, Denmark
Electronic mail: `rehof@diku.dk`

August 15, 1995

Abstract

We study dynamic typing in continuation of Henglein's *dynamically typed λ-calculus*, with particular regard to proof theoretic aspects and aspects of polymorphic completion inference.

Dynamically typed λ-calculus provides a formal framework within which we can reason in a precise manner about properties of the process of *completion* for higher order programming languages. Completions arise from raw programs by insertion of *type coercions* which model run-time type operations of tagging and checking/untagging. Central among the problems studied in dynamic typing are the notions of *minimization* of run-time type coercions in completions and *safety* of completions.

From the monomorphic framework of Henglein's system, we work towards a polymorphic generalization which eventually comprises Hindley-Milner style polymorphism, discriminative, tagged sum types, regular recursive types and so-called coercive types with a notion of coercion parameterization. The resulting system can be seen as a form of polymorphic qualified type system which aims at a common generalization of dynamic typing and certain systems of *soft typing.*

Starting from an equational presentation of categorical co-products, we develop a dynamic typing calculus within which we can reason about completion. We develop a generalized theory of conversion and reduction on coercions and completions. We establish fundamental proof theoretic properties of the calculus, including confluence of general completion reduction and existence and uniqueness of minimal completions in a pragmatically important class of completions. We also give a proof-theoretic treatment of a generalization of Thatte's *quasi-static typing* which arises naturally in the framework of dynamic typing.

We study problems of automatic completion inference in our generalized setting, and we describe an implementation of a simple, global dynamic type analysis. We investigate the problem of achieving modularity of inference, and we argue that safety and minimality are conflicting goals in a modular setting. It is suggested that the use of coercion parameters is an important ingredient in reconciling these goals.

The problem of dynamic type inference for a realistic dynamically typed language is studied, with special reference to *Scheme.* Completion inference as a basis for static debugging tools and for a high-level translation of *Scheme* to *ML* is investigated. We identify major problems and suggest solutions to some of them.

Preface

This report is a slightly revised version of my Master's Thesis which was completed (under the same title) at DIKU, dept. of Computer Science, University of Copenhagen, in March 1995, under the supervision of Fritz Henglein.

The revisions concern only typing errors and the like.

Acknowledgements

As I trust will be evident to anyone who reads this work even superficially, my debt to Henglein is greater than I could possibly give full credit here. The basic framework of dynamic typing as well as many, many central, particular ideas are due to him, stemming not only from his papers on dynamic typing but also from conversations much more numerous than I could possibly have expected. For that, as well as his general support of my studies, I am very greatful.

The degree to which I have taken these ideas in an interesting direction is, however, an entirely different matter which I leave for the reader to judge.

I must also express my gratitude to Jesper Jørgensen who has instructed me since my childhood in computer science; his work on representation analysis provided a constant source of inspiration, and I have been lucky to be able to draw freely on his knowledge of programming languages.

Thanks also to Dave Sands and Mads Tofte for their willingness to lend me their expertise on several points.

Finally, I must thank Morten Heine Sørensen. Although he had nothing to do with this project, working with him on other matters has been a great pleasure.

March 22,

Jakob Rehof

It is a typical error to think that the use of logically complex concepts, or big axioms (*e.g.*, large cardinals), makes "greater" mathematics than elementary concepts or axioms (elephants are not the most intelligent animals ...)

J.-Y. Girard [1]

[1] In the introduction to his *Proof Theory and Logical Complexity*.

Contents

Chapter 1

Introduction

The present introductory chapter falls in three parts. The first part (up to Section 1.6) gives a gentle and rather thorough introduction to the goals, techniques and problems of dynamic typing. The second part (Section 1.6) is a short and rather terse development of the minimal framework of operational semantics needed to appreciate the development in the remainder of the thesis. This is not a gentle introduction, but rather a reference section which probably requires some acquaintance with operational semantics to be easily digested. The remaining two sections of the introduction gives a short statement of the purpose of this thesis and an overview of its contents.

1.1 Static *vs* dynamic typing

Most modern programming languages can be clasified as being either *statically typed* or *dynamically typed*. Examples of the former kind are *Pascal*, *C*, *ML*, examples of the latter kind are *Scheme*, *Lisp* and *Prolog*.

Static type systems in practical use typically come with a *type inference* or *type checking* algorithm which assigns types to program phrases at compile time. If an entire program can be assigned such a compile-time (or *static*) type, the program is accepted as being well-typed and is compiled; otherwise it is rejected and the compilation process abandoned. Typically, a well typed program will be (ideally, at least) provably free of *run-time type errors*, witnessing the slogan "well typed expressions do not go wrong" of Milner [Mil78]. Run-time type errors originate from misuse of constants, such as applying a non-function, as in (`1` `2`), or applying a functional primitive to an object outside its domain of definition, like in (+ `1` `true`)

Dynamic type systems impose no *static* type constraints on programs, deferring type checking to *run-time*. At compile time, run-time type chekcs are compiled into the taget code. A run-time type error typically causes program execution to be aborted, often with some information about the nature of the error. It is perhaps worth stressing that dynamically typed languages *do* impose type discipline on programs. The difference to static typing is that the type system is effective at run-time only, not that there is no type system. For instance, a language such as *Scheme* is far from being *untyped*. If one tries to execute `(1 2)` or `(car 'a)`, the *Scheme* system will issue an error-message. This is possible, only because a whole system of run-time type checking is active as part of the run-time system. On the other hand, in an *untyped* language (such as an assembly language) the result of executing meaningless instructions may be totally unpredictable.

Among the benefits of static typing are efficiency and safety, because it eliminates the need to perform run-time type checking, and it finds many programming errors at compile time. The cost is loss of programming flexibility, since no decidable type inference system can accept all and only those programs that will not go wrong at run-time. Therefore, any type inference algorithm will necessarily reject some run-time type safe programs as ill-typed.

1.2 Mixed systems

Although static *vs* dynamic typing may seem to be conflicting philosophies of language design, several lines of recent work have aimed at combining the benefits of both views. We refer to such combined systems as *mixed type systems* We briefly introduce the major developments related to this area. The presentation is systematic rather than historical. For a more comprehensive overviw we refer to Section 8.2.

Soft Typing

Soft typing as considered here comes in two variants, namely the system developed at Rice by Cartwright, Fagan and Wright in [Fag90], [CF91], [CF92], [Wri94], [WrCa94], and the system developed by Aiken, Wimmers and Lakshman in [AWL94]. Soft typing aims at constructing type inference mechanisms for dynamically typed languages. One can single out three (interrelated) goals central to all existing proposals for soft typing:

Universality

A soft type inference system performs type analysis at compile-time, but rather than rejecting a program as ill-typed, it will insert *run-time type cheks* at program points where the static analysis cannot guarantee type safety. Thus, *every* syntactically correct program is accepted by a soft type system. This is the *universality* property.

Minimality

A major technical objective for soft type inference is to *minimize* the number of type checks. Run-time type cheks will not be generated for sections of the source program which are statically classifiable as being type safe. In particular, for a statically typable program no run-time type checks will be generated at all. The objective of minimality is achieved proportionally to the expressiveness of the type system. Therefore, soft type systems typically attempt to extend conventional polymorphic systems, such as the *Hindley-Milner* system, by introducing more expressive type languages.

Expressiveness

A common core of focus, in the various soft type systems that have been proposed, is to increase expressiveness and programming flexibility particularly wrt. *heterogenous* or *non-uniform* objects, such as lists with elements of different static types, and recursion over such structures. The main technical innovation for achieving this goal is the combination of *union types* and *recursive types*, integrated with *Hindley-Milner*-style polymorphism. These features are found in all present soft type systems.

Partial Typing

Partial typing as introduced by Thatte [Tha88], [Tha94] adresses specifically the *flexibility* issue wrt. heterogenous objects. This is done by introduction of a subtype system, distinguished by a *top type* called Ω, and generated from the axiom $\tau \leq \Omega$, making Ω a supertype of *every* type. Typability of non-uniform structures is achieved by injecting objects into a common supertype, Thatte's system of partial typing is not universal; not all programs can be assigned a type, so Thatte's partial type system remains within the framework of static typing.

In [Gom90] Gomard proposed to use what is effectively also a *top type*, called **untyped**. A modification of Milner's algorithm $\mathcal{W}$ was proposed, which would accept any program by injecting program phrases that could not be assigned a type by $\mathcal{W}$ into the top type. This system is therefore universal.

Quasi-Static Typing

In [Tha90] Thatte specifically adressed the problem of mixing static and dynamic typing. Thatte introduced *type coercions* to explicitly model the insertion of run-time type checks at program points where static typability brakes down, and he considered *coercion reduction* for rewriting dynamic type operations into *minimal* or normal forms. This proposal therefore has the *universality* property, and one of its expressed aims is *minimality.* However, quasi-static typing is limited to a framework of *fixed type declarations* in which programs are assumed to carry explicit type annotations on functional parameters.

Dynamic Typing

Henglein [Hen92], [Hen92a], [Hen94] considers the problem of *completion inference* for dynamically typed languages. In [Hen92] completions arise from a program by insertion of *dynamic type coercions* in the style of Thatte [Tha90]. However, no fixed type declarations are assumed. The dynamic type coercions come in pairs, representing dual, or inverse, run-time type operations. They are, respectively, *tagging* operations and *check-and-untag* operations.

Henglein introduces a *universal type* **Dynamic** representing all completely tagged objects. Every program has a completion well typed in the simply typed λ-calculus generated from **Dynamic** and other usual base types. Hence, the work of Henglein also achieves *universality.* An objective of [Hen92] is *tagging optimization*, the elimination of run-time type operations. Hence, the issue of *minimality* is central to that work. The process of minimization proceeds by cancellation of pairs of inverse operations. This is in contrast to soft typing which assumes all objects to be tagged at run-time. In comparison with systems of soft typing, the type system of the framework of [Hen92], [Hen92a], [Hen94] is less expressive, since it is a monomorphic extension of the simply typed λ-calculus.

In [Hen92a] and [Hen94] these ideas were systematized and generalized, leading in [Hen94] to a comprehensive formal framework of so-called *dynamically typed λ-calculi.* Since the present work lies in direct continuation of this work by Henglein, we give an extended introduction to his framework of dynamic typing.

1.3 Behind the curtains

In a sense, a dynamic type system is visible to the programmer only through its effects, in case an error occurs. In particular, there is no *visible* type information for the programmer to look at. As said, it all operates so to speak behind the curtains, as part of the run-time system. Let us unveil what takes place there, by making it explicit.

First, let us elaborate the earlier remark that it is certainly possible to talk about *types* in, say, *Scheme*. They look as follows [1]

$$\begin{array}{rcl} tc & ::= & \texttt{boolean} \\ & | & \texttt{symbol} \\ & | & \texttt{char} \\ & | & \texttt{vector} \\ & | & \texttt{pair} \\ & | & \texttt{number} \\ & | & \texttt{string} \\ & | & \texttt{procedure} \end{array}$$

These types divide the data domain into eight *disjoint* sets. Coming from a static polymorphic type discipline, one might perhaps be inclined to think that,*e.g.*, since pairs are of type `pair`, a function such as `car` would have a type like

$$\texttt{car} : \texttt{pair}(\alpha, \beta) \to \alpha$$

or, to write this in a more *Scheme*-like fashion,

$$\texttt{car} : \texttt{procedure}(\texttt{pair}(\alpha, \beta), \alpha)$$

But at the level of *Scheme*'s type system we cannot actually use the type constructors to construct complex types. In fact, as every other *Scheme* object, `car` has one and only one type of the above, namely

$$\texttt{car} : \texttt{procedure}$$

This amounts to saying that the types *tc* rather function like *type tags*, which when assigned to an object reveal just the *outermost* type constructor of the (expected) type of the object. We can see that this is all that is needed for performing run-time type checks; for instance, all we need to know about `f` at the operation `(f a)` is that `f` is a procedure. In a larger context such as

$$(\texttt{if}\ (\texttt{f}\ \texttt{a})\ 0\ 1)$$

we must require that `f` send `a` to a `boolean` [2] but checking that this requirement is met can be deferred to the context of the conditional, so that rather than imposing a more detailed requirement on `f` we impose the `boolean` requirement on `(f a)`.

One can make this completely systematic. Let us assume, for each type constructor *tc*, a *tagging operation* called `tc!` and a *check-and-untag operation* called `tc?` For instance, we would have operations `boolean!`, `boolean?`, `procedure!`, `procedure?` and so on. These are the *dynamic type operations* performed at run-time. They are not part of the source program, but they are compiled into the target code as part of the run-time system. Informally, a *tagging* operation `tc!` simply adds the type tag onto its argument:

$$(\texttt{tc!}\ \texttt{x}) = < \texttt{tc}, \texttt{x} >$$

[1]The type system is part of the language definition. See [CR91]. Some types, such as input/output ports are left out here. Also, we could have included `nil` as a type.

[2]We assume here, for the sake of illustration, that the conditional consumes a boolean in the test; this is not so in *Scheme*.

using the pairing notation $< \bullet, \bullet >$ to represent an object (at the second component) with a tag (at the first component.) The *check* operation `tc?` inspects its argument and (assuming it to be tagged) decides whether the tag is correct, `tc?` accepting just the tag `tc`; in case the acceptable tag is found, it is stripped off the object which is returned, and in case a non-acceptable tag is found, a *run-time type error* is generated. This could mean a *program abort*, possibly with an error message; one could also imagine other possibilites. Assuming an operation `ERROR`(), we can write, in *SML*-style,

```
fun tc? x = case x of
              (tc, y) => y
            | _       => ERROR()
```

In this intuitive semantics, we shall normally think of `ERROR` as an *abortive* operation, which models the run-time type error escape to top-level, possibly accompanied by some error message, as is usual in a dynamically typed language. Under this view of the operation, we can think of it in terms of the definition

```
fun ERROR() = raise TypeError
```

Given these operations we can establish a discipline for generating run-time type operations for a source program. We imagine this, slightly abstractly, as a process whereby the run-time type operations are *inserted* into the *program text.* Of course, in a real setting, the source text is itself translated into target code; but since we are here interested in the type operations only (and not in code generation in general) we shall abstract from the compilation of source code. This we do by simply leaving it as it stands in the source program. We shall refer to the process of inserting run-time type operations as *program completion.* Underlying this is the view that, in a dynamically typed language, a raw source program is an *incomplete object*, because it does not specify any operations for the cases where run-time errors occur. To stress the difference in status of the type operations from the source code, we shall write the application of a type operation in special brackets, as in

$$[\texttt{procedure!}]\ \texttt{M}$$

The natural discipline of program completion obeys the principles:

- At every *data construction point* the appropriate tagging operation is inserted
- At every *data destruction point* the appropriate check-and-untag operation is inserted

A construction point is a program point at which a datum is constructed at run-time. Construction points (signified by the box $\Box$) include

$\Box$ `cnst` , for every constant `cnst`

$\Box$ (`lambda` ...)

$\Box$ (`cons` ...)

$\Box$ (+ ...)

Note in particular the construction points of the type $\Box$ (+ ...). This is so classified because the addition constructs a new number from those given to it as arguments. A destruction point is a program point where a datum is *consumed* by application of a destructive operation. Destruction points include the first argument position of the application (writing application explicitly as `@`),

$(@\ \square\ \bullet)$

$(\texttt{car}\ \square)$

$(\texttt{if}\ \square\ \bullet\ \bullet)$

$(+\ \square\ \square)$

These are the points at which run-time type constraints are imposed upon run-time objects. The discipline for completion which obeys the directions just outlined is referred to as *canonical.* The rationale of it is that

- Every run-time object should have a run-time type description tagged onto it so that it can be checked, and
- All run-time type checks are deferred as long as possible, in that they occur only at points where type demands are imposed by a destructive operation. This ensures that the dynamic type system is consistent with the semantics of the language, in the sense that a run-time error cannot be generated, unless a meaningless operation would otherwise be executed.

From these principles we can derive a translation which sends every syntactically correct program `M` into its *canonical completion* $[\![\texttt{M}]\!]$. We specify it for a small example subset of a *Scheme*-like language [3]

$$\begin{array}{rcl}
[\![\texttt{x}]\!] & \equiv & \texttt{x} \\
[\![\texttt{true}]\!] & \equiv & [\texttt{boolean!}]\texttt{true} \\
[\![\texttt{0}]\!] & \equiv & [\texttt{number!}]\texttt{0} \\
[\![(\texttt{lambda}\ (\texttt{x})\ \texttt{M})]\!] & \equiv & [\texttt{procedure!}](\texttt{lambda}\ (\texttt{x})\ [\![\texttt{M}]\!]) \\
[\![(@\ \texttt{M}\ \texttt{N})]\!] & \equiv & (@\ ([\texttt{procedure?}][\![\texttt{M}]\!])\ [\![\texttt{N}]\!]) \\
[\![(\texttt{if}\ \texttt{M}\ \texttt{N}\ \texttt{P})]\!] & \equiv & (\texttt{if}\ ([\texttt{boolean?}][\![\texttt{M}]\!])\ [\![\texttt{N}]\!]\ [\![\texttt{P}]\!]) \\
[\![(+\ \texttt{M}\ \texttt{N})]\!] & \equiv & [\texttt{number!}](+\ ([\texttt{number?}]\texttt{M})\ ([\texttt{number?}]\texttt{N}))
\end{array}$$

According to this, we can view every operator of the source language as a dynamic operation, which works on dynamically typed objects. The compiler uses (usually several) primitive operations of the target language to implement this. For instance, a compiler would implement an equation such as

$$(+\ \texttt{x}\ \texttt{y}) = [\texttt{number!}](\oplus\ ([\texttt{number?}]\texttt{x})\ ([\texttt{number?}]\texttt{y}))$$

where the operation $\oplus$ will implement non-dynamic addition, involving machine code instructions such as `ADD`.

[3] In general, we consider the conditional construct a consumer of booleans in the test argument. This is not so in *Scheme*.

1.4 Laws

In the previous section we have taken an important first step towards *reasoning about dynamic type operations*, simply by *making them explicit.*

It is already evident that certain interesting, algebraic-looking laws hold for the dynamic type operations. Let us note that, when presented with a `tc`-tagged object, the check operation `tc?` acts just like the second projection π_2. For instance, one has

$$\begin{aligned}[\mathtt{number?}]([\mathtt{number!}]\,\mathtt{x}) &= \pi_2(<\mathtt{number}, \mathtt{x}>)\\ &= \mathtt{x}\end{aligned}$$

In general, it seems fair to adopt the law

$$\mathtt{tc?} \circ \mathtt{tc!} = \mathtt{id} \tag{1.1}$$

where `id` is an algebraic way of representing the *no-op* operation, or, as we might say, the identity. We can codify this by adopting the equation

$$[\mathtt{id}]\,\mathtt{M} = \mathtt{M} \tag{1.2}$$

From this we can get larger derivations like

$$\begin{aligned}[\![(+\ 1\ 2)]\!] &= [\mathtt{number!}](+\ ([\mathtt{number?}][\![1]\!])([\mathtt{number?}][\![2]\!]))\\ &= [\mathtt{number!}](+\ ([\mathtt{number?}][\mathtt{number!}]\mathtt{1})([\mathtt{number?}][\mathtt{number!}]\mathtt{2}))\\ &= [\mathtt{number!}](+\ 1\ 2)\end{aligned}$$

From these examples it is a short step to start thinking that equations such as these could be made the basis of *static optimization* of the dynamic type operations. From this perspective, we should want to *orient* the equation:

$$\mathtt{tc?} \circ \mathtt{tc!} \rightarrow \mathtt{id} \tag{1.3}$$

and for `id` we would have

$$[\mathtt{id}]\,\mathtt{M} \rightarrow \mathtt{M} \tag{1.4}$$

Under any reasonable assumptions about the implementation at hand, the right-hand-sides of these rewrite rules are *more efficient*, and therefore more desirable, than the left-hand-sides. All we must presuppose is that performing no operation at all is at least as efficient (and, probably more so) than performing the dynamic type operations. In this vein, we see that a program like $[\![(+\ 1\ 2)]\!]$ can be *rewritten* into a form in which *all* dynamic type checks have disappeared. In general, and more technically, we can imagine a *minimization* process on completions via a *reduction relation* on completions induced in the standard way [4] from *notions of reduction* such as are given by rule 1.3 and rule 1.4. Thus we can place our laws within a general, established rewrite theoretical framework, which means, among other things, that we have access to techniques of that discipline. In particular we can possibly

- make *precise concepts* about the process of minimization,
- formulate and try to answer questions about that process using those techniques, and we can

[4] A *notion of reduction* is a binary relation over the term language; it induces a *reduction relation* by reflexive, symmetric, transitive and compatible closure. See [Bar84]

- seek inspiration as to how we could find interesting algorithms for such a process.

Driven by algebraic patterns, we may go on to ask whether the pairs `tc!`, `tc?` are truly *inverses* of each other. For this to hold we should also require

$$\mathtt{tc!} \circ \mathtt{tc?} = \mathtt{id} \tag{1.5}$$

But here the matter appears to be somewhat different. If we compute with principles adopted so far, we can reason as follows. Suppose `M` is actually of the form `<tc,N>`. Then

$$\begin{array}{rcl} [\mathtt{tc!}][\mathtt{tc?}]\mathtt{M} & = & < \mathtt{tc}, \pi_2(\mathtt{M}) > \\ & = & < \pi_1(\mathtt{M}), \pi_2(\mathtt{M}) > \\ & \overset{?}{=} & \mathtt{M} \end{array}$$

We have put a question mark on top of the last equation, because this is the one whose validity we are asking about. In the equation

$$< \pi_1(\mathtt{M}), \pi_2(\mathtt{M}) > = \mathtt{M} \tag{1.6}$$

we recognize the law of *surjective pairing.* It is a form of *η-rule*, because it concerns the elimination of a constructor ($< \bullet, \bullet >$) applied to its corresponding destructor (π); in the same way, equation 1.1 is a form of *β-rule*, because it concerns the elimination of a destructor applied to its corresponding constructor. Hence, since we normally do not assume any η rules, we may not be so surprised to realize that we *cannot* derive equations 1.6 and 1.5 from the intuitive semantics given for the dynamic type operations [5] In fact, using our intuitive semantics, we would have, *e.g.*,

$$[\mathtt{boolean!}][\mathtt{boolean?}][\mathtt{number!}]\ \mathtt{1} = \mathtt{ERROR}()$$

assuming here that `ERROR()` is abortive wrt. the dynamic type operations. This suggests that we can easily imagine semantics which do *not* validate equation 1.5, since this equation yields

$$[\mathtt{boolean!}][\mathtt{boolean?}][\mathtt{number!}]\ \mathtt{1} = [\mathtt{number!}]\ \mathtt{1}$$

For such a semantics, we should say that the completion $[\mathtt{boolean!}][\mathtt{boolean?}][\mathtt{number!}]\ \mathtt{1}$ of the program `1` is a very unfortunate one, because it can generate a run-time type error where another completion of the same program wouldn't (witness the completion $[\mathtt{number!}]\ \mathtt{1}$) We may say that the former completion is not *safe*, and in a formal framework we should wish to give precise meaning to this notion. Among the properties we would wish to hold wrt. safety is that the canonical completion of a program is safe.

1.5 Dynamically typed λ-calculus

Considerations such as were made above motivated Henglein to begin a systematic study of dynamic typing. This work has taken several directions [Hen91], [Hen92]. [Hen92a], [Hen94]. In these works, a formal framework emerged, within which central problems and concepts of dynamic typing can be given precise meaning. This is the so-called *dynamically typed λ-calulus*, and it is referred to in the present work as system **D**. For a full introduction, we must refer the reader to [Hen92], [Hen94]. In the present section, we outline main features of the calculus and

[5] Recall also that surjective pairing is not definable in the λ-calculus, see [Bar84]

the central concepts it gives rise to. In Appendix A, we give a short but full definition of the calculus, both for reference and for the notational conventions adopted in the present report.

Henglein realized that completing a program could be viewed as a process which is *controlled by a conventional, static type system.* He introduced a *universal type*, called **Dynamic**, and considered a *simply typed λ-calculus* generated from the type **Dynamic** and the other (usual) base types, such as **boolean**, **number** etc. Intuitively, the type **Dynamic** contains all completely tagged objects. Also, we think intuitively of the type **Dynamic** in denotational terms as a *universal domain* [6] $\mathbb{D}$, solving the recursive domain equation [7]

$$\mathbb{D} \cong \mathbb{B} \oplus [\mathbb{D} \to \mathbb{D}]$$

with injection $\Phi : [\mathbb{D} \to \mathbb{D}] \to \mathbb{D}$ into the right component of $\mathbb{D}$ and projection $\Psi : \mathbb{D} \to [\mathbb{D} \to \mathbb{D}]$ from $\mathbb{D}$ onto the right component, such that

$$\Psi \circ \Phi = id$$

Dynamic type operations are viewed as *type coercions.* In general, for every type constructor tc of arity k there is a pair of tagging- and check-and-untag coercions, `tc!` and `tc?` to which are assigned the *signatures* (refer to Figure A.1 of Appendix A)

$$\texttt{tc}! : tc(\mathbf{Dynamic}^{(1)}, \ldots, \mathbf{Dynamic}^{(k)}) \rightsquigarrow \mathbf{Dynamic}$$

$$\texttt{tc}? : \mathbf{Dynamic} \rightsquigarrow tc(\mathbf{Dynamic}^{(1)}, \ldots, \mathbf{Dynamic}^{(k)})$$

A signature $\tau \rightsquigarrow \tau'$ assigned to a coercion means that the coercion can be applied to an object of type τ, and that the resulting, coerced object can be regarded as an object of type τ'. For instance, we would have the signatures

$$\texttt{procedure}! : (\mathbf{Dynamic} \to \mathbf{Dynamic}) \rightsquigarrow \mathbf{Dynamic}$$

$$\texttt{procedure}? : \mathbf{Dynamic} \rightsquigarrow (\mathbf{Dynamic} \to \mathbf{Dynamic})$$

$$\texttt{boolean}! : \mathbf{boolean} \rightsquigarrow \mathbf{Dynamic}$$

$$\texttt{boolean}? : \mathbf{Dynamic} \rightsquigarrow \mathbf{boolean}$$

Given an assignment of signatures $\tau \rightsquigarrow \tau'$ to coercions, a *dynamic type system* (refer to Figure A.3 of Appendix A) can be defined, by adding to the rules of the simply typed λ-calculus the new typing rule for *coercion application*:

$$\frac{\Gamma \vdash M : \tau \quad c : \tau \rightsquigarrow \tau'}{\Gamma \vdash [c]M : \tau'}$$

One can verify that the canonical completion, as defined above, has type **Dynamic** for every program (refer also to Figure A.3 of Appendix A for canonical completions.)

The coercions above are called *primitive* coercions. There is also a rule for constructing *composite* coercions, together with a rule for assigning a signature to a composition in terms of the signatures of its components: If c, d are coercions, then so is $d \circ c$, and the signature is given by the rule

$$\frac{c : \tau \rightsquigarrow \tau' \quad d : \tau' \rightsquigarrow \tau''}{d \circ c : \tau \rightsquigarrow \tau''}$$

[6] See, *e.g.*, [Gun92]

[7] Assuming just the type of the booleans besides the universal dynamic type

There is a rule for constructing *induced* coercions, together with rules for assignment of signatures. In general, if $tc^{(k)}$ is a type constructor of arity $k > 0$, and if $c_1, \ldots, c_k$ are coercions, then $\mathtt{tc}(c_1, \ldots, c_k)$ is also a coercion. In the minimal framework sketched so far, there is only one constructor of non-zero arity (constructors of arity 0 are constants, such as **boolean**) namely the function space constructor, $\rightarrow$. Hence, there is a coercion $c \rightarrow d$, for every pair of coercions c, d. The rule for assignemnt of signatures to this induced coercion is

$$\frac{c : \tau \rightsquigarrow \tau' \quad d : \sigma \rightsquigarrow \sigma'}{c \rightarrow d : (\tau' \rightarrow \sigma) \rightsquigarrow (\tau \rightarrow \sigma')}$$

Note that $\rightarrow$ is contravariant-covariant wrt. $\rightsquigarrow$. This is reminiscent of the variance of function space wrt. type inclusion in subtyping disciplines, and in fact, one can view the induced coercions as *proof terms* which explicitly encode a kind of subtyping proof step. This is analogous to the composition $\circ$ on coercions, which can be seen as encoding a proof step corresponding to the rule of transitivity in conventional subtype systems. For co-variant type constructors (such as pairing, $*$) the signature assignment rule will be this one:

$$\frac{c_i : \tau_i \rightsquigarrow \tau_i'}{tc(c_1, \ldots, c_k) : \mathtt{tc}(\tau_1, \ldots, \tau_k) \rightsquigarrow \mathtt{tc}(\tau_1', \ldots, \tau_k')}$$

1.5.1 Equational theory and coherence

In general there will be many ways to complete the *same* program at the *same* type. It is not difficult to see that there are in fact always *infinitely* many ways of doing this. Given a program `M` we know that $[\![\mathtt{M}]\!]$ is a completion at **Dynamic**, hence every term in the infinite series

$$[\![\mathtt{M}]\!],$$

$$[\mathtt{boolean!}][\mathtt{boolean?}][\![\mathtt{M}]\!],$$

$$[\mathtt{boolean!}][\mathtt{boolean?}][\mathtt{boolean!}][\mathtt{boolean?}][\![\mathtt{M}]\!] \ldots$$

is yet a new way of completing `M` at **Dynamic**. This rather dull theme has other variations, of course, but there are also more interesting examples. Consider for instance the fixpoint combinator $\mathbf{Y}$:

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

One completion of $\mathbf{Y}$ at $\mathbf{D} \rightarrow \mathbf{D}$ is (we abbreviate **Dynamic** as $\mathbf{D}$):

$$\begin{aligned}\mathbf{Y}' \equiv \lambda f : \mathbf{D}.\ & ([\mathtt{procedure?}][\mathtt{procedure!}](\lambda x : \mathbf{D}.([\mathtt{procedure?}]f)(([\mathtt{procedure?}]x)\,x))) \\ & [\mathtt{procedure!}](\lambda x : \mathbf{D}.([\mathtt{procedure?}]f)(([\mathtt{procedure?}]x)\,x))\end{aligned}$$

and another one, also at $\mathbf{D} \rightarrow \mathbf{D}$, is the simpler

$$\begin{aligned}\mathbf{Y}'' \equiv \lambda f : \mathbf{D}.\ & (\lambda x : \mathbf{D} \rightarrow \mathbf{D}.([\mathtt{procedure?}]f)(x\ ([\mathtt{procedure!}]x))) \\ & (\lambda x : \mathbf{D}.([\mathtt{procedure?}]f)(([\mathtt{procedure?}]x)\,x))\end{aligned}$$

If we are interested in minimizing dynamic type coercions, it becomes interesting to know whether we can somehow *relate* all possible ways of completing the same program at the same type. It is not difficult to verify that, with just the equations we have seen so far (equations (1.1), (1.5) in Section 1.4) one cannot prove that, for instance, $\mathbf{Y}' = \mathbf{Y}''$, even though these terms are both completions of the same program, $\mathbf{Y}$, at the same type, $\mathbf{D} \rightarrow \mathbf{D}$.

The full *dynamically typed λ-calculus* (here called **D**) of [Hen92], [Hen94] contains a rich *equational theory.* We refer the reader to the works cited and to Appendix A for complete definitions, and we confine ourselves here to a brief overview. The equational theory falls in three parts. There is a set of equality axioms for coercions, called C (refer to Figure A.2 of Appendix A) which describe identifications we always assume to hold. These eliminate "trivial" distinctions between completions, from which we want to abstract. These are associativity of coercion composition, distributivity of composition of the type constructors (system **D** mentions only the function space) and equations which describe the identity coercions (including equation (1.2) of Section 1.4.) The second set of axioms is called E (refer to Figure A.4 of Appendix A.) These axioms describe how coercions can be "moved around" within a completion. They can be thought of as *flow-equations*, because they let coercions float according to a form of undirectional, static abstraction of the data-flow of the underlying program. Since we always wish to reason *modulo* [8] the equations in group C, the theory E contains the theory C. Finally, there are the so-called ϕ- and ψ-equations. These are just the equations (1.1) and (1.5) which were discussed in Section 1.4. The full equational theory of **D** is referred to as $E\phi\psi$ here [9]

The theory $E\phi\psi$ is exactly strong enough to identify all and only the completions of a given coercion-free program at a given type. It therefore possess the so-called *coherence* property ([Hen94]). In more detail, write $M \cong M'$ (read M and M' *congruent*) if and only if M and M' are both completions at the same type and M and M' have the same *coercion erasure* (obtained by deleting all coercions from the term.) Then the coherence property means that

$$M \cong M' \text{ implies } E\phi\psi \vdash M = M'$$

If we view the process of completion as a process for typing incomplete program phrases, then coherence means that the equational theory gives *formal* conditions, which are necessary and sufficient for *any given semantics* to be *independent of typing.* This notion of semantic invariance under variations of typing arises in principle whenever typing is associated with modifications (like insertion of coercions) of the underlying program, depending on *how* it is being typed [10] The notion of coherence originates in coercion-based analyses of subtyping, where coercions appear as proof witnesses in explicit subtyping systems, [BCGS89], [BGS90], [BCGS91], [CG90].

1.5.2 Reduction

In Section 1.4 we noted that we can *orient* the ϕ- and ψ-equations, leading to a notion of *static completion optimization*, by rewriting under the notions of reduction:

$$\begin{array}{lccc} (\phi) & \texttt{tc?} \circ \texttt{tc!} & \rightarrow & id \\ (\psi) & \texttt{tc!} \circ \texttt{tc?} & \rightarrow & id \end{array}$$

[8]The reader be *warned* that the notion ($R\,mod\,E$) of a reduction relation R *modulo* an equational theory E is taken, in the present report, in the sense of *equivalence class rewriting* (as in, *e.g.*, [DeJo90]) which is *not* the same as the notion for which the concept $R\,mod\,E$ is used in [Hue81].

[9]Meaning in this introduction. Later in the report, we shall follow the conventions of Appendix A, where we use $\phi\psi$ to denote both coercion- and completion equality (and reduction, see below), the latter taken *modulo E*; we rely on disambiguation by context. Although this should never be a problem, we distinguish notationally in this introduction, to make sure the reader is not confused.

[10]Note carefully that "being typed" here means "recieving a typ*ing*", not just "recieving a type". The latter means just the association af a *type* to a program; the former means the association of a *typing proof tree* to a program. The modifications mentioned typically consist in a form of *explicit term-encoding of the typing proof tree* within the typed program. In other words, the proof steps used to type the program are recorded within (a richer version of) the program itself, often in such a way that the typing can be uniquely reconstructed from the typed (richer version of the) program.

In principle, we could choose to work with both of these reductions. However, we also saw that the intuitive semantics considered in Section 1.4, does *not* satisfy the ψ-equations. In other words, that semantics does *not* satisfy the coherence conditions, and, indeed, we gave an example of two congruent completions with quite different behaviour under that semantics. In principle, we could regain coherence, if we were to introduce a form of *algebraic run-time simplification* of coercions under $\phi\psi$-equality. However, such semantics would be very costly, and therefore we shall, for practical purposes, *restrict* our model of run-time coercion evaluation to include *ϕ-reduction* (oriented ϕ-equality) but not ψ-reduction. Compositions of the form `tc!` $\circ$ `tc?` are therefore regarded as being liable to generate *run-time type errors.* In these decisions we follow closely the development in [Hen94]. The reductions ϕ and ψ on coercions are shown in Figure A.2 of Appendix A. These reductions can be lifted to completions in a systematic way, as is indicated in the following section.

1.5.3 Safety, minimality and proof theory

The rejection of ψ-equality leads to the problem of safety, as was briefly indicated in Section 1.4. Intuitively, a completion is *safe*, if it does not generate avoidable run-time type errors. It is possible to give a purely formal definition of safety, within the framework of system **D**. This is done in [Hen94], where a coercion $c : \tau \rightsquigarrow \tau'$ is called (formally) safe, if for every $c' : \tau \rightsquigarrow \tau'$ it holds that c' ψ-reduces *modulo* ϕ to c, written:

$$\phi \vdash c' \rightarrow_{\psi}^{*} c$$

The notions of ϕ- and ψ-reduction (and conversion) are lifted to completions in a systematic way (see Figure A.4 of Appendix A.) In general, if R is a notion of reduction on coercions, we can lift R to a notion of reduction on completions by passing to R *modulo* the flow-equations in E. In this way, we get a natural concept of *safe completions*, defined in analogy with the definition of safety for coercions.

The *formal* notion of safety for completions only models the assymetry, observed earlier, between ϕ- and ψ-reduction wrt. the intuitive semantics. The central property is that two completions are equally safe if and only if they are ϕ-convertible. The point is that this can be the case, even though one of the completions contains a ψ-redex. Consider, as a simple example, the coercion

`boolean!` $\circ$ `boolean?` $\circ$ `boolean!`

Even though this coercion contains a ψ-redex, it is safe at its signature, because the ψ-redex can be eliminated under ϕ-conversion, and evaluating an application of this coercion under the intuitive semantics will generate no errors.

As will be explained in more detail in Section 1.6, this will also work as an acceptable notion of *operational safety*, provided we assume *call-by-name evaluation.* However, as will be seen in the same section, we have the very important restriction:

- *The equations in E are not sound for call-by-value evaluation*

An example to show this is given later in this chapter, in continuation of Property 1.6.10. Hence, new notions of operational safety must be brought into play, if we wish to consider our completion calculi faithful models for run-time type operations in a call-by-value language. One such notion is given in Section 1.6 (Definition 1.6.6) and in the course of this report we shall have occasion to introduce yet another.

It is important to realize, in the midst of the, admittedly, somewhat heavy formal framework of dynamic typing, that the calculi of coercions and completions ultimately have very *practical* motivations. The practical interest in coercion- and completion reduction lies mainly in our desire to *minimize* the amount of run-time type operations. The notion of ϕ-reduction on completions can be regarded both as a formal model of minimization, in which we can reason about properties of this process, and as a *notion of static computation over run-time type operations*, which we could in principle implement to realize the process. The formal concept of minimality is defined in terms of relation $E\phi$: a completion is called *formally minimal* if and only if it is a $E\phi$-normal form. Other, restricted, notions of minimality may be interesting for a variety of reasons. One reason is that we need some restriction if we consider run-time type optimization for a call-by-value language, since the equational theory E is unsound for call-by-value, as just mentioned. Another example is that, for reasons of computational efficiency, we may wish to consider restricted but efficiently solvable subproblems of general minimization.

The formal notion of minimality has a distinctly *proof theoretic* flavour. This should be understood correctly; the dynamically typed λ-calculus certainly does not have a standard propositions-as-types interpretation as a logical system, since if we take this view, the calculus constitutes a syntactically inconsistent logic (in the sense that all propositions are provable.) This is, in fact, a hallmark of a dynamic system, since it must give a typing to *every* program. Rather, the similarity lies in the fact that we can regard $\mathbf{D}$ as a calculus of typing-proof objects. The elimination of a $E\phi$-redex can readily be understood as a *detour-elimination* in a typing proof, much as classical proof theory views β-reduction (under a propositions-as-types reading of the typed λ-calculus [11]) as a rearrangement of a proof object which decreases the amount of logical redundancy in the proof. Let us give a concrete example of this. Consider the completion M:

$$(\lambda x : \mathbf{D}.(\texttt{[procedure?]}x\ \texttt{[boolean!]}\mathit{true})\ \ \texttt{[procedure!]}(\lambda y : \mathbf{D}.y)$$

Looking at this completion, we might say that the type coercions in it constitute redundant operations, because we do something at the argument (tagging it) which is *logically* being undone in the operator (at the checking operation.) At the argument, we coerce the type $\mathbf{D} \to \mathbf{D}$ to $\mathbf{D}$, only to make the inverse coercion at the application point. Indeed, this detour will also make itself felt from an *operational* viewpoint, since reduction of the term will lead to a ϕ-redex of coercions,

$$M \to_\beta (\texttt{[procedure?]}\,\texttt{[procedure!]}(\lambda y : \mathbf{D}.y))\ \ \texttt{[boolean!]}\mathit{true}$$

The flow-equations in E will in many cases (E is but an abstraction of the "real" flow under β conversion) be strong enough to discover this:

$$\begin{array}{lll} M & =_E & [\texttt{procedure!} \to \texttt{id}](\lambda x : \mathbf{D}.([\texttt{procedure?}]x\ \texttt{[boolean!]}\mathit{true})\ \ (\lambda y : \mathbf{D}.y) \\ & =_E & (\lambda x : \mathbf{D} \to \mathbf{D}.([\texttt{procedure?}][\texttt{procedure!}]x\ \texttt{[boolean!]}\mathit{true}))\ \ (\lambda y : \mathbf{D}.y) \\ & \to_\phi & (\lambda x : \mathbf{D} \to \mathbf{D}.(x\ \texttt{[boolean!]}\mathit{true}))\ \ (\lambda y : \mathbf{D}.y) \end{array}$$

Note how the type associated with the bound variable x becomes more articulated under this reduction.

Now, everyone would naturally agree that the reduct shown in the reduction above is a much better completion than M, at the type $\mathbf{D}$. But other examples could be given where this might be less clear, from the *operational* viewpoint. It would be easy to define an operational measure of

[11] See [Pra65] for a classic in elementary proof theory, explaining the concept of a logical detour. See, *e.g.*, [GLT89] for an exposition of the propositions-as-types and programs-as-proofs correspondence between typed programming languages and systems of logic.

completion bonity; we could simply count the number of primitive coercion reductions performed during evaluation. Clearly, the problem of finding an *operationally optimal* completion wrt. such a measure is not computable. In this regard, the stragtegy of dynamic typing may be said to be one of *formal approximation*:

- We use the *logical* notion of optimality as an approximation to (the undecidable) operational notion.

A priori, it is concievable that such a stratgey would turn out to fail utterly in practice. Whereas displacing coercions along the flow-paths under E-equality will not deteriorate the operational cost under call-by-name evaluation, this is not so under call-by-value evaluation. Consider the following example conversion (type annotations left out for readability) :

$$\begin{array}{rcl}(\lambda x.[c]M\{x := [d]x\})\ ((\lambda y.R)P) & =_E & [d \to c](\lambda x.M)\ ((\lambda y.R)P) \\ & =_E & [c]((\lambda x.M)\ [d]((\lambda y.R)P)) \\ & =_E & [c]((\lambda x.M)\ ([id \to d](\lambda y.R)P)) \\ & =_E & [c]((\lambda x.M)\ ((\lambda y.[d]R)P)) \\ & =_E & \cdots\end{array}$$

Suppose (in the extreme) that $\lambda x.M$ does not consume its argument (*e.g.*, $M \equiv (\textbf{if}\ \mathit{true}\,\mathit{true}\,x)$) and that $\lambda y.R$ when applied to P gets into a complex computation, which could involve repeated coercion reductions involving d (which perhaps diasappears into the interior of R under continuation of the conversion shown above); then, under call-by-value evaluation, this conversion may be an unfortunate one in terms of the operational cost measure. However, experience with prototype completion systems for **D** (see [Hen92], [Hen92a]) indicates that the formal approximation works well in practice. Also, closely related work on *representation analysis* by Henglein and Jørgensen ([HJ94], see also Section 8.2) confirms this.

1.5.4 Interpreting completions

A calculus such as **D** is a purely formal system, and in principle it can be regarded as just a game, like chess. Our interest in it is more serious, however, because we believe that it models something which we are interested in understanding about programming languages. We have indicated how a dynamically typed calculus can model aspects of run-time type operations already, but let us be a little more specific here. At the moment we foresee two main areas of application:

1. Compilation of dynamically typed languages

2. Static debugging of (dynamically) typed languages

In the *first* of these applications, dynamic typing must faithfully model the run-time behaviour of run-time type operations. Under this view, a completion is interpreted as a *compilation schema* for the source program, in which coercions indicate where and which run-time type operations should be compiled, checking coercions being compiled to run-time type checks and tagging coercions to run-time type tags. This aspect has been emphasized in the present introduction. In a wider perspective, we may see the language of completions as just another *statically typed* programming language. The process of completion then becomes one of *translating a dynamically typed language into a statically typed language.* In principle, this opens the door to *bringing compilation technology developed for statically typed languages to bear on the compilation of* *dynamically typed languages.* Conceptually, at least, this could be achieved by a composition of translations; if we have a compiler C_{static} for the completion language of **D**, which is a statically typed language, and we have a compiler $C_{completion}$ which completes raw, dynamic source expressions into **D**-expressions, then we can buid a compiler $C_{dynamic}$ for the dynamically typed source language by

$$C_{dynamic} = C_{static} \circ C_{completion}$$

In case the completion performed by $C_{completion}$ works by *minimization*, the step performed by $C_{completion}$ may be one of *optimization.* Thus, completion under minimization can be thought of as aiming for optimized compilation of dynamically typed languages.

In the *second* of the applications mentioned above, we interpret coercions differently. Under this view, a checking coercion can be read as a *warning* to the programmer that an error *may* occur during evaluation at some subexpression. In an expression such as

$$(\lambda x.(\textbf{if}\ M\ ([\texttt{procedure?}]x\ [\texttt{boolean!}]\,\mathit{true})\ ([\texttt{boolean!}]\mathit{true})))\ \ [\texttt{boolean!}]\mathit{true}$$

the checking coercion `procedure?` could be presented to the programmer in the form of the message:

```
Procedure application at ... may be unsafe
```

or, in a more sophisticated framework, perhaps even,

```
Procedure application at ... may be unsafe :
boolean may be bound to x
```

We could even imagine the system tracing and displaying the flow-path to the problematic boolean. All this would happen at the level of static program analysis. A twist of this way of looking at completions is that we might imagine the source language as being actually a *statically* typed language, and the process of completion then becomes one of *static type error recovery.* Under this interpretation, a coercion can be read as a *static type error message*, saying (referring to the example above) something like,

```
Type error at procedure application ... :
boolean used as a procedure
```

Perhaps techniques of dynamic typing could be used to improve the art of *finding the source of type errors* (see [Wan86].)

In principle, one could imagine all of the interpretations listed above as being exploited in a *single* framework of static debugging and compilation. However, variations are possible here. For instance, if we isolate the aspect of type error recovery, we will find that our perception of the calculus may become different, *e.g.*, the unsoundness of E for call-by-value becomes a non-issue.

1.5.5 Problems and goals of dynamic typing

We have introduced the dynamically typed λ-calculus as a formal framework for dynamic typing, in which the notions of safety and minimality appear at the point of focus. We have indicated how these notions can be defined in terms of *reduction relations* in the sense of *rewriting theory* [12] It is therefore possible to investigate dynamic typing from a rewriting theoretic perspective.

[12] See, *e.g.*, [HO80],[Klo87], [DeJo90], [Kir94]. The full reduction relations of system **D** can be regarded as rewrite specifications in *higher order syntax*, because a notion of substitution is used in the theory E.

This was initiated in [Hen94], and within this framework major *theoretical problems* of dynamic typing can be raised. In [Hen94], main problems addressed are:

1. Is $E\phi\psi$ canonical (confluent and terminating) ?
2. Is $E\phi$ canonical?

Henglein [Hen94] gave a negative answer to the first question, whereas the second question was essentially left open. It was shown that confluence fails for general λK-completions (completions of λK-terms) but it was *conjectured* that $E\phi$ is confluent and terminating over λI-completions (see [Bar84] for the λ-calculus term classes of λK, λI.) Henglein specifically conjectured this property for a version of $E\phi$ where the equational theory is *oriented* according to a notion of *polarity* of coercions [13] Other, theoretically oriented, aspects of dynamic typing are:

- The study of the complexity of *completion inference* (see below)
- The study of semantics of dynamic type completions

The main *practical* goal of dynamic typing can be stated thus:

- *Given a source program M, find an optimal completion of M among all the safe completions of M.*

Here, optimality will typically be defined in terms of some rewriting relation contained in $E\phi$, and, in the extreme, it can be taken to be $E\phi$ itself. In other words, we can scale the notion of optimality, according to the underlying notion of reduction. Dynamic typing specifically aims at finding optimal comletions *automatically*, and the process of doing so is referred to as *completion inference.*

In [Hen92] a near-linear *completion inference algorithm* for **D** was presented, relying on algorithmic techniques studied in [Hen91]. The algorithm automatically inserts coercions into an arbitrary source program, such that the resulting completion is *operationally safe* [14] and *relatively minimal.* The latter property refers to a classification of completions, according to what kinds of coercions are used and at which program points they can be inserted. Within a certain restricted completion class the algorithm will infer a "best" completion, meaning that the completion inferred contains as few dynamic type coercions as possible, within its completion class. In particular, the algorithm is *conservative*, since there is guarantee that for any program, which is already well typed (in the simply typed calculus), the algorithm will give back the program itself, *i.e.*, no coercions are inserted at all. In this sense, one will only pay (in terms of dynamic type operations) for the "necessary" amount of dynamic typing in a program.

The major *practical challenge* of dynamic typing is:

- *to construct more powerful completion inference algorithms, to implement them and to study their behaviour on realistic programs.*

1.6 Operational semantics

In this section we give operational semantics to completions of **D**. Semantic rules are given in the style of *natural semantics*; for general principles we refer to [Gun92]. The section is

[13] This reduction is referred to, in the present report, as $P_\phi *$

[14] The notion of operational safety is defined later, in Section 1.6.

intended only to give the absolutely minimal framework necessary to understand, operationally, the developement in this report. It is rather terse reading and is intended to serve more as a refernce than as an introduction. It is certainly *not* intended as an in-depth treatment of the semantics of dynamic type operations.

For generality we consider a slightly extended pure language, which includes two sets of *primitives*, a set *Const* consisting of *basic constants* ranged over by $\mathbf{b}$, and a set *Prim* of *basic functions* ranged over by $\mathbf{f}$. The pure term language is essentially the term language of **Pure Scheme** as presented in [Wri94], or of **Idealized Scheme** as presented in [Fel87], or of **Iswim** as presented in [Plo75].

Following [And94] we define three evaluation functions on $\mathbf{D}$ completions, one which is the standard Call-By-Name (CBN), one which is the standard Call-By-Value (CBV), and finally a variation on CBV, called CBV_ε, using a different strategy for evaluating run-time type errors.

When evaluating with explicit run-time type coercions there are in principle three kinds of *exceptional situations* which can occur during evaluation, and which may or will lead to the evaluator producing something different from an ordinary *value*, either no value at all or an *exceptional value*:

1. the evaluator may generate a *run-time type error* as the result of performing a run-time type check, or

2. the evaluator may be *stuck* for a given expression, or

3. the evaluator may *diverge*

Exceptional values may be produced as the end result if one of the first two situations occurs during evaluation. The presence of a run-time type error will be represented by the *error element* $\boldsymbol{\varepsilon}$, and the presence of a stuck situation will be represented by the *wrong element* $\boldsymbol{\omega}$. In case of divergence, no value is produced at all.

Thus, the explicit representation of run-time type checks allows us to distinguish between two kinds of exceptional situations, type-error situations and stuck situations, which are normally not distinguished in *pure semantics* where run-time type operations are not explicitly represented. In such semantics, both kinds of exceptional situation are usually called *stuck situations* (or *stuck states* in a framework of state transition semantics as given for, *e.g.*, abstract machines. See [Plo75], [Fel87].) Also, stuck states may not always be explicitly defined as part of the evaluation function itself; in that case, the function will be simply *undefined* for stuck situations. We could have opted similarly (eliminating the wrong element $\boldsymbol{\omega}$), but in a typed framework one is usually interested in reasoning about these situations, specifically for proving the property of *type soundness* stating that "well typed programs cannot go wrong". This we do below.

1.6.1 Evaluation rules

To accomodate for the specification of several semantics without too much verbosity we give our rules in a slightly parameterized form. The rules are divided into four classes, as given by the four figures below:

$$
\begin{array}{lcl}
W & ::= & \boldsymbol{\omega} \mid \boldsymbol{\varepsilon} \\
V_t & ::= & \mathit{true} \mid \mathit{false} \mid \mathbf{b} \mid \mathbf{f} \mid \lambda x:\tau.M \\
V_{t\varepsilon} & ::= & V_t \mid \boldsymbol{\varepsilon} \\
V_p & ::= & V_t \mid W \\
V & ::= & V_p \mid [p^+]V_t
\end{array}
$$

Figure 1.1: Value classes

$$\delta(\mathbf{f}, W) = W \qquad [W1]$$

$$\frac{M_1 \Downarrow W}{(\mathbf{if}\ M_1\ M_2\ M_3) \Downarrow W} \qquad [W2]$$

$$\frac{M_1 \Downarrow W}{M_1\ M_2 \Downarrow W} \qquad [W3]$$

$$\frac{M_2 \Downarrow W}{M_1\ M_2 \Downarrow W} \qquad [W4]$$

Figure 1.2: Propagation rules

$$V \Downarrow V \qquad [PE1]$$

$$\frac{M \Downarrow \mathbf{f} \quad N \Downarrow V_p \quad \delta(\mathbf{f}, V_p) \text{ defined}}{M\, N \Downarrow \delta(\mathbf{f}, V_p)} \qquad [PE2]$$

$$\frac{M \Downarrow \mathbf{f} \quad N \Downarrow V_p \quad \delta(\mathbf{f}, V_p) \text{ undefined}}{M\, N \Downarrow \boldsymbol{\omega}} \qquad [PE3]$$

$$\frac{M \Downarrow \mathbf{b} \quad \mathbf{b} \in \mathit{Const} \setminus \mathit{Prim}}{M\, N \Downarrow \boldsymbol{\omega}} \qquad [PE4]$$

$$\frac{M_1 \Downarrow \mathit{true} \quad M_2 \Downarrow V}{(\mathbf{if}\ M_1\ M_2\ M_3) \Downarrow V} \qquad [PE5]$$

$$\frac{M_1 \Downarrow \mathit{false} \quad M_3 \Downarrow V}{(\mathbf{if}\ M_1\ M_2\ M_3) \Downarrow V} \qquad [PE6]$$

$$\frac{M_1 \Downarrow V \quad V \notin \{\mathit{true}, \mathit{false}, \boldsymbol{\varepsilon}\}}{(\mathbf{if}\ M_1\ M_2\ M_3) \Downarrow \boldsymbol{\omega}} \qquad [PE7]$$

$$\frac{M \Downarrow^{v*} \lambda x.P \quad N \Downarrow^{v*} V_{t*} \quad P\{x := V_{t*}\} \Downarrow^{v*} V}{M\, N \Downarrow^{v*} V} \qquad [CBV]$$

$$\frac{M \Downarrow^{n} \lambda x.P \quad P\{x := N\} \Downarrow^{n} V}{M\, N \Downarrow^{n} V} \qquad [CBN]$$

Note : It is assumed that all $\mathbf{f} \in \mathit{Prim}$ are strict (in the usual sense that $\delta(\mathbf{f}, \bot) = \bot$), and that every $\mathbf{f}$ is also strict in both elements of W, *i.e.*, $\delta(\mathbf{f}, W) = W$ (cf. Figure 1.2.)

Figure 1.3: Pure evaluation rules

$$\frac{M \Downarrow V \quad [c]V \Downarrow V'}{[c]M \Downarrow V'} \qquad [CA]$$

$$\frac{p^- \equiv (p^+)^{-1}}{[p^-]([p^+]V) \Downarrow V} \qquad [C\phi]$$

$$\frac{p^- \not\equiv (p^+)^{-1}}{[p^-]([p^+]V) \Downarrow \boldsymbol{\varepsilon}} \qquad [C\boldsymbol{\varepsilon}]$$

$$[p^-]V_t \Downarrow \boldsymbol{\omega} \qquad [C\boldsymbol{\omega}]$$

$$[c]W \Downarrow W \qquad [CW]$$

$$[id]V \Downarrow V \qquad [Cid]$$

$$[c^{\tau_2}_{\tau_1} \to d^{\tau_4}_{\tau_3}]V \Downarrow \lambda x : \tau_1.[d^{\tau_4}_{\tau_3}](V\ ([c^{\tau_2}_{\tau_1}]x)) \qquad [C{\to}]$$

$$\frac{[c_1]V \Downarrow V' \quad [c_2]V' \Downarrow V''}{[c_2 \circ c_1]V \Downarrow V''} \qquad [C\circ]$$

Figure 1.4: Coercion evaluation rules

Figure 1.1 The rules of Figure 1.1 give the definition of *values* of evaluation. The values of **D** semantics are given by the class V. A value can be either an exceptional element of W, or it can be a *term value* of V_t which is the ordinary value set of *pure semantics*, or it can be a *primitive tagging coercion*, ranged over by p^+, applied to a term value. The remaining subclasses are there to specify possible restrictions of some of the evaluation rules, as explained below.

Figure 1.2 The rules of Figure 1.2 give *propagation rules* for the exceptional elements, $\boldsymbol{\varepsilon}$ and $\boldsymbol{\omega}$ of the class W. The rules $[W1]$ and $[W2]$ express that every basic function as well as the conditional is assumed to be strict wrt. both exceptional elements; note that, in particular, such functions are assumed to be *defined* for these elements. The rules $[W3]$ and $[W4]$ propagate the exceptional elements through applications. The last rule is "optional" in the sense that it will not be present in all evaluation functions to be defined, as will be explained below.

Figure 1.3 The rules of Figure 1.3 give *pure evaluation rules*. They are suitable for defining standard CBV and CBN semantics of the pure language, not containing explicit run-time type operations. The rule called $[CBV]$ is to be read as a *parameterized rule* in the value class V_{t*}. By fixing a specific choice for V_{t*} and leaving out the Call-By-Name rule $[CBN]$ we can define different versions of Call-By-Value semantics, choosing either $V_{t*} = V_t$ or $V_{t*} = V_{t\varepsilon}$, as will be explained below. By leaving out the rule $[CBV]$ we can define Call-By-Name semantics.

Note that taking $V = V_t$ and leaving out the rules $[PE3], [PE4], [PE7], [CBN]$ in Figure 1.3 defines exactly Plotkin's Call-By-Value evaluation function $Eval_V$ of [Plo75], and leaving out the rule $[CBV]$ together with the rules $[PE3], [PE4], [PE7]$ defines exactly Plotkin's Call-By-Name evaluation function $Eval_N$ of [Plo75]. Note that all rules of Figure 1.3 are independent of the *error element* $\boldsymbol{\varepsilon}$. The *wrong element* $\boldsymbol{\omega}$ is present in the rules $[PE3], [PE4], [PE7]$, and it is seen to correspond exactly to the *stuck states* of Plotkin's standard pure evaluators.

Figure 1.4 The rules of Figure 1.4 are the *coercion evaluation rules.* When taken in conjunction with the *propagation rules* and the *pure rules* they define semantics with explicit run-time type operations, as explained below. Note that the rule $[C\boldsymbol{\varepsilon}]$ is the only one, among all the rules considered in this section, which introduces the *error element* $\boldsymbol{\varepsilon}$.

The rule $[CA]$ expresses that evaluation of a coercion application proceeds by *first* evaluating the coerced object and then the application. In other words, "evaluation goes under coercions". The rule $[C\phi]$ is just *run-time ϕ reduction*, and rule $[C\boldsymbol{\varepsilon}]$ generates the error element in case a run-time type checking operation (ranged over by checking coercion p^-) is performed on a value tagged with the "wrong" type tag. There is a special introduction rule ($[C\boldsymbol{\omega}]$) for introducing $\boldsymbol{\omega}$: A run-time type check can only be evaluated if applied (at run-time) to a tagged value. The rule $[CW]$ is a special coercion *propagation rule* which propagates exceptional elements past coercion applications. The rule $[Cid]$ expresses that the identity coercions are *no-op* operations. The rule $[C \rightarrow]$ expresses the operational meaning of the higher order coercions. It interprets such a coercion as a *wrapper function*, which applies the value to a coerced argument, and then coerces the result of that application. The rule $[C\circ]$ expresses the operational meaning of coercion composition. Compositions are evaluated *inside-out.* In particular, there is *no algebraic rewriting of coercions at run-time.* Hence, a completion such as

$$[\mathtt{F!} \circ \mathtt{F?} \circ \mathtt{B!}]\mathit{true}$$

will generate a run-time error, although it is ψ-reducible to the completion value $[\mathtt{B!}]\mathit{true}$ which doesn't.

Our semantics follow those given by Andersen in [And94] closely, with one (important) exception. Apart from superficial differences of presentation, the system considered in [And94] differs from ours just in having the "lazy" rule

$$[c \rightarrow d](\lambda x : \tau.M) \Downarrow \lambda x : \tau.[d]M\{x := [c]x\}$$

instead of our rule $[C\rightarrow]$. The rule of [And94] *postpones* the operation of a higher order coercion until its argument is applied. We have two reasons for not adopting this rule

1. We feel that the rule logically belongs only in a Call-By-Name framework due to its element of lazyness. Nothing is lost wrt. expressing CBN semantics by adopting our rule, though, since (as is easily seen) the difference between our rule and the rule of [And94] disappears in a CBN discipline. In other words, we prefer our rule for giving CBV semantics.

2. The rule of [And94] imposes much stronger demands on implementation techniques, since it cannot be implemented with ordinary closures. Furthermore, this would have negative consequences for the task (to be considered later in this report) of implementing the dynamically typed language via high level translations. See Chapter 7.

We now give the definitions of the semantics $CBV, CBV_\varepsilon, CBN$. The systems CBV and CBN are the expected standard Call-By-Value and Call-By-Name semantics, respectively. With respect to the treatment of run-time type errors, standard philosophy has it that a type error goes off like a bomb : *You're dead if you step on a bomb; only, in CBN one may step on a bomb less frequently than in CBV.* In less martial jargon, a run-time type error has the effect of *aborting* computation (rather than, say, blowing up the computer). In usual implementations (and certainly, in *most* cases, in high level languages with any aspiration to practical use) execution aborts in a *controlled* manner with some kind of error message (rather than, say, a segmentation fault.) This is really what is being explicitly modelled with the error element $\boldsymbol{\varepsilon}$. It may be understood as an *abortive control operator*, like an ML exception to top level, possibly containing some kind of error message to the user.

The system CBV_ε is a non-standard CBV semantics which departs from standard philosophy in treating the error element as, in some cases, an ordinary value which can be passed around. It is mentioned here mostly for the sake of demonstrating that, whereas *ϕ convertibility and even the equational theory E of $\mathbf{D}$ is not sound wrt. ordinary CBV semantics*, one has soundness properties for the non-standard CBV_ε semantics (see below.)

We begin by defining the standard Call-By-Value semantics.

Definition 1.6.1 (Error-strict Call-By-Value, CBV, $\Downarrow^v$)
The usual Call-By-Value evaluation is strict in the error element $\boldsymbol{\varepsilon}$. This relation is given by

- All rules of Figure 1.2, and
- All rules of Figure 1.4, and
- The rules of Figure 1.3 leaving out the rule $[CBN]$ and taking $V_{t*} = V_t$ in the rule $[CBV]$.

The resulting evaluation function is called $\Downarrow^v$, and the evaluation strategy is referred to as simply (ordinary) CBV.

Note that the error-element $\boldsymbol{\varepsilon}$ effectively works like a *non-local control operator* equivalent to, *e.g.*, Felleisen's non-local abort operator $\mathcal{A}$ (see [FFKD87], [FH89].) Had we specified our operational semantics as a *context rewriting machine* (see the references just given for this notion), we could have compressed the complete propagational behaviour of the error element into just one rule, of the form

$$E[\boldsymbol{\varepsilon}] \mapsto \boldsymbol{\varepsilon}$$

where E ranges over an appropriate notion of CBV evaluation context. Compare with Wright's specification along these lines in [Wri94] (where the error element is called **check**.) $\Box$

We consider next non-standard Call-By-Value semantics.

Definition 1.6.2 (Non-errorstrict Call-By-Value, CBV_ε, $\Downarrow^{v\varepsilon}$)
The non-strict (wrt. the error element $\boldsymbol{\varepsilon}$) Call-By-Value evaluation results from treating $\boldsymbol{\varepsilon}$ as an ordinary value in the rule $[CBV]$ of Figure 1.3, and dropping the argument propagation rule $[W4]$ for $\boldsymbol{\varepsilon}$. In all detail, by CBV_ε we understand:

- The rules of Figure 1.2 restricting W to be $\boldsymbol{\omega}$ in the rule $[W4]$, and
- All rules of Figure 1.4, and
- The rules of Figure 1.3 leaving out the rule $[CBN]$ and taking $V_{t*} = V_{t\varepsilon}$ in the rule $[CBV]$.

The effect is that the error-element can be passed round as an argument to abstractions. Whenever it is used (as argument of a basic function or as an operator) it propagates. □

Finally, we consider standard Call-By-Name semantics.

Definition 1.6.3 (Call-By-Name, CBN, $\Downarrow^n$)
The usual Call-By-Name evaluation is obtained as follows:

- The rules of Figure 1.2, and restricting W to be $\boldsymbol{\omega}$ in the rule $[W4]$, and
- All rules of Figure 1.4, and
- The rules of Figure 1.3 leaving out the rule $[CBV]$.

Note that there is no interesting distinction between error-strict and non-errorstrict Call-By-Name evaluation. Hence, there is only one CBN variant, which is the usual one. □

Definition 1.6.4 (Pure semantics, $\Downarrow^{*p}$)
The pure evaluation functions are obtained by considering the value class

$$V_{pure} ::= V_t \mid \boldsymbol{\omega}$$

and taking just the rules of Figure 1.2 and Figure 1.3, reading just $\boldsymbol{\omega}$ for W in the former rules, and V_{pure} for V_p , V_t for V in the latter rules. With these conventions, we can define standard pure CBN and CBV semantics by choosing one of the rules $[CBN]$ or $[CBV]$ from Figure 1.3 as usual. These semantics are called $\Downarrow^{*p}$ where $*$ can be either n of v, according to whether we have chosen a CBN or a CBV semantics. □

1.6.2 Relating completions

Below we state some fundamental properties of completion evaluation wrt. operational semantics. The properties center around comparing the operational behaviour of congruent completions. All claims are sufficiently close to well known properties that we can omit proofs. See in general [WF91], [Wri94], and in particular, we refer the reader to [And94], where the properties are proven for a very closely related semantics, as explained above. In addition, we refer to [Hen92].

We wish to emphasize the following negative facts about soundness, to be stated in Property 1.6.10 below:

- The equational theory E is *not* sound wrt. ordinary CBV semantics.
- ψ conversion is *not* sound wrt. any semantics considered here.

Hence, *no semantics considered satisfy unrestricted coherence* (see [Hen94] for further information.)

However, as already mentioned, there is a form of Call-By-Value evaluation, albeit non-standard, which does satisfy soundness properties wrt. ϕ conversion. This is CVB_ε as defined above (see Property 1.6.11.)

We emphasize also that (standard) CBN is sound for ϕ conversion (see Property 1.6.10). As explained in [Hen94], the strategy wrt. failure of coherence is to restrict ourselves to computing in the ϕ-equivalence classes of *safe* completions.

To state the properties we need some definitions. First, we must introduce a typing rule for the error element in order to state the *type soundness* property:

Definition 1.6.5 (Error typing, $\vdash_\varepsilon$)
Let the typing relation $\vdash_\varepsilon$ be given by adding the following rule to **D** :

$$\Gamma \vdash_\varepsilon \boldsymbol{\varepsilon} : \tau$$

So the error element $\boldsymbol{\varepsilon}$ has any type. Note that there is *no* type τ such that $\Gamma \vdash_\varepsilon \boldsymbol{\omega} : \tau$. □

Second, we introduce the notion of *operational safety*, the semantic counterpart to the formal notion of safety of [Hen94]. We are obviously interested in the relation between these two notions of safety:

Definition 1.6.6 (Operational safety)
Define the partial order $\sqsubseteq$ on completions wrt. a given evaluation relation $\Downarrow$, by:

$$M \sqsubseteq M'$$

if and only if $M \cong M'$ and for every context C it holds that

$$C[M] \Downarrow \boldsymbol{\varepsilon} \text{ implies } C[M'] \Downarrow \boldsymbol{\varepsilon}$$

We say that a completion M is *operationally safe* if and only if

$$M \cong M' \text{ implies } M \sqsubseteq M'$$

In other words, a completion is operationally safe, if it does not generate any avoidable type errors in any context. □

Let $\Downarrow$ denote either one of the relations $\Downarrow^v$, $\Downarrow^{v\varepsilon}$ or $\Downarrow^n$. Then

Property 1.6.7 (*Functionality.*)
$\Downarrow$ is a partial function $\Downarrow$: *Programs* $\rightarrow V$. In particular, if $M \Downarrow V$ and $M \Downarrow V'$ then $V \equiv V'$. If M is undefined, then M does not terminate. □

Property 1.6.8 (*Type Soundness.*)
If $\vdash_\varepsilon M : \tau$ then either

(*i*) M diverges, or

(*ii*) $M \Downarrow \boldsymbol{\varepsilon}$, or

(*iii*) $M \Downarrow V$ with $\vdash_\varepsilon V : \tau$.

In particular, a well typed completion cannot produce the *wrong* element $\boldsymbol{\omega}$. Hence, "well typed completions cannot go wrong".

Note that one would have a more comprehensive notion of *wrongness* in a *pure typed* framework without explicit representation of run-time type operations, where stuck situations ($\boldsymbol{\omega}$) *as well as* type error-situations ($\boldsymbol{\varepsilon}$) would be considered *wrong.* In such a framework one would usually prove the stronger property that "well typed programs cannot go wrong", meaning that such programs can generate *neither* run-time type errors *nor* stuck states. In our case this reduces to type soundness of the simply typed λ-calculus, which is well known to hold. □

Property 1.6.9 (*Soundness of $\phi\psi$ conversion wrt. $\boldsymbol{\omega}$.*)
Let $\Downarrow^*$ be either $\Downarrow^n$ or $\Downarrow^v$. Then $M \cong M'$ and $M \Downarrow^* V \not\equiv \boldsymbol{\varepsilon}$ together imply that there exists a value V' such that $M' \Downarrow^* V'$ with $V \cong V'$ or $V' \equiv \boldsymbol{\varepsilon}$.

As noted in [And94], the property does not hold for $\Downarrow^{v\varepsilon}$. As an example (taken from [And94]) consider

$$M \equiv (\lambda x : \mathbf{D}.\lambda y : \mathbf{D}.x)([\mathtt{F!} \circ \mathtt{F?} \circ \mathtt{B!}]\mathit{true})$$

and

$$M' \equiv (\lambda x : \mathbf{D}.\lambda y : \mathbf{D}.x)([\mathtt{B!}]\mathit{true})$$

where one has $M \Downarrow^{v\varepsilon} \lambda y : \mathbf{D}.\boldsymbol{\varepsilon}$ but $M' \Downarrow^{v\varepsilon} \lambda y : \mathbf{D}.[\mathtt{B!}]\mathit{true}$.

It is also noted, that for $V \equiv \boldsymbol{\varepsilon}$ one cannot draw any interesting conclusions about the evaluation of M'; for instance M' might diverge under those conditions. $\square$

Property 1.6.10 (*Soundness of ϕ conversion wrt. CBN.*)
If $M =_\phi M'$ and $M \Downarrow^n V$, then there exists V' such that $M' \Downarrow^n V'$ with $V \cong V'$.

Note that this implies in particular that the equational theory E of $\mathbf{D}$ completions is sound wrt. CBN.

Note also (as in [And94]) that *the equational theory E is not sound wrt. CBV*, even when restricted to *operationally safe* completions. A good example (taken from [And94]) is the completion

$$M \equiv (\lambda x : \mathbf{D}.(\mathbf{if}\ \mathit{true}\ ([\mathtt{B!}]\mathit{true})\ (([\mathtt{F?}]x)([\mathtt{B!}]\mathit{true})))) \ ([\mathtt{B!}]\mathit{true})$$

which is equal to

$$M' \equiv \lambda x : \mathbf{D} \to \mathbf{D}.(\mathbf{if}\ \mathit{true}\ ([\mathtt{B!}]\mathit{true})\ (x([\mathtt{B!}]\mathit{true})))) \ ([\mathtt{F?} \circ \mathtt{B!}]\mathit{true})$$

One has $M \Downarrow^v [\mathtt{B!}]\mathit{true}$ but $M' \Downarrow^v \boldsymbol{\varepsilon}$.

Finally, note that *ψ conversion is not sound wrt. any of the semantics considered here*, due to the "non-algebraic" nature of coercion evaluation, as mentioned earlier. See [Hen94] for further information. $\square$

Property 1.6.11 (*Weak soundness of ϕ conversion wrt. CBV_ε*)
Following [And94], write $M \approx M'$ if and only if $M \cong M'$ with the exception that whenever the erasure of M or M' has $\boldsymbol{\varepsilon}$ as a subexpression, the other may have anything else. Then it holds:

If $M =_\phi M'$ and $M \Downarrow^{v\varepsilon} V$ then $M' \Downarrow^{v\varepsilon} V'$ with $V \approx V'$ and there exists a context C such that $C[V] \Downarrow^{v\varepsilon} V''$ and $C[V'] \Downarrow^{v\varepsilon} V''$.

See [And94] for this property. To see that the weakening by $\approx$ is necessary, consider the completions

$$M \equiv (\lambda x : \mathbf{D}.\lambda y : \mathbf{B}.(\mathbf{if}\ y\ ([\mathtt{B!}]\mathit{true})\ (([\mathtt{F?}]x)([\mathtt{B!}]\mathit{true}))))([\mathtt{B!}]\mathit{true})$$

and

$$M' \equiv (\lambda x : \mathbf{D} \to \mathbf{D}.\lambda y : \mathbf{B}.(\mathbf{if}\ y\ ([\mathtt{B!}]\mathit{true})\ (x([\mathtt{B!}]\mathit{true}))))([\mathtt{F?} \circ \mathtt{B!}]\mathit{true})$$

Then we have $E \vdash M = M'$, but

$$M \Downarrow^{v\varepsilon} \lambda y : \mathbf{B}.(\mathbf{if}\ y\ ([\mathtt{B!}]\mathit{true})\ (([\mathtt{F?} \circ \mathtt{B!}]x)([\mathtt{B!}]\mathit{true})))$$

and

$$M' \Downarrow^{v\varepsilon} \lambda y : \mathbf{B}.(\mathbf{if}\ y\ ([\mathtt{B!}]\mathit{true})\ (\boldsymbol{\varepsilon}\ ([\mathtt{B!}]\mathit{true})))$$

$\square$

Property 1.6.12 (*Operational safety of formally safe completions.*)
If $M \Rightarrow^*_{\psi/\phi} M'$ then $M \sqsupseteq M'$.

For this property, see [Hen92]. □

Property 1.6.13 (*Operational safety of canonical completions.*)
The canonical completion $[\![M]\!]$ is operationally safe, for any pure term M.

See [Hen92]. □

1.7 Purpose of the thesis

The general purpose of the present thesis can be summarized in the following tasks:

1. To study the proof theoretic aspects of dynamic typing in order to see how far the methods introduced in [Hen94] can be pushed, preferably with solutions to the major open problems stated therein as part of the outcome.
2. To generalize the monomorphic framework of Henglein's dynamically typed λ calculus to include polymorphism.
3. To develop completion inference methods for a generalized framework of polymorphic dynamic typing and to study their behaviour in practice.
4. To investigate *modular* dynamic typing for a *Scheme*-like language and the possibility of using dynamic typing as a basis for high-level compilation of a *Scheme*-like language to *ML*.

1.8 Overview of the thesis

The two first chapters of the main development, Chapter 2 and Chapter 3 investigate properties of Henglein's calculus **D**. Later, we define extensions of this framework which we believe should supersede **D**. However, we have generally adopted the policy of considering certain problems in **D**, although we wish to extend their solution to the general framework. We use **D** in this capacity as a core reference system, much like the simply typed λ-calculus, where many problems can be stated in a simple manner and essential techniques can be presented in a minimal framework. The techniques and results given in Chapter 2 will be called upon later in the work.

Chapter 2 is concerned with the problem of confluence of ϕ-reduction in Henglein's calculus **D**. We introduce the notion of coercion polarity and study coercion factorization. This development provides some technical background to the main development, which is contained in the second section of the chapter. Here we introduce the notion of polarized ϕ-reduction which was defined in [Hen94] and conjectured to be confluent and terminating. We show that termination fails, but that confluence holds. Our confluence proof is an extended study of commutation properties of dynamically typed λ-calculus. The chapter should be considered an in-depth reference for rewrite analyses of extensions of **D**, which are considered later in the work.

Chapter 3 deals with a natural subcalculus of **D**, which can be seen as a generalized form of quasi-static typing. We show that all desirable fundamental properties (confluence, termination, safety and coherence) can relatively easily be established for this system. We observe, however, that almost none of the techniques employed for the subcalculus will work for non-restricted systems like **D**. This is the main motivation for our study in this chapter. The chapter is relatively short, and it is probably the least important of the present work.

Chapter 4 begins the extension of Henglein's calculus **D**, which will eventually result in a generalized calculus of dynamic typing including regular recursive types and polymorphic, discriminative sum types. We like to think of this system as "the least upper bound" of Henglein's dynamic typing and certain systems of soft typing. Guided by an equational presentation of categorical co-products, we observe that the introduction of an *error*-type with explicit error-coercions leads to a system in which generalized forms of ϕ-reduction can naturally de defined. Out of this generalized basis we project a pragmatically oriented system and establish fundamental syntactic properties of it. The resulting system comes in two versions, one called Σ^{μ} and the other called Σ^{∞}; the former contains explicit type recursion coercions, whereas the latter is equivalent to an infinitary type system over regular trees. The latter is more powerful than the former, and the latter is the system we are inteested in working with in practice. Our interest in the former system is of a more technical nature, because it is sometimes convenient to be able to represent the structure of type recursion explicitly. We show that a universal type is definable in these systems and that **D** can therefore be embedded in them.

Chapter 5 is concerned with the problem of termination of completion minimization in **D** and Σ^{∞}. We note that the reason for non-termination of Henglein's original reduction system are concentrated in the use of so-called neutral coercions, which can be eliminated from Σ^{∞} due to the presence of explicit error coercions. We restate Henglein's termination conjecture for this system. Regrettably, though, we must give up on deciding the general termination problem. Nevertheless, this chapter contains perhaps the most important theoretical result of the thesis. This result says that, in **D** and in Σ^{∞}, one has existence and uniqueness of normal forms in a restricted class of completions, which is defined by requiring all coercions to be primitive, but these can be placed anywhere in a program. In the remainder of the thesis, we indicate in several places that this could be a practically significant result, since the class is both natural and extremely powerful. An analogous result is given for a further restricted class where coercions cannot be placed in arbitrary places.

Chapter 6 takes first steps towards practical completion inference based on the calculus Σ^{∞}. Following [Hen92] we introduce coercion parameterization and so-called coercive types for polymorphic dynamic inference. The result is a form of qualified polymorphic type system. We specify a constraint based completion inference algorithm, a descendant of Henglein's algorithm [Hen91] for higher order binding time analysis. We formulate and prove correctness properties for this algorithm. We describe an implementation of the algorithm which works for a core functional subset of a *Scheme*-like language. We observe that, for reasons of operational safety, the inference system is inherently a global analysis, since it cannot safely be used in a compositional manner. It is argued that the use of coercion parameters is an important means of balancing conflicting demands of safety and minimality. We give examples of the algorithm at work, with a focus on determining the limitations of the current implementation. We suggest extensions which would be desirable and practical. Finally, we give an extended comparison of the inference framework of this chapter and systems of soft typing.

Chapter 7 has two parallel topics. One is the extension of polymorphic dynamic typing to achieve fully modular completion inference for *Scheme*. The other is an investigation of the possibility of using our framework of dynamic typing to implement a high-level compilation of *Scheme* to *ML*. We observe that a stronger form of safety condition is needed for modular inference, and we give its definition. We identify a form of neededness analysis to achieve minimization under this strong notion of safety. Two closely related completion languages are defined, *Dynamically Typed Scheme* and *ML-Scheme*, the former being intended to support a type system for *Scheme* for static debugging purposes; the latter to support compilation to *ML*. It is argued that these two enterprises may not share common goals and problems, hence the definition of two variants of the completion language. The main bulk of the chapter is concerned with identifying major problems that extension to full *Scheme* will meet, both regarding type inference and regarding translation to *ML*.

Chapter 8 concludes the thesis with a summary, discussion of related work and future work.

Chapter 2

Confluence of ϕ reduction

This chapter studies the problem of confluence of ϕ-reduction in system **D**. The most important part of the chapter is the second section, so we begin by introducing that section first. We introduce (*second section*) the problem of confluence in dynamically typed λ-calculus and prove that ϕ is confluent over λI, confirming a conjecture of [Hen94]. The proof of the second section is illuminating and is given in great detail. The techniques used are important and will be appealed to later in this report. The reader who wishes to appreciate the details concerning confluence properties here and later must read the second section of the present chapter somewhat carefully and remember (at least so as to be able to refer to) the main techniques used. The central insight of the proof is due to Henglein [Hen94] who introduced the critical notion of *coercion polarity.* This notion really explains *why* the calculus is confluent. The main idea is define an *orientation* of ϕ-reduction based on an orientation of the equations in E. This, in turn, is based on the notion of polarity. The oriented ϕ-reduction is therefore referred to as *polarized reduction.* Such a polarized system was defined in [Hen94] and shown to be locally confluent. It was conjectured in [Hen94] that the system is also terminating. We show (also second section) that this is unfortunately not true. However, the oriented system is still useful for proving confluence; only, instead of relying on termination via Newman's Lemma, our proof becomes an extended exercise in the Hindley Rosen Lemma, relying on close analyses of commutation properties. The fact that the proof essentially relies on a single, very clear idea makes it robust wrt. certain variations and extensions of the rules, and therefore we are able to call upon the insights of the proof on later occasions without having to redo all of the analysis.

It turns out that the confluence proof given here can be simplified if we extend dynamic typing to include explicit error-coercions and an error-type. We shall do this later in this report. However, we give the more general proof here, since, apart from showing that such extension is not strictly needed for confluence, the rewrite analyses given here are interesting in their own right. The analyses constitute an exhaustive reference for commutation properties of a dynamic type calculus.

In the *first section* of this chapter, we introduce the notion of polarity. We then develop a small theory of *prime coercions.* This notion, we believe, is of independent interest, since it identifies a quite natural thing, namely the idea of a *primitive* (or prime) *proof step*, viewing coercions a term encodings of typing proof steps. Moreover, the some of the consequences derived here are important for the confluence proof of the second section. In particular, that proof relies on a somewhat technical but very important property about coercion polarity and factorization of sets of coercions. The property (called Theorem 2.1.13 here) was proven in [Hen94] (called Theorem 27 there.) However, the proof given there is not so detailed, and it relies on a somewhat involved rewrite analysis for confluence of an auxiliary rewrite system. In this section, we try to give a more detailed and robust proof, relying on induction instead of rewrite analysis [1] It is technically quite involved, though, and a full understanding of the details is not a prerequisite for the remainder of the report. Therefore, main technical proofs are placed in Appendix B.1.

2.1 Coercion factorization

In this section we first introduce the important notion of *coercion polarity*, as defined by Henglein [Hen94]. We then develop a small theory of *prime* coercions under C-equality. The main result is a prime factorization theorem stating that every coercion has a prime factorization (w.r.t. $\circ$) which is unique up to a certain restricted form of permutation of the prime factors. One consequence we can draw from this theorem is that sets of equal coercions can be given common left and right factors of equal polarity. This property is crucial for our proof of confluence of ϕ reduction on completions, to be given later.

If nothing to the contrary is said, equality ($=$) between coercions denotes C-equality in this section.

2.1.1 Polarity

The notion of coercion polarity classifies coercions as being *positive* ($+$), *negative* ($-$) or *neutral* ($*$). The assignment of polarity to coercions is defined in Figure 2.1. The rules extend the assignment of positive polarity to tagging coercions, negative polarity to check-and-untag coercions, and neutral polarity to "error-coercions" from the primitive base cases to induced coercions, by induction. The "error-coercions" arise from compositions of the form `tc'?` $\circ$ `tc!` with `tc` $\not\equiv$ `tc'`, which model the situation where a run-time type error is present. Note that **D** contains no reduction for neutral coercions; error-situations are represented as "stuck states" wrt. reduction in **D**. Note also that a neutral coercion cannot interact under ϕ-reduction with any neighbouring coercion in composition. Not every coercion has a defined polarity. In particular, a ϕ-redex `tc?` $\circ$ `tc!` does not have a polarity, neither $+$, $-$ or $*$. Polarity is invariant under C-equality, with the exception that a coercion which has no polarity may be reassociated into one which has a neutral (but not negative and not positive) polarity, or vice versa. An example is $\texttt{B?} \circ (\texttt{F!} \circ (\texttt{F?} \rightarrow \texttt{F!}))$ which has the structure of polarity $- \circ (+ \circ +)$ and which is therefore neither positive, negative nor neutral. By reassociation to the left, as $(\texttt{B?} \circ \texttt{F!}) \circ (\texttt{F?} \rightarrow \texttt{F!})$, one gets the polarity structure $(- \circ +) \circ +$ which is neutral. We can eliminate this lack of invariance by adding that $c \circ (c' \circ c'')$ is neutral if $(c \circ c') \circ c''$ is, and vice versa. However, this will not be of importance for our development. A coercion c such that $C \vdash c = id$ is called *trivial.* A coercion c with positive (negative) polarity is called *properly* positive (negative) if c is positive (negative) and not trivial.

As will be seen in the next section, every coercion *factors*, under C-equality, into coercions each of which has a well defined polarity, either positive or negative. Also, it will be seen later that there is an interesting property about factorization of coercions into positive, neutral and

[1] Confluence proofs by case analysis of the rules of a rewrite system are sometimes "vulnerable". In particular, a case analysis logically must contain, at the end, a claim to the effect that *all* cases have been considered. This is normally (if the problem is complicated enough) just taken to mean that all *interesting* cases have been considered, since it is too burdensome to consider the entire cartesian product of the rules for analysis of all overlaps in search of critical pairs. However, exhaustiveness of the analysis may sometimes (depending on the nature of the system) be hard to verify, and faults may easily creep into such a proof.

negative composants under ϕ-reduction. These properties will become very important in the proof of confluence, in Section 2.2.

$$\mathtt{F!} : + \qquad \mathtt{F?} : - \qquad \mathtt{F?} \circ \mathtt{B!} : * \qquad \mathtt{B?} \circ \mathtt{F!} : *$$

$$\mathtt{B!} : + \qquad \mathtt{B?} : - \qquad \frac{c_1 : * \quad c_2 : -}{c_2 \circ c_1 : *}$$

$$\frac{c : + \qquad d : +}{c \circ d : +} \qquad \frac{c : - \qquad d : -}{c \circ d : -} \qquad \frac{c_1 : * \quad c_2 : +}{c_1 \circ c_2 : *}$$

$$\frac{c : - \qquad d : +}{c \to d : +} \qquad \frac{c : + \qquad d : -}{c \to d : -} \qquad \frac{c_1 : * \quad c_2 : *}{c_1 \circ c_2 : * \quad c_1 \to c_2 : *}$$

A positive (negative) coercion which is not trivial is said to be *properly* positive (negative). The identity coercions are both positive and negative, but not *properly* so: $id_\tau : +, -$

Figure 2.1: Coercion polarity

2.1.2 Prime coercion factors

First we define the notion of prime coercions:

Definition 2.1.1 (Prime coercion)
Let p' range over the set $\{c \mid c \text{ is primitive}, c \not\equiv id\}$. A coercion is called *prime* if it is generated by π where:

$$\begin{array}{lcl} \pi & ::= & \pi' \mid id \\ \pi' & ::= & p' \mid \pi' \to id \mid id \to \pi' \end{array}$$

Note that for every π' one has $\pi' \neq id$. □

We introduce some technical notions to talk about sequences of coercions. In general, we write sequences of elements as $< e_1, \dots e_n >$, we use :: to denote concatenation of sequences, we write $<>$ for the empty sequence (the neutral element for ::), and if $s =< e_1, \dots e_n >$ and $1 \leq i \leq n$ we let $s_i = e_i$.

Definition 2.1.2 Let s be a non-empty sequence of coercions. Then the operation $s \setminus ID$ as given below throws away *trivial* elements from a sequence of coercions (recall that a coercion c is called trivial iff $C \vdash c = id$)

$$s \setminus ID \;=\; \begin{cases} < id > & \text{if every coercion in } s \text{ is trivial} \\ s' & \text{otherwise, where } s' \text{ is the result of deleting} \\ & \text{all trivial coercions from } s. \end{cases}$$

If $s =< c_1, \dots, c_n >$ is a non-empty sequence of coercions then the operation $[\circ]s$ is defined as

$$\begin{array}{lcl} [\circ] < c > & \equiv & c \\ [\circ]\,(< c >:: s) & \equiv & (c \circ [\circ]\, s) \end{array}$$

So $[\circ]\, s$ is the right-associative normalized composition of all the coercions in s.

If d is a coercion and s a non-empty sequence of coercions, $s = < c_1, \ldots, c_n >$, we define the operations $s \Rightarrow d$ and $d \Rightarrow s$ by

$$\begin{array}{lcl} s \Rightarrow d & = & < c_1 \to d, \ldots, c_n \to d > \\ d \Rightarrow s & = & < d \to c_1, \ldots, d \to c_n > \end{array}$$

□

If $s = < c_1, \ldots, c_n >$ is a sequence of coercions, we let S^R denote the reversed sequence, $s^R = < c_n, \ldots, c_1 >$.

We now define a prime factoring function. It is a function from coercion *terms* to sequences of coercions.

Definition 2.1.3 (Prime factoring function)
Let p range over primitive coercions. Then define the prime factoring function π by

$$\begin{array}{lcl} \pi(c) & = & \rho(c) \setminus ID \text{ , where} \\ \rho(p) & = & < p > \\ \rho((c \circ d)) & = & \rho(c) :: \rho(d) \\ \rho(c \to d) & = & (\rho(c)^R \Rightarrow id) :: (id \Rightarrow \rho(d)) \end{array}$$

We define

$$\pi^\circ(c) \equiv [\circ]\,(\pi(c))$$

So $\pi(c)$ is a sequence of coercions, and $\pi^\circ(c)$ is a coercion composed by factors of such a sequence.
□

Lemma 2.1.4 *$\pi(c)$ is a nonempty sequence of prime coercions, for every c.*

Proof By structural induction on c. □

Lemma 2.1.5 *For every coercion sequence s,*

1. $C \vdash [\circ]\,(s^R \Rightarrow id) = ([\circ]\, s) \to id$
2. $C \vdash [\circ]\,(id \Rightarrow s) = id \to ([\circ]\, s)$

Proof It is easy to see that

$$([\circ] < c_1, \ldots, c_{n-1} >) \circ c_n = [\circ] < c_1, \ldots, c_n >$$

by reassociation. We now prove the claims by induction on the length n of $s = < c_1, \ldots, c_n >$. We only go through the proof of the first part of the claim, since the proof of the second part is analogous. The base case is trivial, so consider the inductive step. We have

$$\begin{array}{lll} [\circ]\,(s^R \Rightarrow id) & \equiv & \\ [\circ]\,(< c_1, \ldots, c_n >^R \Rightarrow id) & \equiv & \\ [\circ]\,(< c_n, \ldots, c_1 > \Rightarrow id) & \equiv & \\ [\circ] < c_n \to id, \ldots, c_1 \to id > & \equiv & \\ (c_n \to id) \circ [\circ] < c_{n-1} \to id, \ldots, c_1 \to id > & \equiv & \\ (c_n \to id) \circ [\circ] < c_1 \to id, \ldots, c_{n-1} \to id >^R & \equiv & \\ (c_n \to id) \circ [\circ] < c_1, \ldots, c_{n-1} >^R \Rightarrow id & = & \text{induction} \\ (c_n \to id) \circ (([\circ] < c_1, \ldots, c_{n-1} >) \to id) & = & \\ (([\circ] < c_1, \ldots, c_{n-1} >) \circ c_n) \to id & = & \\ ([\circ] < c_1, \ldots, c_n >) \to id & \equiv & \\ ([\circ]\, s) \to id & & \end{array}$$

□

Proposition 2.1.6 $C \vdash c = \pi^\circ(c)$

PROOF Clearly, the claim is entailed by

$$C \vdash c = [\circ]\,\rho(c)$$

To see this latter claim, we proceed by structural induction on c. Only the case where $c \equiv c_1 \to c_2$ is interesting and considered here. We use the previous lemma, as follows.

$$\begin{array}{lll}
[\circ]\,\rho(c_1 \to c_2) & \equiv & \\
[\circ]\,((\rho(c_1)^R \Rightarrow id) :: (id \Rightarrow \rho(c_2))) & = & \\
[\circ]\,(\rho(c_1)^R \Rightarrow id) \circ [\circ]\,(id \Rightarrow \rho(c_2)) & = & \text{Lemma 2.1.5} \\
(([\circ]\,\rho(c_1)) \to id) \circ (id \to ([\circ]\,\rho(c_2))) & = & \text{induction} \\
(c_1 \to id) \circ (id \to c_2) & = & \\
c_1 \to c_2 & &
\end{array}$$

□

Corollary 2.1.7 *For every coercion c, there is a factoring*

$$C \vdash c = c_1 \circ \ldots \circ c_n$$

where each c_i has a well defined polarity, either + or −.

PROOF Take $< c_1, \ldots, c_n >= \pi(c)$; clearly, every prime factor c_i in $\pi(c)$ is either positive or negative. Now apply Proposition 2.1.6. □

2.1.3 Legal permutations and factorization

Definition 2.1.8 Let $s =< c_1, \ldots, c_n >$ be a sequence of coercions, and let σ be a permutation in S_n. Then we denote by $\sigma(s)$ the permuted sequence $\sigma(s) =< c_{\sigma(1)}, \ldots, c_{\sigma(n)} >$. Here it is understood that the identity coercion id does not carry a type-signature as part of its identity. So for instance, we have $< id_\tau \to p, p >= \sigma(< p, id_{\tau'} \to p >)$ with $\sigma = \{1 \mapsto 2, 2 \mapsto 1\}$, although $\tau \not\equiv \tau'$.

If s and σ are as above and moreover

$$C \vdash [\circ]\,s = [\circ]\,\sigma(s)$$

then σ is called a *legal permutation* (for s.) □

Example 2.1.9 π° gives *one* prime factoring out of several possible. In general, prime factorings are unique only up to associativity and legal permutation. As an example, consider that

$$(id_{\mathbf{D}\to\mathbf{D}} \to \texttt{F?}) \circ (\texttt{F!} \to id_{\mathbf{D}}) = (\texttt{F!} \to id_{\mathbf{D}\to\mathbf{D}}) \circ (id_{\mathbf{D}} \to \texttt{F?})$$

The coercion on the left factors through the type $(\mathbf{D} \to \mathbf{D}) \to \mathbf{D}$, whereas the one on the right factors through $\mathbf{D} \to (\mathbf{D} \to \mathbf{D})$. We do, however, consider $< id_{\mathbf{D}\to\mathbf{D}} \to \texttt{F?}, \texttt{F!} \to id_{\mathbf{D}} >$ a (legal) permutation of $< \texttt{F!} \to id_{\mathbf{D}\to\mathbf{D}}, id_{\mathbf{D}} \to \texttt{F?} >$, as will be recalled from Definition 2.1.8. Both compositions are prime factorizations of the coercion $\texttt{F!} \to \texttt{F?}$. It is clear that the *only* source of legal permutativity for prime factorizations resides in the possibility of letting identity-coercions in an arrow "commute" with other factors, like in this example.

Note also that the order of polarities may change with legal permutations. For instance, one has

$$\texttt{F!} \to \texttt{F!} = (id \to \texttt{F!}) \circ (\texttt{F!} \to id) = (\texttt{F!} \to id) \circ (id \to \texttt{F!})$$

where the first factorization is $+ \circ -$ and the second is $- \circ +$. □

We have considered legal permutations above. It is an elementary fact that permutations can be factored into *transpositions* (2-cycles.) It is natural to ask whether a *legal* permutation can be factored into *legal* transpositions. As we shall now show, it can, and from this result we can draw a very important consequence concerning common factors of sets of coercions. First we introduce a few technical concepts.

Definition 2.1.10 If two coercions c_1, c_2 satisfy $C \vdash c_1 \circ c_2 = c_2 \circ c_1$ then we say that c_1 and c_2 *commute.*

Define for arbitrary coercion sequences s, s'

$$s \mapsto s'$$

if and only if $s = s_1 ::< c, d >:: s_2$ and $s' = s_1 ::< d, c >:: s_2$, for some s_1, s_2, c, d where c and d commute. Let $\sim$ denote the reflexive, symmetric, transitive closure of $\mapsto$ on coercion sequences, $\mapsto^*$ its reflexive, transitive closure. □

Lemma 2.1.11 *One has*

1. $(s_1 \Rightarrow id) :: (id \Rightarrow s_2) \sim (id \Rightarrow s_2) :: (s_1 \Rightarrow id)$
2. $s_1 \sim s_2$ *implies* $s_1 \Rightarrow id \sim s_2 \Rightarrow id$ *and* $id \Rightarrow s_1 \sim id \Rightarrow s_2$
3. $s_1 \sim s_2$ *if and only if* $s_1^R \sim s_2^R$

Proof The first property follows from axiom $[C4]$ together with the fact that the identities are neutral elements for the composition.

To see the second claim, consider that commutativity of c and d entails $< c \to id, d \to id > \sim < d \to id, c \to id >$, that is, $c \to id$ and $d \to id$ also commute. From this it is seen that, if c and d commute, then $< c, d > \Rightarrow id \sim < d, c > \Rightarrow id$.

Now consider the third claim. Suppose

$$s_1 = s_1' ::< c_1, c_2 >:: s_1'' \mapsto s_1' ::< c_2, c_1 >:: s_1'' = s_2$$

Then, since $\mapsto$ is symmetric, we have

$$s_1^R = (s_1'')^R ::< c_2, c_1 >:: (s_1')^R \mapsto (s_1'')^R ::< c_1, c_2 >:: (s_1')^R = s_2^R$$

from which the claim follows. □

Lemma 2.1.12 $C \vdash c = c'$ *if and only if* $\pi(c) \sim \pi(c')$.

PROOF By induction, using previous lemmas. The proof is given in Appendix B.1. □

We are now ready to state the proof of the main property about factorization. The property is proven, using other methods, in [Hen94], Theorem 27. The proof given here is more detailed, and it relies on induction rather than rewrite analysis, as was the method used in [Hen94]. A key technical ingredient is Lemma 2.1.12 which allows us to proceed by induction in the length of a prime factorization and by cases over the justification of $\sim$. It turns out to be surprisingly difficult to give a completely detailed proof of the desired property. Our proof is given in Appendix B.1. The theorem below is used in the proof of Proposition 2.2.2.

Theorem 2.1.13 *(C-Factoring)*
Let $\diamond$ be any fixed polarity, $+$ or $-$. Suppose either

1. $C \vdash < c_1, \ldots c_n > = < h^\diamond \circ c'_1, \ldots, h^\diamond \circ c'_n > = < g^\diamond \circ c''_1, \ldots, g^\diamond \circ c''_n >$ *, or*
2. $C \vdash < c_1, \ldots, c_n > = < c'_1 \circ h^\diamond, \ldots, c'_n \circ h^\diamond > = < c''_1 \circ g^\diamond, \ldots, c''_n \circ g^\diamond >$

Then there exist coercions $\tilde{c}_1, \ldots, \tilde{c}_n, \tilde{h}^\diamond, \tilde{g}^\diamond$ such that if 1. *is the case then*

i $C \vdash < c'_1, \ldots, c'_n > = < \tilde{h}^\diamond \circ \tilde{c}_1, \ldots, \tilde{h}^\diamond \circ \tilde{c}_n >$
ii $C \vdash < c''_1, \ldots, c''_n > = < \tilde{g}^\diamond \circ \tilde{c}_1, \ldots, \tilde{g}^\diamond \circ \tilde{c}_n >$
iii $C \vdash h^\diamond \circ \tilde{h}^\diamond = g^\diamond \circ \tilde{g}^\diamond$

and if 2. *is the case then*

i $C \vdash < c'_1, \ldots, c'_n > = < \tilde{c}_1 \circ \tilde{h}^\diamond, \ldots, \tilde{c}_n \circ \tilde{h}^\diamond >$
ii $C \vdash < c''_1, \ldots, c''_n > = < \tilde{c}_1 \circ \tilde{g}^\diamond, \ldots, \tilde{c}_n \circ \tilde{g}^\diamond >$
iii $C \vdash \tilde{h}^\diamond \circ h^\diamond = \tilde{g}^\diamond \circ g^\diamond$

PROOF See Appendix B. □

2.2 Confluence of ϕ reduction

This section is devoted to establishing the main result of the present chapter: We prove that Henglein's conjecture ([Hen94]) of confluence for ϕ reduction over λI-completions is correct. We feel that this result is both surprising (at least from a perspective prior to [Hen94]) and important. It is surpising, because it demonstrates that there is an interesting mathematical *order* in the process of completion. That such order should be there is, in our view, not at all obvious a priori. In fact, as is shown in the first subsection below, one can give evidence that appears to indicate the contrary, and one actually needs to impose a restriction to λI-completions (or, alternatively, a restriction of the reduction rules) for confluence to hold. However, the counterexamples producable in λK are "pathological" special cases which do not overshadow the fundamental point.

The confluence property is important, because it shows that the non-deterministic reduction defines a meaningful notion of deterministic static computation over type coercions in completions. As is well-known (see. *e.g.*, [Bar84] and [HS90]) confluence is equivalent to the Church-Rosser property for the equational theory generated from the reduction, and it implies uniqueness of normal forms. Apart from the fundamental nature of these properties (which makes them interesting by themselves) they can be very useful in practice. For instance, we often want to know whether a certain completion is "optimal" or whether a certain reduction strategy is good; we may ask such questions in many contexts, *e.g.*, we might try to construct an algorithm for minimization based on abstract, implementation independent considerations of a reduction system. In such context, we shall naturally want to know things like whether our strategy can be "fooled" into "stupid" lines of reduction, taking us away from another reduction path leading to a much better end result. Once we have established confluence, we shall know that, if our strategy terminates in some normal form, it will necessarily be unique, and therefore no better option could have been missed. The fact that we know that we can reason in terms of *local*, non-deterministic reduction steps is of great help in, say, algorithm construction. Another important application is that one can prove non-convertibility between two terms by reducing them to different normal forms. This can be useful, if we want to compare two completions.

In the following development we use that ϕ-reduction on coercions is canonical (shown in [Hen94].)

2.2.1 The problem

Let us begin by seeing why confluence for the completion reductions of dynamically typed λ-calculus presents a problem. The following negative results were established in [Hen94]:

- The congruence of $\mathbf{D}$ cannot be oriented to a confluent reduction (not even over λI)
- ϕ and ψ reduction do not commute (not even over λI)
- $\phi\psi$ reduction is not confluent (not even over λI.)
- ϕ reduction is *not* confluent over *non-λI-* completions

The counterexample to confluence for general λK-completions exploits the fact that rule $[E4]$ of the equational theory is pathological in case the bound variable of a λ-abstraction does not occur in the body; in this case, one can freely "invent" a coercion and pull it out of the abstraction, reading rule $[E4]$ from right to left. This is what happens in the counterexample of [Hen94]: Let M, M_1 and M_2 be completions given as follows

$M \equiv (\lambda x : \mathbf{D}.\mathit{true})\ (\mathbf{if}\ \mathit{false}\ ([\mathtt{F!}](\lambda y : \mathbf{D}.y))\ ([\mathtt{B!}]\mathit{true}))$

$M_1 \equiv (\lambda x : \mathbf{D} \to \mathbf{D}.\mathit{true})\ (\mathbf{if}\ \mathit{false}\ (\lambda y : \mathbf{D}.y)\ ([\mathtt{F?} \circ \mathtt{B!}]\mathit{true}))$

$M_2 \equiv (\lambda x : \mathbf{B}.\mathit{true})\ (\mathbf{if}\ \mathit{false}\ ([\mathtt{B?} \circ \mathtt{F!}](\lambda y : \mathbf{D}.y))\ \mathit{true})$

Then one can verify that $M \Rightarrow_\phi M_1$ and $M \Rightarrow_\phi M_2$, but there is no common reduct of M_1 and M_2.

It is consistent with these observations that ϕ reduction might be confluent over λI, leaving all the non-confluence cases to be of the pathological sort just illustrated. However, ϕ reduction is a relation *modulo* a highly non-trivial equational theory, E. It is therefore only to be expected that confluence could be complicated. General rewrite theoretical techniques for reductions *modulo* equational theories (*e.g.*, [Kir94], [DeJo90], and [BD86] which is a classic here) typically rely on very powerful properties of the equational system; in particular (and very understandably), a confluent *orientation* of (parts of) the equational system is usually involved. However, the theory E has no confluent orientation, and properties usually exploited in more specialized treatments (such as linearity properties, orthogonality etc.) are simply not satisfied in our case.

In fact, the non-orientability of the theory E makes it not at all intuitively obvious that confluence should hold, even disregarding the pathological λK-cases. Consider for example the major (unsolvable) critical pair for the left-to-right orientation $E^{\to}$:

$M \equiv ([c \to d](\lambda x.M))\ N$

$M_1 \equiv (\lambda x.[d]M\{x := [c]x\})\ N$

$M_2 \equiv [d]((\lambda x.M)\ ([c]N))$

One has $M \to M_1$, $M \to M_2$ with no common reduct of M_1 and M_2, under $E^{\to}$. The divergence here appears to be a fundamental one: Either the function or the argument gets coerced, and there seems to be little telling what could happen to those coercions inside the abstraction or inside the argument, respectively.

However, in [Hen94] it was suggested that one might obtain a tractable system by orienting the equations in E using a much deeper principle of orientation than just picking some direction (left-right, right-left.) The principle of orientation rests on the *polarity* of coercions, and the idea is to consider the orientation *together* with ϕ reduction, exploiting properties of coercion factorizations under ϕ reduction.

2.2.2 Polarized reductions

Recall from Section 2.1 the notions of negative, positive and neutral coercions. The central property of ϕ reduction in relation to polarity is given in the following factorization lemma, proved in [Hen94]:

Lemma 2.2.1 *(Henglein)*
In $\mathbf{D}$*, every coercion* ϕ*-reduces to a coercion of the form* $c^+ \circ c^* \circ c^-$.

PROOF See [Hen94] □

Recall also the factorization theorem, Theorem 2.1.13, which is the main ingredient for controlling the equations in C. The idea then is to consider the *polarized reduction* $P_\phi*$, shown in Figure 2.2 at the end of this chapter [2]

[2]The reader should inspect the definition of this reduction relation before continuing.

Indeed, as was also shown in [Hen94], this relation is *locally confluent*. We repeat the proof in some details here, since the insights provided by it are crucial for our later development:

Proposition 2.2.2 *(Henglein)*
The polarized ϕ-reduction $P_\phi*$ is locally confluent.

PROOF We consider possible critical pairs obtained by superposition of
I. ϕ-rules and P-rules
II. ϕ-rules and ϕ-rules
III. P-rules and P-rules.

Case I. We consider a coercion application $[c]N$. We can assume that c is the only coercion at this "point", since that can always be obtained by use of $[E3]$. A P-rule applies only at the point $[c]N$ if $C \vdash c = c' \circ c^-$ or $C \vdash c = c^+ \circ c''$ where c^+ is properly positive and c^- properly negative. If a ϕ-rule is applicable in c then it can only be applicable within c', respectively, c''. Therefore a PF-rule and a ϕ-rule cannot give rise to a critical pair.

Case II. Critical pairs generated from applying two ϕ-rules are solvable by confluence of ϕ ([Hen94])

Case III. We can have two kinds of situation.

(1) The same P-rule is applied at the same point, only with different negative right, respectively, positive left factors. For example, with $M \equiv (\textbf{if}\ N_1\ ([c]N_2)\ ([d]N_3))$ we may have

$$M \Rightarrow_P [h^+](\textbf{if}\ N_1\ ([c']N_2)\ ([d']N_3)) \equiv M_1$$

and

$$M \Rightarrow_P [g^+](\textbf{if}\ N_1\ ([c'']N_2)\ ([d'']N_3)) \equiv M_2$$

because

$$C \vdash < c, d > = < h^+ \circ c', h^+ \circ d' > = < g^+ \circ c'', g^+ \circ d'' >$$

But then, by C-factoring (Theorem 2.1.13), we know that there exist coercions $\tilde{c}, \tilde{d}, \tilde{h}^+, \tilde{g}^+$ such that

i $C \vdash < c', d' > = < \tilde{h}^+ \circ \tilde{c}, \tilde{h}^+ \circ \tilde{d} >$

ii $C \vdash < c'', d'' > = < \tilde{g}^+ \circ \tilde{c}, \tilde{g}^+ \circ \tilde{d} >$

iii $C \vdash h^+ \circ \tilde{h}^+ = g^+ \circ \tilde{g}^+$

Therefore we have, by (i)

$$M_1 \Rightarrow_P [h^+][\tilde{h}^+](\textbf{if}\ N_1\ ([\tilde{c}]N_2)\ ([\tilde{d}]N_2)) \equiv M_1'$$

and, by (ii)

$$M_2 \Rightarrow_P [g^+][\tilde{g}^+](\textbf{if}\ N_1\ ([\tilde{c}]N_2)\ ([\tilde{d}]N_2)) \equiv M_2'$$

and so, by (iii), we see that $M_1' = M_2'$, a common reduct of M_1 and M_2. All critical pairs arising in this way can be solved in analogous manner.

(2) Two different P-rules are applied at the same point. This is possible for the pairs of rules $([P1^-], [P1^+])$ and $([P2^-], [P2^+])$, respectively. We consider just an example of former the type, the latter being analogous. With $M \equiv [id_\tau \to c^-](\lambda x : \tau.[c^+]N)$ we have

$$M \Rightarrow_P \lambda x : \tau.[c^- \circ c^+]N \equiv M_1$$

and

$$M \Rightarrow_P [id_\tau \to (c^- \circ c^+)](\lambda x : \tau.N) \equiv M_2$$

Now, by $+ \circ * \circ -$ Factoring (Proposition 2.2.2), there are a^+, a^*, a^- such that $c^- \circ c^+ \to^*_\phi a^+ \circ a^* \circ a^-$, and therefore we have

$$M_1 \Rightarrow^*_{P_\phi *} \lambda x : \tau.[a^+ \circ a^* \circ a^-]N \Rightarrow_P [id_\tau \to (a^+ \circ a^*)](\lambda x : \tau.[a^-]N)$$

and

$$M_2 \Rightarrow^*_{P_\phi *} [id_\tau \to (a^+ \circ a^*)][id_\tau \to a^-](\lambda x : \tau.N) \Rightarrow_P [id_\tau \to (a^+ \circ a^*)](\lambda x : \tau.[a^-]N)$$

a common reduct of M_1 and M_2. The other case of overlap can be solved similarly. □

Let us also note some simple consequences of our earlier results: Polarized reduction P induces the original congruence E:

Lemma 2.2.3 *The congruence $=_P$ induced by $\Rightarrow_P$ is E.*

Proof We trivially have $=_P \subseteq E$. To see the converse inclusion, note that every coercion can be factored into a product of coercions with a polarity by C (Corollary 2.1.7), i.e., for every c we have $C \vdash c = c_1 \circ c_2 \circ \ldots \circ c_n$ for some c_i where each of the c_i has a polarity, $+$ or $-$. □

This shows that we also have :

Corollary 2.2.4 *The congruence $=_{P_\phi *}$ induced by $\Rightarrow_{P_\phi *}$ is ϕ-conversion.*

Proof Follows easily from the previous Lemma. □

From this observation it follows that confluence of $P_\phi *$ implies confluence of ϕ reduction.

Now, it was further conjectured in [Hen94] that the relation $P_\phi *$ is strongly normalizing. If this were true, we should have confluence of ϕ via Newman's Lemma (local confluence and termination implies confluence.) Unfortunately, the termination conjecture is *not* true, as the following counterexample shows.

Example 2.2.5 Recall from Figure 2.2 that $P*$ denotes the polarized orientation P of E taken *modulo* the equations for neutrals. $P*$ is not strongly normalizing: Let d^- be negative, c^* neutral. Let $M \equiv [id_\tau \to d^-](\lambda x : \tau.[c^*]N)$. Then

$$\begin{array}{lll} M & \Rightarrow_P & \lambda x : \tau.[d^- \circ c^*]N & \text{by } [P1^-] \\ & = & [id_\tau \to (d^- \circ c^*)]\lambda x : \tau.N & \text{since } d^- \circ c^* \text{ is neutral, by } [*1] \\ & = & [id_\tau \to d^-][id_\tau \to c^*]\lambda x : \tau.N & \\ & = & [id_\tau \to d^-](\lambda x : \tau.[c^*]N) & \text{by } [*1] \\ & \equiv & M & \end{array}$$

So the $P*$-system is *cyclic.* □

Note that cyclicity is due to the equations for neutrals. Later in this work, we shall prescribe a notion of reduction for neutral coercions, by introduction of explicit error-coercions. This will eliminate the need for special rules for neutral coercions, and it will be seen that the resulting system is not cyclic.

Nonetheless, the basic idea of adding the neutral equations to the polarized reduction P_ϕ is still very valuable. One also does not need to extend the system with error-reductions in order to establish confluence. As we now proceed to show, one can prove confluence in the absence of termination by appealing to commutativity properties instead of relying on Newman's Lemma.

2.2.3 Proof of confluence

The basic ideas in the rewrite analysis of the following confluence proof are as follows. In the absence of termination (recall Example 2.2.5 above) we must rely on commutativity properties (see [BD86]) The general technique is standard: We split a system R into parts, $R = \bigcup_i R_i$, and the parts are shown to commute (R_i commutes with R_j for $i \neq j$) Confluence of R then follows via the Hindley-Rosen Lemma, *provided* that the parts R_i are themselves confluent. To show this latter property, we show that each part R_i commutes with itself, which means just that each R_i has the *diamond property.* However, the natural parts R_i of the system $R = P_\phi *$ do not have the diamond property, since *several* ϕ reductions may be needed to resolve the critical pairs of the type *Case III* (2) of the proof of Proposition 2.2.2. This problem can be overcome by considering instead $P * \circ \phi^*$, the idea being that the critical pairs can be solved by *first* taking a *single* step of *several* ϕ reductions and then taking a *single* step of $P*$ reduction; in total, this amounts to taking a single step of $P * \circ\phi^*$. One then needs a few technical observations in order to transfer results about the relation $P * \circ\phi^*$ to the relation $P_\phi *$.

Lemma 2.2.6 *1. Let R, S, T be rewrite relations over the same term algebra. Then $T^* \subseteq S^*$ implies $((R \circ T^*) \cup S)^* = (R \cup S)^*$.*

2. Let $R_1, \ldots, R_n$ be any binary set theoretic relations over some set S, $R_i \subseteq S \times S$, and let T be any relation from S, $T \subseteq S \times U$, for some set U. Then one has

$$(\bigcup_i R_i) \circ T = \bigcup_i (R_i \circ T)$$

Proof *Ad 1.* One has $R \subseteq R \circ T^*$, which shows that the inclusion from right to left, $(R \cup S)^* \subseteq ((R \circ T^*) \cup S)^*$ holds.

Since $T^* \subseteq S^*$ by assumption, we have $R \circ T^* \subseteq (R \cup S)^*$, hence also $(R \circ T^*) \cup S \subseteq (R \cup S)^*$, hence also $((R \circ T^*) \cup S)^* \subseteq (R \cup S)^*$.

Ad 2. The claim follows from elementary reasoning with the definition of set theoretic composition of relations,

$$R_2 \circ R_1 = \{< x, z >\mid \exists y :< x, y >\in R_1 \wedge < y, z >\in R_2\}$$

□

Lemma 2.2.7 *Suppose R and S commute with T^*, and that T is confluent. Suppose also that one has*

$$M \rightarrow_R M_1, M \rightarrow_S M_2 \Rightarrow \exists P. M_1 \rightarrow_{S \circ T^*} P, M_2 \rightarrow_{R \circ T^*} P$$

Then $R \circ T^*$ and $S \circ T^*$ commute.

Proof By the assumptions one can erect the diagram:

$$\begin{tikzcd}
 & & & M \arrow[dll, dash, "T^*"'] \arrow[drr, dash, "T^*"] & & & \\
 & M_1 \arrow[dl, dash, "R"'] \arrow[drr, dash, dashed, "T^*"] & & & & M_2 \arrow[dll, dash, dashed, "T^*"'] \arrow[dr, dash, "S"] & \\
M_3 \arrow[dr, dash, dashed, "T^*"] & & & P_1 \arrow[dll, dash, dashed, "R"'] \arrow[drr, dash, dashed, "S"] & & & M_4 \arrow[dl, dash, dashed, "T^*"'] \\
 & P_2 \arrow[dr, dash, dashed, "T^*"] & & & & P_3 \arrow[dl, dash, dashed, "T^*"'] & \\
 & & P_4 \arrow[dr, dash, dashed, "S"] & & P_5 \arrow[dl, dash, dashed, "R"'] & & \\
 & & & P & & &
\end{tikzcd}$$

□

In the sequel, we let R_i denote each of the the P-rules of Figure 2.2 considered *modulo* $*$, one rule for each $i = 1, \ldots, 6$, so we have $P* = \bigcup_i R_i$.

Lemma 2.2.8 *For the reductions R_i one has:*

1. *Each reduction R_i has the diamond property*
2. *Each reduction R_i commutes with ϕ^**
3. *Each reduction $R_i \circ \phi^*$ has the diamond property*

Proof

1. The only critical pairs arising from superposition of R_i onto itself are caused by different factorizations (under C-equality) of coercions to the form $+ \circ c$ or different factorizations to the form $c \circ -$. The argument here is completely analogous to the one given in the proof of Proposition 2.2.2, *Case III* (2).
2. There are no critical pairs, as argued in the proof of Proposition 2.2.2, *Case I.*
3. We already know from [Hen94] that ϕ-reduction on coercions is confluent. From the previous two properties of this Lemma one can therefore erect the diagram:

$$\begin{tikzcd}
 & & M \arrow[dl, dash, "\phi^*"'] \arrow[dr, dash, "\phi^*"] & & \\
 & M_1 \arrow[dl, dash, "R_i"'] \arrow[dr, dash, dashed, "\phi^*"] & & M_2 \arrow[dl, dash, dashed, "\phi^*"'] \arrow[dr, dash, "R_i"] & \\
M_3 \arrow[dr, dash, dashed, "\phi^*"] & & P_1 \arrow[dl, dash, dashed, "R_i"'] \arrow[dr, dash, dashed, "R_i"] & & M_4 \arrow[dl, dash, dashed, "\phi^*"'] \\
 & P_2 \arrow[dr, dash, dashed, "R_i"'] & & P_3 \arrow[dl, dash, dashed, "R_i"] & \\
 & & P & &
\end{tikzcd}$$

□

Lemma 2.2.9 *For each i, j, it holds that $R_i \circ \phi^*$ commutes with $R_j \circ \phi^*$.*

Proof We show that the conditions of Lemma 2.2.7 are satisfied. First, we know from Lemma 2.2.8 that R_i and R_j both commute with ϕ^*.

Now consider an overlapping reduction

$$M \Rightarrow_{R_i} M_1, M \Rightarrow_{R_j} M_2$$

Using Lemma 2.2.2, one can show that all such overlaps can be resolved by *first* ϕ-reducing the coercions involved to the form $c^+ \circ c^* \circ c^-$, by analysis completely analogous to that given in the proof of Proposition 2.2.2, *Case III* (2). The only twist is that we must use the equations of $*$ to displace "stuck" coercions corresponding to errors. This analysis is also given in [Hen94]. Hence one has the diagram:

$$\begin{tikzcd}
 & & M \arrow[lld, dash, "R_i"'] \arrow[rrd, dash, "R_j"] & & \\
M_1 \arrow[rd, dash, dashed, "\phi^*"] & & & & M_2 \arrow[ld, dash, dashed, "\phi^*"'] \\
 & P_1 \arrow[rd, dash, dashed, "R_j"] & & P_2 \arrow[ld, dash, dashed, "R_i"'] & \\
 & & P & &
\end{tikzcd}$$

Now apply Lemma 2.2.7. □

Lemma 2.2.10 *The relation $\bigcup_i R_i \circ \phi^*$ is confluent.*

Proof By Lemma 2.2.8, each of the $R_i \circ \phi^*$ is confluent (by diamond property.) By Lemma 2.2.9, they all commute. Then repeated application of the Hindley-Rosen Lemma proves the claim. □

Theorem 2.2.11 *The relation $P_{\phi *}$ is confluent.*

Proof We know already that ϕ is confluent over coercions ([Hen94]) and also that ϕ^* commutes with each of the R_i (Lemma 2.2.8). Therefore we can erect the diagram:

$$\begin{tikzcd}
 & M \arrow[ld, dash, "\phi^*"'] \arrow[rd, dash, "\phi^*"] & & \\
M_1 \arrow[rd, dash, dashed, "\phi^*"] & & M_2 \arrow[ld, dash, dashed, "\phi^*"'] \arrow[rd, dash, "R_i"] & \\
 & P_1 \arrow[rd, dash, dashed, "R_i"] & & P_2 \arrow[ld, dash, dashed, "\phi^*"'] \\
 & & P &
\end{tikzcd}$$

This shows that ϕ^* commutes with $R_i \circ \phi^*$. Hence, by Lemma 2.2.10 and the Hindley-Rosen Lemma, we can conclude that the relation

$$(\bigcup_i R_i \circ \phi^*) \cup \phi$$

is confluent. But then

$$((\bigcup_i R_i \circ \phi^*) \cup \phi)^*$$

is also confluent. Now, by Lemma 2.2.6 (second part) we have

$$\bigcup_i R_i \circ \phi^* = (\bigcup_i R_i) \circ \phi^*$$

from which we get that

$$(((\bigcup_i R_i) \circ \phi^*) \cup \phi)^*$$

is confluent. But by Lemma 2.2.6 (first part) we have

$$(((\bigcup_i R_i) \circ \phi^*) \cup \phi)^* = ((\bigcup_i R_i) \cup \phi)^*$$

Since we clearly have

$$(\bigcup_i R_i) \cup \phi = P * \cup \phi = P_\phi *$$

it follows that $(P_\phi *)^*$ is confluent, and therefore $P_\phi *$ is confluent. □

Corollary 2.2.12 *In* **D**, *ϕ-reduction is confluent over λI.*

Proof Since the congruence induced by $P_\phi *$ is just ϕ conversion, the previous theorem entails that ϕ conversion has the Church-Rosser property, hence it is also confluent. □

2.2.4 Generalizations

We end the present chapter by observing that the proof *method* just presented is *robust* wrt. a number of interesting variations.

First, we note that this holds for certain *restrictions* of ϕ (or $P_\phi *$) reduction, we might be interested in. The main observation is that a high degree of commutativity entails a high degree of *modularity*; this is the content of the Hindley-Rosen Lemma: commuting relations are confluent in union, if the relations themselves are. By closely inspecting the properties used in the confluence proof above it can be seen that the following observation is true.

Observation 2.2.13 (Modularity of confluence)
Every restriction of the polarized reduction $P_\phi *$ of **D**, which emerges by removing any special case of the P-rules, will be confluent, and this fact can be stablished, in each case, by the same method of proof. One considers each rule left in the restriction and establishes comutativity with the other rules (including itself, yielding diamond property); in some cases it is necessary to compose with ϕ in order to obtain $+ \circ -$ factoring or $+ \circ * \circ -$ factoring. □

By the observation, we can define any restriction (along the lines mentioned) of $P_\phi *$ without worrying about confluence.

The proof methods employed in the present chapter (originating in [Hen94]) generalize to a quite significant range of problems. In general, the techniques could come into play whenever we consider *systems of inverse operations under data-flow equations E*. Such systems may arise in a variety of contexts. A closely related example is *representation analysis* (see [HJ94], [Ler92]) where the operations are inverse representation shift coercions, *boxing* and *unboxing*. Another, related, example, which we shall consider later, is *explicit type recursion* where the operations are *folding* and *unfolding* of type recursion. One can imagine yet other interesting variations over this theme. It may be noted here that the case of dynamic typing is especially complicated in this respect. This is due to the fact that "error-situations" can occur. In the other examples mentioned, such situations cannot arise, and therefore the complexity introduced by neutral coercions does not enter. Later, in Chapter 4, we shall eliminate the neutral coercions from dynamic typing, by prescribing a notion of reduction for neutrals, which introduces *explicit error-coercions*. We could, in principle, have done this for system **D**; however, we have chosen not to do so, because the results given here show that, from the point of view of confluence, one does not *need* any reduction on neutrals.

Polarized equality (P)

$$
\begin{array}{lcll}
[c]M & = & [c']M \quad \text{if } C \vdash c = c' & [E1] \\
[id_\tau]\,M & = & M & [E2] \\
[c \circ d]\,M & = & [c][d]M & [E3] \\
[d^+ \to c^-]\,(\lambda x : \tau.M) & \Rightarrow_P & \lambda x : \tau'.[c^-]M\{x := [d^+]x\} & [P1^-] \\
\lambda x : \tau.[d^+]M\{x := [c^-]x\} & \Rightarrow_P & [c^- \to d^+](\lambda x : \tau'.M) & [P1^+] \\
([c^- \to d^+]\,M)\,N & \Rightarrow_P & [d^+]\,(M\,([c^-]\,N)) & [P2^+] \\
[c^-](M\,([d^+]N)) & \Rightarrow_P & ([d^+ \to c^-]M)\,N & [P2^-] \\
(\mathbf{if}\ M\,([d^+]N)\,([d^+]N')) & \Rightarrow_P & [d^+](\mathbf{if}\ M\ N\ N') & [P3^+] \\
[c^-](\mathbf{if}\ M\ N\ N') & \Rightarrow_P & (\mathbf{if}\ M\,([c^-]N)\,([c^-]N')) & [P3^-]
\end{array}
$$

Side conditions :
c^- properly negative and d^+ properly positive.

Equations for neutral coercions ($*$)

$$
\begin{array}{lcll}
[c^* \to d^*]\,(\lambda x : \tau.M) & = & \lambda x : \tau'.[d^*]M\{x := [d^*]x\} & [*1] \\
([c^* \to d^*]\,M)\,N & = & [d^*]\,(M\,([c^*]\,N)) & [*2] \\
[c^*]\,(\mathbf{if}\ M\ N\ N') & = & (\mathbf{if}\ M\,([c^*]\,N)\,([c^*]\,N')) & [*3]
\end{array}
$$

Polarized ϕ reductions ($P_\phi, P_\phi *$)

P_ϕ-reduction ($\Rightarrow_{P_\phi}$) is ϕ-reduction on coercions together with the polarized equality $\Rightarrow_P$. $P*$-reduction and $P_\phi*$-reduction are, respectively, P *modulo* the equations for neutrals, and P_ϕ *modulo* the equations for neutrals.

Figure 2.2: Polarized reductions

Chapter 3

Quasi-static typing

The present chapter defines and briefly studies a subcalculus af **D** where the flow-equations in E are restricted in such a way that type assumptions (on bound variables) are held fixed. This subcalculus, called **Q**, can be seen as a form of generalized *quasi-static typing*, as introduced by Thatte [Tha90] (see also Section 1.2 of the Introduction and Section 8.2.) The generalization lies in the fact that Thatte's system only contained *local* reductions on coercions, and no "flow-movements" were allowed in the static reduction process. While quasi-static typing may be of some interest in its own right, our motivation for studying it here is mainly that **Q** defines a natural subsystem of **D**, which can be shown to be *formally completely well-behaved.* We note already at this point that the restricted equality of **Q** (see below for its definition) is sound for call-by-value evaluation, in distinction to the equational theory E in **D**. Moreover, as will be seen below (first section), ϕ-reduction is canonical in **Q**; hence we have existence and uniqueness of minimal completions in **Q**. The equational theory of **Q** is also simple in itself, since it turns out (second section) to be essentially finite (every equivalence class is finite, modulo the equations in C.) Finally (third section), the formal notion of safety satisfies a central property which indicates that it satisfactorily models what we want for **Q**. None of these properties are very difficult to establish, and, in a sense, they are all to be expected.

While this chapter can thus be read as a list of nice results about **Q**, it can, unfortunately, also be read as a list of broken dreams for system **D**, and for this reason, the present chapter contributes to our understanding of more powerful calculi like **D**. We believe that all of the properties mentioned for **Q** also hold in appropriate versions for **D** (with some restrictions, mainly the restriction to λI, which was seen to be necessary for confluence in **D**, in the previous chapter.) However, it will turn out that we can *prove* almost none of these results in their completely general form. The confluence property is a happy exception. However, the major open problem about **D** - the problem of termination of general ϕ-reduction - is left unsolved in the present report. This state of affairs is somewhat regrettable, because it indicates that the formal approach of dynamically typed λ-calculus may run into some serious problems of proof technology [1] However, we shall see later (Chapter 5) that it is possible to give quite natural proofs for restricted termination properties which are potentially of great practical interest. The study of **Q** throws light on these problems, by contrast, so to speak. The reader who wishes to appreciate the nature of the termination problem for **D** is advised to realize, in each case, why

[1] The author is humble enough to be aware of the possibility that the problems are caused by his own shortcomings. However, having spent almost one year of thinking about (among other things) the problem of termination for **D**, we are not prepared to readily give up the idea that the problem may *be* difficult. A termination proof or disproof, and nothing less, could convince us that this is not so.

the proof techniques employed in this chapter brake down in the case of **D**.

3.1 System Q

The calculus **Q** of quasi-static typing is obtained from **D** by restricting the completion equation $[E4]$ of **D** to:

$$[id_\tau \to d]\,(\lambda x:\tau.M) \quad = \quad \lambda x:\tau.[d]M \qquad [E_Q 4]$$

The coercion and type language of **Q** are those of **D**. We refer to the restricted equational theory of **Q** as E_Q.

The resulting calculus is intended to model dynamic typing for a language with *fixed type assumptions*, i.e. the language of *underlying terms* are the *preterms* of the *explicit* simply typed λ-calculus, $\lambda^{\to}$. The restriction of completion equality by $[E_Q 4]$ means that the equational theory of **Q** *respects the type assumptions of the underlying terms.*

Accordingly, completions are seen as resulting from a process of inserting coercions into a *preterm* of $\lambda^{\to}$ in such a way that the completion is well typed in system **Q** (although the underlying preterm may not be well typed in system $\lambda^{\to}$.) In other words, the *underlying term* or coercion erasure of a **Q** completion is a preterm of $\lambda^{\to}$, and it is obtained from the completion by erasing all coercions (but *not* erasing type assumptions on bound variables.)

3.1.1 Calculus of Coercions

The calculus of coercions of **Q** is the same as that of **D**.

3.1.2 Calculus of Completions

The completion calculus of **Q** is obtained from that of **D** by restricting the completion equality using axiom $[E_Q 4]$ instead of $[E4]$. The new notion of underlying terms gives rise to a new notion of canonical completions, which respects the type assumptions of the underlying term. The equational theory and the definition of canonical coercions is given in Figure 3.1.
Recall from Appendix A the notion of *canonical coercions*, ranged over by $\kappa_\tau^{\tau'}$. As in the case of **D**, we also define a *canonical completion* at an arbitrary type τ, $\langle\!|\, M \,|\!\rangle\, \Gamma\; \tau$ given by

$$\langle\!|\, M \,|\!\rangle\, \Gamma\; \tau \equiv [\kappa_{\mathbf{D}}^{\tau}]\langle\!|\, M \,|\!\rangle\, \Gamma$$

We denote by $\langle\!|\, M \,|\!\rangle$ the canonical completion $\langle\!|\, M \,|\!\rangle\, \Gamma_{\mathbf{D}}$.
We take over the remaining terminology of canonical completions from **D**, and (as is easily seen) the properties of canonical completions stated for that system also hold for **Q**. The reader is referred to Appendix A.

3.1.3 Conversion and Reduction

The notions of $\phi\psi$-conversion and -reduction remain the same, except for the fact that anything which is considered with the equational theory E in **D** *is considered with the restricted equational theory* E_Q in **Q**. In particular, this holds for the ϕ and ψ reductions $\Rightarrow_\phi$, $\Rightarrow_\psi$ on completions.

It is easy to establish that coherence holds for **Q**, using the techniques of the coherence proof in [Hen94].

Completion equality

$$
\begin{array}{lcll}
[c]M & = & [c']M \quad \text{if } C \vdash c = c' & [E_Q1] \\
[id_\tau]\,M & = & M & [E_Q2] \\
[c \circ d]\,M & = & [c][d]M & [E_Q3] \\
[id_\tau \to d]\,(\lambda x:\tau.M) & = & \lambda x:\tau.[d]M & [E_Q4] \\
([c \to d]\,M)\,N & = & [d]\,(M\,([c]\,N)) & [E_Q5] \\
[c]\,(\mathbf{if}\;M\;N\;N') & = & (\mathbf{if}\;M\;([c]\,N)\;([c]\,N')) & [E_Q6]
\end{array}
$$

Canonical completions

$$
\begin{array}{rcl}
\langle\!|\, x \,|\!\rangle\,\Gamma & \equiv & [\kappa^{\mathbf{D}}_{\Gamma(x)}]x \\
\langle\!|\, true \,|\!\rangle\,\Gamma & \equiv & [\mathtt{B}!]true \\
\langle\!|\, false \,|\!\rangle\,\Gamma & \equiv & [\mathtt{B}!]false \\
\langle\!|\, \lambda x.M \,|\!\rangle\,\Gamma & \equiv & [\kappa^{\mathbf{D}}_{\tau\to\mathbf{D}}](\lambda x:\tau.\langle\!|\, M \,|\!\rangle\,\Gamma[x:\tau]) \\
\langle\!|\, M\;N \,|\!\rangle\,\Gamma & \equiv & ([\mathtt{F}?]\langle\!|\, M \,|\!\rangle\,\Gamma)\;(\langle\!|\, N \,|\!\rangle\,\Gamma) \\
\langle\!|\, (\mathbf{if}\;M\;N\;P) \,|\!\rangle\,\Gamma & \equiv & (\mathbf{if}\;([\mathtt{B}?]\langle\!|\, M \,|\!\rangle\,\Gamma)\;(\langle\!|\, N \,|\!\rangle\,\Gamma)\;(\langle\!|\, P \,|\!\rangle\,\Gamma))
\end{array}
$$

Figure 3.1: Completion equality and canonical completions

Theorem 3.1.1 *For any* $\mathbf{Q}$ *completions* M *and* M', $M \cong M'$ *if and only if* $\mathbf{Q} \vdash M =_{\phi\psi} M'$. *Moreover, this property fails for any proper subset of the* $\mathbf{Q}$*-equations.*

Proof As in [Hen94]. □

3.2 Normalization and confluence

In this section, we prove that, in $\mathbf{Q}$, $\phi\psi$-reduction is canonical. The proof of termination is by a straight-forward well founded measure argument. Confluence follows by the rewrite analyses of the previous chapter; we can even drop the λI-restriction in $\mathbf{Q}$. This establishes (what we might perhaps have guessed a priori) that the proof theoretic analysis becomes much easier when type assumptions are held fixed.

For use in the normalization proof we introduce a number of technical definitions.

We introduce a notion of syntactic distance for occurrences of subterms within a term, expressing the nesting level within λ's and **if**'s at which the occurrences stand.

Definition 3.2.1 (Syntactic distances)
Given a completion M we ascribe to each subterm occurrence of M numbers δ_λ and $\delta_{\mathbf{if}}$ denoting, respectively, the number of occurrences of λ and **if** above the given subterm in the syntax tree of M. If a subterm M' occurs at a point associated with δ_λ ($\delta_{\mathbf{if}}$) within M then we say that M' is at λ-distance δ_λ (**if**-distance $\delta_{\mathbf{if}}$) from the root of M. We also write $\delta_\lambda(M') = n$ to express that the λ-distance of M' from the root of M is n (similarly we write $\delta_{\mathbf{if}}(M') = n$.) □

Definition 3.2.2 (Coercion contexts)
Let c range over arbitrary coercions.

$$\mathbf{C} ::= [] \mid (\mathbf{C} \circ c) \mid (c \circ \mathbf{C}) \mid (c \to \mathbf{C}) \mid (\mathbf{C} \to c)$$

□

We let $\mathbf{C}[c]$ denote the coercion that results from placing c in the hole ($[]$) in $\mathbf{C}$. It is understood that this operation is only well defined when $\mathbf{C}[c]$ is a well formed coercion.

For our proofs we shall be interested in various measures on coercions. Since our rewriting systems are *modulo C* we shall be interested in measures that are invariant w.r.t. core coercion equality.

Lemma 3.2.3 *(Invariance)*
Let μ be any map from coercions to numbers such that

$$\begin{array}{lll} (1) & \mu(\mathrm{id}) & = & 0 \\ (2) & \mu(c \circ d) & = & \mu(c) + \mu(d) \\ (3) & \mu(c \to d) & = & \mu(c) + \mu(d) \end{array}$$

for any coercions c, d. Then μ is invariant w.r.t. C-equality, i.e. $\mu(c) = \mu(d)$ whenever $C \vdash c = d$. Moreover, suppose μ is extended to coercion contexts by stipulating $\mu([]) = 0$. Then $\mu(\mathbf{C}[c]) = \mu(\mathbf{C}) + \mu(c)$ for any context $\mathbf{C}$ and coercion c.

Proof By (2) we see that μ is invariant w.r.t. reassociation. By (3) we see that μ is invariant w.r.t. distribution of composition and arrow. By (1) we have invariance w.r.t. elimination/introduction of id. Finally, (3) and (1) together imply invariance w.r.t. the equation $id_{\tau\to\tau'} = id_\tau \to id_{\tau'}$. The second claim is easily seen to hold (in virtue of (2) and (3)) by induction on $\mathbf{C}$. □

Define the measure $| \bullet |$ of a coercion (or coercion context) as the number of non-trivial, primitive coercion occurrences in the coercion (or context) :

Definition 3.2.4 (Coercion measure $| \bullet |$)

$$\begin{array}{lll} |id| & = & 0 \\ |p| & = & 1 \text{ if } p \not\equiv id \\ |(c \circ d)| & = & |c| + |d| \\ |(c \to d)| & = & |c| + |d| \end{array}$$

□

Note that, by Lemma 3.2.3, the measure defined above is invariant w.r.t. C, and also $|\mathbf{C}[c]| = |\mathbf{C}| + |c|$.

Definition 3.2.5 (Occurrences)
For a term M we let $\mathcal{O}(M)$ denote the set of all coercion occurrences in M. □

Theorem 3.2.6 *The reduction $\phi\psi$ on* **Q** *completions is strongly normalizing.*

Proof The proof is by defining a well founded measure which is invariant wrt. E_Q and which decreases under reduction. It is based on simple counting of non-trivial, primitive coercion occurrences, as given by the coercion measure $|\bullet|$ which we lift to completions by summation over all coercion occurrences within the subject term. The only problem arises from the non-linear rule $[E_Q6]$. This is handled by a weighting scheme, as follows.

For a fixed term M we let $m = \max\{\delta_{\mathbf{if}}(c) \mid c \in \mathcal{O}(M)\}$, that is the maximum of all **if**-distances of the subterms of M. Now define weights $w_0 = 2^m$ and $w_i = w_0/2^i$ for $0 \leq i \leq m$. So $w_{i+1} = w_i/2$, $0 \leq i \leq m-1$. The idea is that a coercion c occurring at **if**-distance $\delta_{\mathbf{if}}$ within M will be measured as $w_{\delta_{\mathbf{if}}}|c|$. Note that a coercion c ocurring at distance 0 will be measured by w_0.

Accordingly, for a term M and coercion $c \in \mathcal{O}(M)$ we let

$$\omega_M(c) = w(c)|c|$$

where $w(c)$ is w_i if c occurs at **if**-distance i from the root of M. We then define the measure $|\bullet|$ on completions by

$$|M| = \sum_{c \in \mathcal{O}(M)} \omega_M(c)$$

Since the measure $|\bullet|$ on coercions is invariant wrt. C it is easy to see that the measure $|\bullet|$ on completions is also invariant wrt. C.

To see that $|\bullet|$ is in fact invariant wrt. E_Q we reason as follows. We consider each of the axioms $[E_Q4]$ – $[E_Q6]$ ($[E_Q1]$ is already clear, and $[E_Q2]$ – $[E_Q3]$ are trivial.) As for rules $[E_Q4]$ and $[E_Q5]$, it is evident that if a coercion is displaced by any of these rules, the **if**-distance of the coercion will not change. Therefore the measure $|\bullet|$ will remain invariant under displacements effected by any of these two rules. This leaves us only with rule $[E_Q6]$, and the measure is taylored to neutralize this rule, as can be seen as follows.

Consider a subterm occurrence N within M of the form $N \equiv (\mathbf{if}\, N_1\, ([d]N_2)\, ([d]N_3))$ occurring at **if**-distance $\delta_{\mathbf{if}}(N)$. Then

$$|M| = \sum_{c \in \mathcal{O}(M)} \omega_M(c) = \sum_{c \in \mathcal{O}(M) \setminus \mathcal{O}(N)} \omega_M(c) + \sum_{c \in \mathcal{O}(N)} \omega_M(c)$$

Now, we have

$$\sum_{c \in \mathcal{O}(N)} \omega_M(c) = \left(\sum_{i=1}^{3} \sum_{c \in \mathcal{O}(N_i)} \omega_M(c) \right) + \omega_M(d_{N_2}) + \omega_M(d_{N_3})$$

where d_{N_2} is the occurrence of d applied to N_2 and d_{N_3} is the occurrence of d applied to N_3. Let d be either d_{N_2} or d_{N_3}. Then

$$\omega_M(d) = w_{\delta_{\mathbf{if}}(d)}|d| = w_{\delta_{\mathbf{if}}(N)+1}|d| = (w_{\delta_{\mathbf{if}}(N)}/2)|d|$$

and therefore

$$\omega_M(d_{N_2}) + \omega_M(d_{N_3}) = (w_{\delta_{\mathbf{if}}(N)}/2)|d| + (w_{\delta_{\mathbf{if}}(N)}/2)|d| = w_{\delta_{\mathbf{if}}(N)}|d| = \omega_{M'}(d)$$

with $M' \equiv M\{N := N'\}$ and $N' \equiv [d](\mathbf{if}\ N_1\ N_2\ N_3)$. From this we get

$$\sum_{c \in \mathcal{O}(N)} \omega_M(c) = \sum_{c \in \mathcal{O}(N')} \omega_{M'}(c)$$

and it follows that

$$|M| = |M\{N := N'\}|$$

This completes the proof that $|\bullet|$ is invariant wrt. E_Q. To see that $\phi\psi$ is strongly normalizing over **Q** completions we note that $\phi\psi$ evidently decreases the measure $|\bullet|$ on coercions, hence this reduction decreases that measure on completions, by invariance of $|\bullet|$ wrt. the equations. Since furthermore any term containing a $\phi\psi$-redex must contain at least one (in fact at least two) occurrences of a non-trivial primitive coercion it follows that any term containing a $\phi\psi$-redex has positive measure. Therefore there can be no infinite $\phi\psi$-reduction sequences from any term in **Q**. □

Corollary 3.2.7 *In* **Q***, ϕ-reduction is canonical.*

PROOF We need just to establish local confluence, since we have termination by the previous theorem. Local confluence is easily established by the analysis performed in Section 2.2, noting that the "pathological" counterexamples of Section 2.2.1 for λK cannot be produced, due to the restriction on the equational theory of **Q**. □

It is important to be aware that the method of proof used for termination in this section is not *transferrable to system* **D**. The reason is that the equational theory of **D** is no longer invariant wrt. the measures. It appears impossible to extend the method so as to regain invariance for **D** equality. The reader who wishes to understand the nature of the full termination problem for system **D** is advised to check this for himself. In general, we believe that a *very* good way to acquire a correct (as we take it) attitude towards that problem is to analyze, very carefully, how and why virtually *all* methods of proof of the present chapter brake down, once the restriction on the equational theory is dropped as we move to system **D**.

3.3 Equality

This section contains a quick study of the equational theory of **Q**. First we define a confluent orientation of the congruence E_Q. Moreover, both the oriented reduction and its inverse will be strongly normalizing. Next, we show how these properties entail that the congruence E_Q is finite *modulo* C[2] In other words, the equivalence E_Q is essentially *thin* [Der87]. This throws another light on the simplicity of **Q** and, by comparison with Section 2.2.1, the relative complexity of the calculus **D**.

Interestingly, Ghelli arrives at essentially the same orientation in his study of $F_{\leq}$ [Ghe90] (also an explicitly typed system with fixed declarations.) [3] We believe that the orientation shown

[2]Finiteness of a congruence A *modulo* a congruence A' means that the A'-quotient of every A- equivalence class is a finite set. This will be precisely defined in a short while.

[3]We arrived at the orientation independently, since we only discovered Ghelli's orientation after defining it. This is not to claim "discovery"; finding the orientation is in any case an easy exercise. The interesting point is that there appears to be only this one way of orienting the equality to a confluent reduction.

here is the only way of orienting the equality to a confluent reduction, and, characteristically, the method used method fails for **D**.

We introduce an oriented version of $[E_Q4] - [E_Q6]$. The objective is to obtain a confluent orientation of E_Q.

$$\lambda x : \tau.[c]M \quad \Rightarrow \quad [id_\tau \to c](\lambda x : \tau.M) \qquad [\vec{E}\ 1]$$

$$([c \to d]M)\ N \quad \Rightarrow \quad [d](M\ ([c]N)) \qquad [\vec{E}\ 2]$$

$$(\mathbf{if}\ M\ ([c]N)\ ([c]N')) \quad \Rightarrow \quad [c](\mathbf{if}\ M\ N\ N') \qquad [\vec{E}\ 3]$$

We now define $\vec{E}$-reduction $\Rightarrow_{\vec{E}}$ to be the union of $\Rightarrow$ above and the E_Q-axioms $([E_Q1] - [E_Q3])$, so

$$\Rightarrow_{\vec{E}} \quad = \quad \Rightarrow\ (mod\, C^+)$$

where C^+ is C lifted to completions, by the axioms $[E_Q1]$-$[E_Q3]$. We have the following easy Lemma:

Lemma 3.3.1 *1. The congruence induced by* $\Rightarrow_{\vec{E}}$ *is* E_Q

2. The oriented equality $\Rightarrow_{\vec{E}}$ *is locally confluent.*

3. The reduction $\Rightarrow_{\vec{E}}$ *is strongly normalizing.*

4. The oriented equality $\Rightarrow_{\vec{E}}$ *is confluent.*

5. $(\Rightarrow_{\vec{E}})^{-1}$ *is strongly normalizing.*

PROOF

1. Trivial.
2. The rules $[\vec{E}\ 1] - [\vec{E}\ 3]$ are non overlapping.
3. Rule $[\vec{E}\ 2]$ is strongly normalizing since it eliminates an arrow constructor on a non-trivial coercion, decreasing the number of coercion arrows present in the coercion compression (normalization under $([C2] - [C7])^{\to}$ $(mod\, [C1])$) of the term. From this it follows that in any infinite $\Rightarrow_{\vec{E}}$-reduction sequence the rules $[\vec{E}\ 1]$ and $[\vec{E}\ 3]$ must be used infinitely often. We now show that this is impossible.

   For a completion M define $\mathcal{C}_M = \{c \mid c \in \mathcal{O}(M)\}$, the set of all coercion occurrences in M, and consider $\mathcal{C}_M$ as a multiset of coercion terms. Now define the well founded orderings $<_\lambda$ and $<_{\mathbf{if}}$ on coercion occurrences w.r.t. M by $c <_\lambda c'$ iff $\delta_\lambda(c) < \delta_\lambda(c')$ and $c <_{\mathbf{if}} c'$ iff $\delta_{\mathbf{if}}(c) < \delta_{\mathbf{if}}(c')$. Then the rule $[\vec{E}\ 1]$ decreases the multiset ordering [4] on $\mathcal{C}_M$ induced by $<_\lambda$, and the rule $[\vec{E}\ 3]$ decreases the corresponding multiset ordering induced by $<_{\mathbf{if}}$. Further, all rules except $[\vec{E}\ 1]$ keep the multiset ordering induced by $<_\lambda$ invariant, and all rules except $[\vec{E}\ 3]$ keep the multiset ordering induced by $<_{\mathbf{if}}$ invariant. Therefore, neither $[\vec{E}\ 1]$ nor $[\vec{E}\ 3]$ reduction steps can occur infinitely often in any reduction sequence starting from M.

[4]See [DeMa79].

4. By properties established earlier in this proof together with Newman's Lemma.

5. The inverses of the rules $[\vec{E}\ 1]$ and $[\vec{E}\ 3]$ move non-trivial coercions downwards in the syntax tree of the coercion erasure of the subject term. The multiplication of coercions effected by rule $[\vec{E}\ 3]$ can be neutralized by considering the lexicographic measure introduced earlier in the present proof. The inverse of rule $[\vec{E}\ 2]$ either moves a non-trivial coercion downwards, or it moves a non-trivial coercion to the left in the syntax tree. Since the syntax tree is finite, this can only take place a finite number of times.

□

We shall now use the results of the previous lemma to show that the congruence E_Q is finite $(mod\, C^+)$. The following result gives a technique for showing finiteness (or quasi-termination [5]) of a congruence from König's Lemma (see, *e.g.*, [Cam94] for this very useful combinatorial property.) Although the general power of the technique is rather weak (assuming as it does very strong conditions on the congruence relation) it is applicable to the congruence E_Q.

For a set of congruence axioms A over a set of terms $\mathcal{T}$ let $Eq^A(t) = \{t' \in \mathcal{T} \mid A \vdash t = t'\}$, the A-equivalence class of t. Let A and A' be equational theories over $\mathcal{T}$. Then we say that the congruence A is *finite modulo* A' if and only if the quotient set

$$Eq^A(t)/A' = \{Eq^{A'}(t') \mid A \vdash t' = t\}$$

is a finite set of equivalence classes for every $t \in \mathcal{T}$.

Theorem 3.3.2 *(Finiteness of Congruence)*
Let R be a congruence over $\mathcal{T}$ with a division of the form $R = R_1 \cup (R_1)^{-1}$ satisfying

1. *R_1 is confluent and weakly normalizing*

2. *$(R_1)^{-1}$ is finitely branching and strongly normalizing*

Then $Eq^R(t)$ is finite for every $t \in \mathcal{T}$.

Proof By contradiction. Suppose $Eq^R(t)$ were infinite for some $t \in \mathcal{T}$. By 1. there exists a $\tilde{t} \in \mathcal{T}$ such that $t'\ R_1^*\ \tilde{t}$ for every $t' \in Eq^R(t)$, implying that the set $\{t'' \mid \tilde{t}\ (R_1^*)^{-1}\ t''\}$ is infinite. Since R_1^{-1} is finitely branching by 2., König's Lemma implies that the (R^{-1})-reduction tree starting from $\tilde{t}$ contains an infinite branch, contradicting the strong normalization assumption of 2. □

Theorem 3.3.3 *The congruence E_Q is finite modulo C^+.*

Proof $\Rightarrow_{\vec{E}}$ is confluent and strongly normalizing, by Lemma 3.3.1 above. It is easy to verify that $(\Rightarrow_{\vec{E}})^{-1}$ is finitely branching $(mod\, C^+)$, i.e. the the set of C^+-equivalence classes $\{Eq^{C^+}(M') \mid M(\Rightarrow_{\vec{E}})^{-1}M'\}$ is finite for every M. The claim now follows by Lemma 3.3.1 together with Lemma 3.3.2. □

[5]An abstract rewriting relation R is *quasi-terminating* if every infinite reduction sequence contains a cycle. If R is finite then R is quasi-terminating if and only if every reduction sequence contains only finitely many distinct terms. See [Der87] for further information.

3.4 Safety

In this section we study the notion of *formal safety* in **Q**. Recall the definition of *formal safety* of [Hen94]:

- A completion M' of the pure term M at τ wrt. Γ is called *formally safe* if and only if it holds that, for every other completion M'' at τ wrt. Γ, we have $M'' \Rightarrow^*_{\psi/\phi} M'$.

As noted in [Hen94] for **D**,

- *The canonical completion is safe at any type*

By the definition of safety and by safety of canonical completions, the study of formal safety reduces to the study of the properties of canonical completions wrt. the relation $\Rightarrow_{\psi/\phi}$; an arbitrary completion M' of M is safe, if and only if the canonical completion of M reduces under $\Rightarrow_{\psi/\phi}$ to M'.

In [Hen94] it was shown, for **D**, that in fact it is enough to consider ϕ conversion, since a major theorem of [Hen94] states that ψ reduction is *conservative* over ϕ conversion, in the sense that

- In **D**, one has $M \Rightarrow^*_{\psi/\phi} M'$ and $M' \Rightarrow^*_{\psi/\phi} M$, if and only if $M =_\phi M'$.

Hence, all the safe completions at a type are unique up to ϕ conversion, and the safe completions are exactly the ϕ-equivalence class of the canonical completion. Note, though, that even in the absence of a conservativity theorem, it would appear to be intuitively quite satisfactory to actually *define* formal safety as ϕ convertibility with the canonical completion. The reason is that the canonical completion is very obviously *operationally safe*, as was argued in the Introduction to this report. Hence, it is always "safe" to compute in the ϕ-equivalence class of the canonical completion. *Conservativity* would then state the property of the safe completions: A completion is safe, if and only if it and the canonical completion are interreducible under $\Rightarrow_{\psi/\phi}$. This entails that every completion (at the type in question) ψ/ϕ-reduces to any safe completion, since we know that every completion ψ/ϕ-reduces to the canonical completion. Thus, one would get back the definition of formal safety as defined in [Hen94], in case the conservativity property could be established.

In the remainder of this section we establish a fundamental property of formal safety in relation to canonical completions **Q**:

- For any *well typed pure term M of type* τ, one has

  $$\llparenthesis\, M \,\rrparenthesis\, \Gamma\, \tau \Rightarrow^*_\phi M$$

  in **Q**. In particular, any well typed pure term is formally safe.

The question whether this property holds or not is probably the single most important one, concerning formal safety, left over from the investigation of [Hen94]. Given the intuitive interpretation of the notion of formal safety, the property is not only to be expected; it is really *required*, since if it didn't hold, we should have to doubt whether the formal notion of safety were adequate: if any completion is safe at its type, then certainly a *pure* term which is already typable (using no coercions) is safe. Establishing that the property *does* hold is therefore a basic check that the *formal* notion models what we intend it to model.

It turns out that canonical completions of **Q** satisfy a very strong condition from which the desired property follows: If we are given any well typed pure term M at τ, then the canonical completion of M at **D** wrt. Γ will ϕ-reduce to $[\kappa_\tau^{\mathbf{D}}]M$. In other words, ϕ reduction will "clean up" the interior of $\langle\!| M |\!\rangle\, \Gamma$, pulling out all coercions into a canonical head-coercion applied to M itself. This is the content of the lemma below.

The lemma follows. In its proof we appeal to two properties of canonical coercions: First, one has

$$\kappa_{\mathbf{D}}^{\tau} \circ \kappa_\tau^{\mathbf{D}} \rightarrow_\phi^* id$$

For this property the reader is referred to [Hen94]. Second, one can infer the form of a canonical coercion from its signature. There is a syntactically unique canonical coercion at a given signature, and its form can be established by inspection of the definition of canonical coercions. For example, $\kappa_{\tau\to\sigma}^{\mathbf{D}}$ must be of the form $\mathtt{F!} \circ (\kappa_{\mathbf{D}}^{\tau} \to \kappa_\sigma^{\mathbf{D}})$.

Lemma 3.4.1 *If* $\Gamma \vdash_{\lambda^\to} M : \tau$ *then* $\langle\!| M |\!\rangle\, \Gamma \Rightarrow_\phi^* [\kappa_\tau^{\mathbf{D}}]M$

Proof By induction on the derivation of $\Gamma \vdash_{\lambda^\to} M : \tau$.

Case

$$\Gamma, x : \tau \vdash_{\lambda^\to} x : \tau$$

Here we have $\langle\!| x |\!\rangle\, \Gamma[x : \tau] \equiv [\kappa_\tau^{\mathbf{D}}]x$, which satisfies the claim.

Case

$$\Gamma \vdash_{\lambda^\to} \mathit{true} : \mathbf{B}$$

Here we have $\langle\!| \mathit{true} |\!\rangle\, \Gamma \equiv [\mathtt{B!}]\mathit{true} \equiv [\kappa_{\mathbf{B}}^{\mathbf{D}}]\mathit{true}$.

Case

$$\frac{\Gamma, x : \tau \vdash_{\lambda^\to} M : \sigma}{\Gamma \vdash_{\lambda^\to} \lambda x : \tau.M : \tau \to \sigma}$$

One has

$$\begin{array}{lll}
\langle\!| \lambda x : \tau.M |\!\rangle\, \Gamma & \equiv & \\
[\kappa_{\tau\to\sigma}^{\mathbf{D}}](\lambda x : \tau.\langle\!| M |\!\rangle\, \Gamma[x : \tau]) & \Rightarrow_\phi^* & \text{Induction} \\
[\kappa_{\tau\to\sigma}^{\mathbf{D}}](\lambda x : \tau.[\kappa_\sigma^{\mathbf{D}}]M) & = & \\
[\kappa_{\tau\to\sigma}^{\mathbf{D}}][id_\tau \to \kappa_\sigma^{\mathbf{D}}](\lambda x : \tau.M) & = & \text{Inspection of canonical coercions} \\
[\mathtt{F!}][\kappa_{\mathbf{D}}^{\tau} \to id_{\mathbf{D}}][id_\tau \to \kappa_\sigma^{\mathbf{D}}](\lambda x : \tau.M) & = & \text{Inspection of canonical coercions} \\
[\kappa_{\tau\to\sigma}^{\mathbf{D}}](\lambda x : \tau.M) & &
\end{array}$$

Case

$$\frac{\Gamma \vdash_{\lambda^\to} M : \tau \to \sigma \qquad \Gamma \vdash_{\lambda^\to} N : \tau}{\Gamma \vdash_{\lambda^\to} M\,N : \sigma}$$

One has

$$\begin{array}{lll}
\langle\!| M\,N |\!\rangle\, \Gamma & \equiv & \\
([\mathtt{F?}]\langle\!| M |\!\rangle\, \Gamma)(\langle\!| N |\!\rangle\, \Gamma) & \Rightarrow_\phi^* & \text{Induction} \\
([\mathtt{F?}][\kappa_{\tau\to\sigma}^{\mathbf{D}}]M)([\kappa_\tau^{\mathbf{D}}]N) & \equiv & \text{Inspection of canonical coercions} \\
([\mathtt{F?}][\mathtt{F!}][\kappa_{\mathbf{D}}^{\tau} \to \kappa_\sigma^{\mathbf{D}}]M)([\kappa_\tau^{\mathbf{D}}]N) & \Rightarrow_\phi & \\
([\kappa_{\mathbf{D}}^{\tau} \to \kappa_\sigma^{\mathbf{D}}]M)([\kappa_\tau^{\mathbf{D}}]N) & = & \\
[\kappa_\sigma^{\mathbf{D}}](M\,([\kappa_{\mathbf{D}}^{\tau}][\kappa_\tau^{\mathbf{D}}]N)) & \Rightarrow_\phi^* & \\
[\kappa_\sigma^{\mathbf{D}}](M\,N) & &
\end{array}$$

Case

$$\frac{\Gamma \vdash_{\lambda^\rightarrow} M : \mathbf{B} \qquad \Gamma \vdash_{\lambda^\rightarrow} N : \tau \qquad \Gamma \vdash_{\lambda^\rightarrow} P : \tau}{\Gamma \vdash_{\lambda^\rightarrow} (\mathbf{if}\; M\; N\; P) : \tau}$$

One has

$$\begin{array}{lll}
\langle\!|\, (\mathbf{if}\; M\; N\; P) \,|\!\rangle\, \Gamma & \equiv & \\
(\mathbf{if}\; ([\mathtt{B?}]\langle\!|\, M \,|\!\rangle\Gamma)\; (\langle\!|\, N \,|\!\rangle\Gamma)\; (\langle\!|\, P \,|\!\rangle\Gamma)) & \Rightarrow^*_\phi & \text{Induction} \\
(\mathbf{if}\; ([\mathtt{B?}][\kappa^{\mathbf{D}}_{\mathbf{B}}]M)\; ([\kappa^{\mathbf{D}}_\tau]N)\; ([\kappa^{\mathbf{D}}_\tau]P)) & \equiv & \text{Inspection of canonical coercions} \\
(\mathbf{if}\; ([\mathtt{B?}][\mathtt{B!}]M)\; ([\kappa^{\mathbf{D}}_\tau]N)\; ([\kappa^{\mathbf{D}}_\tau]P)) & \Rightarrow_\phi & \\
(\mathbf{if}\; M\; ([\kappa^{\mathbf{D}}_\tau]N)\; ([\kappa^{\mathbf{D}}_\tau]P)) & = & \\
[\kappa^{\mathbf{D}}_\tau](\mathbf{if}\; M\; N\; P) & &
\end{array}$$

□

Corollary 3.4.2 *(Safety of well typed terms)*
If $\Gamma \vdash_{\lambda^\rightarrow} M : \tau$ *then* $\langle\!|\, M \,|\!\rangle\, \Gamma\, \tau \Rightarrow^*_\phi M$

Proof By definition of $\langle\!|\, M \,|\!\rangle\, \Gamma\, \tau$ and the previous Lemma, we have

$$\langle\!|\, M \,|\!\rangle\, \Gamma\, \tau \equiv [\kappa^\tau_{\mathbf{D}}]\langle\!|\, M \,|\!\rangle\, \Gamma \Rightarrow^*_\phi [\kappa^\tau_{\mathbf{D}}][\kappa^{\mathbf{D}}_\tau]M \Rightarrow^*_\phi M$$

□

As the proof of the Lemma shows, the safety of a **Q** canonical completion is of a *local, compositional* nature, since ϕ reduction "cleans up" any such completion by "cleaning up" all its subcompletions. This is yet another reflection of the simplicity of completions under fixed type assumptions.

There is reason to inspect the proof of this property, because it illuminates a difference with the canonicals of **D**, where the lemma, characteristically again, *fails.* This reflects that severe difficulties will be encountered in an attempt to establish the same property for that stronger calculus. We have been unable to do so in the present work. The problem is that one ends up in a *global* consideration as to what can happen to the canonical head-coercion of an entire program under reduction. Straight-forward induction is out of the question here.

Safety of well typed terms can be read as a kind of *completeness property for ϕ reduction wrt. typability.* If a well typed term is ϕ convertible with the canonical completion, then (by confluence of ϕ reduction) the canonical completion ϕ reduces to any such term. Conversely, if the canonical completion reduces to a pure term, then that term must be well typed, because reduction preserves typing. Hence we see that

- A *pure* term is typable, if and only if it is safe.

While this may not be a very surprising property of typability in systems with fixed type declarations (because the typability problem here is trivial anyway), it would be a striking property in calculi without such restrictions. In any case, it appears to be an interesting characterization of safety, and it shows that, with fixed type assumptions, one can do *type checking by rewriting.*

3.5 Conclusion

We have shown that the natural subcalculus **Q** of quasi-static typing is proof-theoretically completely well-behaved. This indicates that a rich formal theory of quasi-static typing is possible.

The development presented above for **Q** is very much indicative of what we should ideally like the proof theoretic approach to do, for the general case of **D** as well as extensions thereof, which we might contemplate. However, a major lesson to be learned from the present chapter is, in our belief, that none of the techniques used here will be of use in attacking the remaining, general proof theoretic problems for **D**. The main problems here are

- The general termination problem for **D** (restricted to λI)
- The property of formal safety of well typed terms for **D**

These are natural properties which recieve equally natural proofs in **Q**. In the case of **D** (and stronger systems) we believe that we may have to face the possibility that the properties could be extremely difficult combinatorial problems. We suspect that this is the case for the termination problem, and we are very inclined to believe that classical techniques of *rewrite theory* will not solve the problem. For the problem of formal safety, it is evidently a disgrace if it turns out to be a very difficult property to establish that well typed terms are formally safe; this would be a disgrace because, intuitively, nothing could be safer than just such a term. If these considerations are to the point, then one conclusion we may have to draw is that is not likely that the approach of dynamically typed calculi will bring *all* the results they were intended to bring. This will be the more so if we find other means of coping with the problems. In the case of safety, we shall see later that one can reason with, *e.g.*, operational notions from semantics in a way which may be good enough for many practical purposes.

However, it is also important to stress that the approach of proof theory has brought some interesting results for the general system. Confluence is one, important instance of this. And certainly, the calculi have proven to be extremely useful in our "dayly" work on dynamic typing. This is hard to document in detail, but no less important for that. The other major example, in the present report, of success, as we take it, will be shown in Chapter 5 where it is proven that, in **D** as well as a powerful extension of **D**, unique normal forms always exist (over λI) in restricted completion classes which may be of great practical interest. We shall therefore *not* leave proof theoretic methods aside in the remainder of this work, and we shall continue to build our extension within the framework of a dynamic typing calculus in the spirit of **D**.

Chapter 4

Calculus of discriminative sums

The present chapter works towards the definition of a *generalized dynamic type calculus* which will eventually contain the elements

1. error-type
2. polymorphic types
3. (restricted) sum-types
4. recursive types

Thus, the system will contain the core characteristic elements of soft type systems. In addition, it will contain an explicit representation of run-time type error situations, in the form of *error-coercions*. As well as increasing the expressiveness of the system, these error-coercions provide a simplification of certain aspects of the calculus. In particular, they allow us to eliminate the neutral coercions, which were introduced for **D** in Chapter 2. This simplification was suggested by Henglein in [Hen94].

In working towards a natural generalization containing the elements mentioned above, we have found it very useful to consider an equational presentation of a category with sums. It turns out that, in such a theory, one can define a natural dynamic type calculus with a notion of ϕ-reduction and error-coercions, if we consider a category of sums with a terminal object **1** for which we require the logical absurdity (characteristic of dynamic typing)

$$\mathbf{0} = \mathbf{1}$$

to hold, where **0** is the initial object (empty sum.) The use of category theory is quite minimal and ought certainly to be "harmless" even to someone who thinks that category theory is being overdone in computer science these years. We use it simply to get an elegant, natural axiomatization of sums, for free as it were, and it turns that certain standard categorical constructions make perfect sense in our setting, which guide our extension in a natural way.

While the categorical viewpoint could presumably be taken further than we do here, it should be stressed from the beginning that our ultimate aim in this chapter is a *practical* one. We want to arrive at a framework in which we can extend the existing techniques of dynamic typing to yield, ultimately, more powerful systems of completion inference. Given such aims, we strongly believe that it is sometimes more important to strive for simplicity than simply to go for expressiveness. An example may be in place here. One of the most striking advances in programming language research appears (to us) to be the definition and study of Milner's polymorphism, culminating (in practical terms) in *ML* [MTH90] [MTH91] and its relatives. Much research since Milner's paper [Mil78] seems to have centered around attempts to extend this system, in various directions. Work on polymorphic subtyping is an example of this. However, it appears to be *very* difficult to extend the system radically without loosing properties which are important in practice. Instances of this is that the usual notion of principality fails when moving to polymorphic subtyping (see, *e.g.*, [FM90]) and a more radical example could be that moving to higher (than 2) rank fragments of second order polymorphic λ-calculus leads to undecidable type inference problems (see [KW94].) The strategy taken in the present report is to try to perform an "approximation from below" of the ideal, practical goal. We shall try to indicate, in the remainder of this report, that the framework suggested here may be further refined to improve the approximation. In a sense, this can be contrasted with the, very impressive, system of soft typing constructed by Aiken, Wimmers and Lakshman [AWL94] which is rather an attempt at "approximation from above" since this system relies on algorithmic techniques which may be very costly. [1]

The remainder of the chapter is organised as follows. The first section gives an equational presentation of categorical co-products, on the basis of which the second section introduces the notion of error-coercions. The third section demonstrates how constructions and derived rules can be found in the categorical framework which lead to a notion of "categorical ϕ-reduction". The fourth section goes more practical, imposing pragmatically motivated restrictions on the general categorical framework. It results in a basic dynamic type calculus, called Σ, with *polymorphic, discriminative sums.* Confluence of the completion reduction is seen to be transferrable by the techniques deveoped in Chapter 2. The fifth section adds *regular recursive types*; this extension is given in two varieties, leading to extended systems called Σ^{μ} (with explicit type recursion coercions) and Σ^{∞} (with implicit recursion, equivalent to an infinitary system over regular trees.) The last, sixth section of the chapter shows that (not surprisingly) **D** can be embedded in the recursive systems, because a universal type can be defined in these systems. We round off with a quick comparison with system **D**, indicating that the extension presented here can lead to arbitrarily large improvements in completion.

4.1 Categories with co-products

We recall the notion of a *categorical sum*, or *co-product* (see *e.g.*, [BW90], [LaSc86].) Given two objects A and B of a category, their co-product is given by two kinds of data: first, an object, called $A+B$, and second, morphisms $\iota_A^{A+B} : A \to A+B$, $\iota_B^{A+B} : B \to A+B$, called *(canonical) injections*, such that for any object C and morphisms $f : A \to C$, $g : B \to C$ there exists a unique morphism $[f,g] : A+B \to C$, called the *mediating morphism*, for which the following diagram commutes:

$$\begin{tikzcd} & A+B \arrow[d, "{[f,g]}"] & \\ A \arrow[ur, "\iota_A"] \arrow[r, "f"] & C & B \arrow[ul, "\iota_B"'] \arrow[l, "g"'] \end{tikzcd}$$

The co-product is a product of the dual category, hence the name.

[1]This is not to pass any judgement on the ultimate relative merits of various soft type systems; this would be quite premature. Chapter 6 contains a comparison between our system and systems of soft typing, including the system of [AWL94].

It is rather straight-forward to give an equational presentation of categorical sums; we simply axiomatize the above conditions together with the basic categorical properties of composition and identities [2]

$$\begin{array}{lll}
[C1] & f \circ (g \circ h) & = (f \circ g) \circ h \\
[C2] & f \circ id & = f \\
[C3] & id \circ f & = f \\
 & & \\
[S1] & [f_A^C, g_B^C] \circ \iota_A^{A+B} & = f_A^C \\
 & & \\
[S2] & [f_A^C, g_B^C] \circ \iota_B^{A+B} & = g_B^C \\
 & & \\
[S3] & [f_{A+B}^C \circ \iota_A^{A+B}, f_{A+B}^C \circ \iota_B^{A+B}] & = f_{A+B}^C
\end{array}$$

The first three axioms are the basic ones; the next three axiomatize the particular properties of sums: $[S1]$ and $[S2]$ express the commutativity property of the diagram, and $[S3]$ expresses the uniqueness property of the mediating morphism. As already indicated, every one of the S-axioms has a dual mate in the corresponding axiomatization of products in the dual category, and it is worth noting that axiom $[S3]$ corresponds with the axiom of *surjective pairing*

$$< \pi_{A\times B}^A \circ f_C^{A\times B}, \pi_{A\times B}^B \circ f_C^{A\times B} >= f_C^{A\times B}$$

The construction of sums is generalized to an n-ary operation (or, for that matter, an infinite one, but we shall not consider that here) producing n-ary sums denoted

$$\sum_{i\in I} A_i$$

with the mediating morphism $[f_1, \ldots, f_n]$, in short notation [3]

$$\bigvee_{i\in I} f_i$$

for a collection of objects A_i and morphisms f_i indexed by I. We shall usually leave I implicit, assuming $I = \{1, \ldots, n\}$ for some n. In the extreme case where $I = \emptyset$ we get the 0-ary sum. This sum can be seen to be an *initial object*, called $\mathbf{0}$, uniquely determined up to isomorphism by the properties: for every object A of the category there is a *unique* morphism $\uparrow^A : \mathbf{0} \to A$. Below is an equational characterization of the initial object and $\uparrow^A$:

$$\begin{array}{lll}
[I1] & \uparrow^{\mathbf{0}} & = id_{\mathbf{0}} \\
[I2] & f_A^B \circ \uparrow^A & = \uparrow^B
\end{array}$$

[2]See [LaSc86] for a corresponding equational presentation of products; given the duality between products and sums, our axiomatization can actually be read off from theirs.

We give explicit typing in the fundamental co-product equations below. This is done here to make it quite clear that categorical morphisms are indeed typed (by their sources and targets); however, in the sequel we shall often adopt the common policy of leaving typing implicit. In such cases it is always understood that morphisms which are mentioned in any context are subject to the restriction that they must be well typed.

[3]The present short notation for the mediating morphism of an n-ary sum is not standard (at least not as far as we're aware.) It is convenient for our purposes, though.

Note that, considering how the *functions* $[f, g]$ act in the category of sets, it is not difficult to get the idea that the morphisms

$$\bigvee_i f_i$$

can intuitively be realized in a typed programming language, by a general `case`-construct:

```
[f1,...,fk] = fn x : A1+...+Ak => case x of
                                     inA1 y => f1 y
                                   | inA2 y => f2 y
                                           ...
                                   | inAk y => fk y
```

Under this interpretation, the categorical equations express simple program equalities (or transformations) which are quite intuitive.

Following Lambek and Scott [LaSc86], we can regard our equations so far as a so-called *deductive system*, a categorical presentation of a logical proof system, for disjunction. Assuming the existence of the initial object corresponds to adopting the intuitionistic absurdity axiom [4] We can also regard the system as a form of subtyping logic for explicit union types. Under this view, we can try to read a morphism $f : A \to B$ as a proof witness that the inclusion $A \subseteq B$ holds. Under this reading, the equations give the follwing axiomatization (in the style of natural deduction)

$$\frac{}{A \subseteq A + B}$$

$$\frac{A \subseteq B \quad A' \subseteq B}{A + A' \subseteq B}$$

From these rules we can derive other laws, like this one (verification left for the reader):

$$\frac{A \subseteq A' \quad B \subseteq B'}{A + A' \subseteq B + B'}$$

and the proof witness corresponding to this derivation is, in general, the morphism

$$\bigvee_i (\iota_i \circ f_i) : \sum_i A_i \to \sum_i B_i$$

which can be constructed from the "assumptions" $f_i : A_i \to B_i$.

It turns out that it is also possible to read our axiomatization as a natural basis for a generalized dynamic type coercion calculus which includes explicit error coercions. To this view we now turn.

4.2 Categorical view of dynamic type coercions with errors

Let us expand an observation made in [Hen94], by noting that the equational theory C of system **D** contains (in the axioms $[C1]$-$[C3]$) the basic categorical axioms (also called $[C1]$-$[C3]$ above.) So, dynamic type coercions actually constitute a category with types as objects and coercions as morphisms. But the categorical nature of coercions goes further than that, since we can

[4]We are here touching upon the presently flourishing *objects-as-propositions-as-types* correspondence between typed programming languages, systems of logic, and categories. See [LaSc86], [RS94] for further information and references. See also the next footnote here.

recognize in the equations $[C4] - [C5]$ of $\mathbf{D}$ a characterization of the coercion constructor $\rightarrow$ as a contravariant bi-functor; this is also noted in [Hen94]. The idea naturally suggests itself that, in general, coercion constructors are to be regarded as having a functorial behaviour. For instance, suppose we have product types $A \times B$. Then, following the general outlines of system $\mathbf{D}$, we should want to introduce primitive coercions $\texttt{pair!} : \mathbf{D} \times \mathbf{D} \rightsquigarrow \mathbf{D}$, $\texttt{pair?} : \mathbf{D} \rightsquigarrow \mathbf{D} \times \mathbf{D}$ and induced coercions $c_A^B \times d_C^D : (A \times C) \rightsquigarrow (B \times D)$. Regarding $\_ \times \_$ as a functor immediately generates the following coercion equations, which turn out to make perfect sense:

$$\begin{array}{lcl} id_A \times id_B & = & id_{A \times B} \\ (c \times c') \circ (d \times d') & = & (c \circ c') \times (d \circ d') \end{array}$$

Furthermore, it is a natural thought that the universal dynamic type $\mathbf{D}$ can somehow be articulated as a *sum-type*. Indeed, this is evident from the denotational viewpoint, where the recursive equation for the universal domain interpreting $\mathbf{D}$ shows that passing from one side to the other in one direction of the isomorphism exactly exposes a sum-structure on the domain. The mappings constituting the isomorphism are indeed also injections and projections.

Generalizing the coercion calculus from a categorical basis is now straightforward: For every n-ary type constructor F we expect positive and negative primitives $F^!$, $F^?$, respectively, and, in case $n > 0$, a functorial constructor F, such that we have the morphisms

$$F^! : F(A_1, \ldots, A_n) \rightsquigarrow F(A_1, \ldots, A_n) + B$$

$$F^? : F(A_1, \ldots, A_n) + B \rightsquigarrow F(A_1, \ldots, A_n)$$

$$F(c_1, \ldots, c_n) : F(A_1, \ldots, A_n) \rightsquigarrow F(B_1, \ldots, B_n)$$

with $c_i : A_i \rightsquigarrow B_i$, and the functor equations

$$\begin{array}{lcl} F(id_{A_1}, \ldots, id_{A_n}) & = & id_{F(A_1, \ldots, A_n)} \\ F(c_1, \ldots, c_n) \circ F(d_1, \ldots, d_n) & = & F(c_1 \circ d_1, \ldots, c_n \circ d_n) \end{array}$$

Now, suppose we want to model *explicit run-time type errors*. Like the sums, this is a new feature we might like to express in a coercion calculus. Consider the situation where we have a coercion of the form

$$G^? \circ F^!$$

with functors $F \neq G$. Such a coercion can indeed be formed, and it signifies the presence of a *functor-clash* which we naturally interpret as a run-time type error. Such coercions are irreducible (neutral) coercions in system $\mathbf{D}$ (take, *e.g.*, $\texttt{B?} \circ \texttt{F!}$) where only negative/positive and positive/negative compositions are reducible (by ψ and ϕ reduction, respectively.) It is a natural thought that we might try to signify more explicitly that an error-situation has arised, by prescribing some kind of reduction on neutrals. This, in turn, could be done by introducing *explicit error-coercions*, once again following a suggestion contained in [Hen94]. What should they look like? Consulting our semantics for the behaviour of run-time errors, we see that the error element (called ε, to recall) has the following properties:

- It can arise nested inside arbitrary coercion contexts
- It *propagates*, at least at the level of coercions

Combining these properties, it is natural to require that an error coercion should be able to propagate into an arbitrary context. From the point of view of typing, this suggests that a *single* error coercion should exist at *every* possible signature.

There is a very obvious categorical realization of this, namely by adding a *terminal object*, called **1**, with a unique morphism $\downarrow_A : A \to \mathbf{1}$ at every type A, equationally:

$$\begin{array}{lll} [T1] & \downarrow_{\mathbf{1}} & = & id_{\mathbf{1}} \\ [T2] & \downarrow_B \circ f_A^B & = & \downarrow_A \end{array}$$

The object **1** could be regarded as an *error-type*, and at every type A we have the unique coercion $\downarrow_A$ to signify that a run-time type error is present. Moreover, to satisfy the requirements of our type system in the presence of a propagating element we would expect to include an "intuitionistic" axiom. Indeed, we already did this in our operational semantics, when we introduced the error element ε; anyway, the intuitionistic typing of abortive operators has become quite standard [5] Furthermore, as we have already noted, the initial object **0** gives us just the intuitionistic construction. The final step needed, then, to arrive at a notion of error-coercions is to make the logically absurd identification

$$\mathbf{0} = \mathbf{1}$$

The resulting initial-and-terminal object can aptly be called **E** (for **Error**.) At every pair of types A, B, we now have a unique morphism $\uparrow^B \circ \downarrow_A : A \to B$ factoring through **E**, and for this we introduce the convenient shorthand notation

$$\updownarrow_A^B \equiv \uparrow^B \circ \downarrow_A$$

The resulting system is indeed proof-theoretically inconsistent, since *every* formula is now provable. This is well and only to be expected, since a system suitable for dynamic typing must give every program a (completion-) type.

We can regard the axioms introduced so far as a formal *equational theory.* We refer to it as System **Co**, and the axioms of **Co** are summarized in Figure 4.1, where the equations [S1]-[S3] appear in generalized (and simplified) form, as [**Co**1]-[**Co**2].

[5] Witness the rule for typing exceptions in *ML* [MTH90]. See also, *e.g.*, [DHM91],[RS94], [deGr95] where further references can be found. It is perhaps well worth stressing that the logical (intuitionistic) view of abortive operations of the kind we are concerned with here is arguably *not* something we have to *force* upon them; rather, it appears to force itself upon us. In the case of more powerful control operations there appears to be an interesting correspondence with classical logic, although here the propositions-as-types correspondence is perhaps less smooth; in any case, it is hardly completely understood. In [RS94] the reader may find this issue discussed together with further references.

$$
\begin{array}{lcll}
(\bigvee_i f_i) \circ \iota_j & = & f_j & [\mathbf{Co}1] \\
\bigvee_i (f \circ \iota_i) & = & f & [\mathbf{Co}2] \\
f_A^B \circ \uparrow^A & = & \uparrow^B & [\mathbf{Co}3] \\
\downarrow_B \circ f_A^B & = & \downarrow_A & [\mathbf{Co}4] \\
\uparrow^{\mathbf{E}} & = & id_{\mathbf{E}} & [\mathbf{Co}5] \\
\downarrow_{\mathbf{E}} & = & id_{\mathbf{E}} & [\mathbf{Co}6]
\end{array}
$$

Figure 4.1: System **Co**

4.3 Derived rules and categorical ϕ-reduction

On the basis of the **Co**-axioms we can make a number of interesting constructions and derivations. For later convenience we shall now start working exclusively with sums $\sum$ of arbitrary arity.

First, we have the derived equations for $\bigvee$:

$$\begin{array}{lll}
[D1] & \bigvee_i \iota_i & = & id \\
[D2] & f \circ \bigvee_i g_i & = & \bigvee_i (f \circ g_i)
\end{array}$$

To verify, we have

$$\bigvee_i \iota_i = \bigvee_i (id \circ \iota_i) = id$$

and

$$\begin{array}{rcl}
f \circ \bigvee_i g_i & = & \bigvee_i (f \circ (\bigvee_j g_j)) \circ \iota_i \\
& = & \bigvee_i f \circ ((\bigvee_j g_j) \circ \iota_i) \\
& = & \bigvee_i (f \circ g_i)
\end{array}$$

Next, let us recall the standard categorical construction of sums of morphisms. If we have morphisms $f_i : A_i \to B_i$ then we can form the morphism

$$\sum_i f_i : \sum_i A_i \to \sum_i B_i$$

given by

$$\sum_i f_i \equiv \bigvee_i (\iota_i \circ f_i)$$

Using this definition and properties already shown it is easy (and left to the reader) to verify the laws

$$\begin{array}{llll}
[D3] & (\bigvee_i f_i) \circ (\sum_i g_i) & = & \bigvee_i (f_i \circ g_i) \\
[D4] & (\sum_i f_i) \circ \iota_j & = & \iota_j \circ f_j \\
[D5] & (\sum_i f_i) \circ (\sum_i g_i) & = & \sum_i (f_i \circ g_i) \\
[D6] & \sum_i id & = & id
\end{array}$$

The reader may find it interesting to interpret the equations introduced so far as program equalities, using the translation given earlier, and see that they are quite intuitive.

Now, suppose we define (for $j \in I$) the morphism

$$\pi_j : \sum_i A_i \to A_j$$

such that

$$\pi_j = \bigvee_i \varepsilon_i^j$$

with

$$\varepsilon_i^j = \begin{cases} id_{A_i} & \text{for } i = j \\ \updownarrow_{A_i}^{A_j} & \text{for } i \neq j \end{cases}$$

Then we have the equation

$$[D7] \quad \pi_j \circ \iota_i \quad = \quad \begin{cases} id & \text{for } i = j \\ \updownarrow_{A_i}^{A_j} & \text{for } i \neq j \end{cases}$$

because

$$\pi_j \circ \iota_i = (\bigvee_k \varepsilon_k^j) \circ \iota_i = \varepsilon_i^j$$

We recognize in [$D7$] a generalization of ϕ-conversion to arbitrary sums and error-coercions. For suppose we transport our error-coercions back to system **D**. Then we would expect just equations like

$$\begin{aligned} \texttt{F?} \circ \texttt{F!} &= id \\ \texttt{B?} \circ \texttt{F!} &= \updownarrow^{\mathbf{B}}_{\mathbf{D}\to\mathbf{D}} \end{aligned}$$

In short, we have found that introducing the equation $\mathbf{0} = \mathbf{1}$ in a category with sums and terminal object we can *define* negative coercions π_j, and in such a way that we get a generalized ϕ conversion. Our categorical framework gives the following intuitive "implementation" of the projections π_j :

```
pi_j = fn x => case x of
                 in_Aj y => y
               | _       => Error
```

With these definitions we can also derive another interesting law, namely

$$[D8] \quad \pi_j \circ \textstyle\sum_i f_i \quad = \quad f_j \circ \pi_j$$

which is justified by previous equations and definitions, thus:

$$\begin{aligned}
\pi_j \circ (\textstyle\sum_i f_i) &= \pi_j \circ \bigvee_i(\iota_i \circ f_i) \\
&= \bigvee_i(\pi_j \circ \iota_i \circ f_i) \\
&= \bigvee_i(\varepsilon_i^j \circ f_i) \\
&= \bigvee_i(\left\{ \begin{array}{ll} f_j & \text{for } i = j \\ \updownarrow \circ f_i & \text{for } i \neq j \end{array} \right\}_i) \\
&= \bigvee_i(\left\{ \begin{array}{ll} f_j \circ id & \text{for } i = j \\ f_j \circ \updownarrow & \text{for } i \neq j \end{array} \right\}_i) \\
&= \bigvee_i(f_j \circ \left\{ \begin{array}{ll} id & \text{for } i = j \\ \updownarrow & \text{for } i \neq j \end{array} \right\}_i) \\
&= f_j \circ \bigvee_i(\left\{ \begin{array}{ll} id & \text{for } i = j \\ \updownarrow & \text{for } i \neq j \end{array} \right\}_i) \\
&= f_j \circ \bigvee_i \varepsilon_i^j \\
&= f_j \circ \pi_j
\end{aligned}$$

The equation [$D8$] is in a very pleasing correspondence with the equation [$D4$]. Note that we used the identification $\mathbf{0} = \mathbf{1}$ in an essential way during the derivation; indeed, in general categories with sums we cannot expect any "left analogue" of [$D4$] to hold.

Now that we have found that we can derive the ϕ equations, we also want to know whether we can do the same thing for the ψ equations. A ψ-redex should be of the form $\iota \circ \pi$, so let us try just $\iota_i \circ \pi_i$,

$$\begin{array}{rcl} \iota_i \circ \pi_i & = & \iota_i \circ \bigvee_j \varepsilon^i_j \\ & = & \bigvee_j (\iota_i \circ \varepsilon^i_j) \end{array}$$

which does *not* verify ψ equality. Indeed, if we take the most blatantly *unsafe* type of coercion,

$$\updownarrow^A_A$$

we can see that this coercion is not provably equal to id_A. On the other hand, adding the new axiom

$$[IT] \quad \updownarrow^A_A \;=\; id_A$$

we can verify the ψ-equations. This is even trivially the case, because adding $[IT]$ to *just* the other error-axioms, $[I]$ and $[T]$, we have, for *arbitrary* $f : A \to B$,

$$f^B_A = f^B_A \circ id_A = f^B_A \circ \updownarrow^A_A = \updownarrow^B_A$$

In other words, the system becomes trivially *coherent* by just the axioms mentioned. In this situation it may well be asked why we should be interested in any equations apart from those. The answer is is that, as earlier, we shall reject the ψ-equations, our main goal being to obtain a good notion of ϕ-reduction. And in this enterprise, many more equations will be needed, as will be seen in the following section.

In the next section we shall begin the definition of a generalized calculus of dynamic typing, which will eventually include polymorphic sum types and recursive types. In this enterprise, we shall so to speak *project* a simpler (less expressive) theory out of the categorical calculus outlined here, by restricting the constructions allowed. This is done in order to obtain what we believe to be the natural, *basic* system suited for *practical use.* We do note, however, that it is in principle possible to define a general notion of "categorical" ϕ-reduction, which arises by Knuth-bendix completion [KB67] of the equational theory. This results in the confluent relation:

$$\begin{array}{lcl} (\bigvee_i f_i) \circ \iota_j & \to & f_j \\ \bigvee_i (f \circ \iota_i) & \to & f \\ f \circ \uparrow & \to & \uparrow \\ \downarrow \circ f & \to & \downarrow \\ f \circ \bigvee_i g_i & \to & \bigvee_i (f \circ g_i) \\ \bigvee_i \downarrow_{A_i} & \to & \downarrow_{\sum_i A_i} \end{array}$$

The relation should be taken *modulo* the equation $\uparrow^{\mathbf{E}} = \downarrow_{\mathbf{E}} = id_{\mathbf{E}}$. See Remark 4.4.8 below for a further comment on this reduction.

We note also that, by dualization, we could construct a similar system $\prod$ which, rather than being based on sum-types, is based on products. The construction of injections and projections will proceed dually to the way it was done for sums here. In particular, the *projections* π_i are now "built-in" and the *injections* ι_i can be *defined* from $\updownarrow$ together with the characteristic construction, pairing $\langle \bullet, \bullet \rangle$, by

$$\iota_j \equiv \bigwedge_i \varepsilon^j_i$$

As an example (assuming the type language of $\mathbf{D}$) we could represent $[\texttt{B!}]M$ as $\langle M, [\updownarrow]M \rangle$, $[\texttt{F!}]M$ as $\langle [\updownarrow]M, M \rangle$, $\texttt{B?}$ as π_1, $\texttt{F?}$ as π_2, by which we have $[\texttt{B?} \circ \texttt{B!}]M = M$ and $[\texttt{B?} \circ \texttt{F!}]M = [\updownarrow]M$.

4.4 System Σ

In this section we shall use the categorical framework just developed to set up a calculus of dynamic type coercions with sums. While it might well be interesting to investigate the full, categorical system previously outlined in this chapter, we shall henceforth make some pragmatic concessions in the present work. For instance, co-products are associative, up to isomorphism: $(A+B)+C \cong A+(B+C)$. A consequence of the categorical abstraction is that any association of a repeated binary sum of objects A_i can also be regarded as an n-ary sum $\sum_i A_i$ (see [BW90], 5.3.3) We should not wish to *implement* this abstraction directly. Rather, we shall regard the sum as, effectively, an n-ary constructor, relative to a fixed index set $I = \{1, \ldots, n\}$. Furthermore, sums will be required to be *discriminative* in the sense that no two summands may have the same top-level constructor, and summands appear in a fixed order. Some pragmatically desirable features of this requirement are seen below.

Moreover, as will also be seen below, our system arises from the categorical theory of the previous sections by restriction: We take now projections π (negative coercions) as *primitive* (rather than definable) notions, leaving out the general, categorical construction $\bigvee$ (in terms of which the projections were definable.) This results in a simplification, suitable for a first study as well as for practical purposes.

4.4.1 Types and coercions

Type expressions

We assume an arbitrary, but fixed, ranked alphabet of *type-constructors*,

$$\mathcal{F} = \{F_1, \ldots, F_n\}$$

indexed by $I = \{1, \ldots, n\}$, ordered numerically. The rank (or arity) of F_i is called α_i, and for $\alpha_i = 0$, F_i is just a type constant. We always assume that $\mathcal{F}$ contains at least one constant; in particular, unless anything else is said, we assume a type constant $\mathbf{E}$ (error type.)

The type expressions, ranged over by $\tau, \theta, \eta, \zeta$ are given by

$$\tau \quad ::= \quad F_i(\tau_1, \ldots, \tau_{\alpha_i}) \mid \textstyle\sum_{i=1}^{n} F_i(\tau_1, \ldots, \tau_{\alpha_i})$$

with F_i ranging over $\mathcal{F}$. Note that this is a good definition, since $\mathcal{F}$ contains a constant. The sum is required to be dicriminative.

We introduce some shorthand notation. A vector of types $\tau_1, \ldots, \tau_k$ is sometimes written $\overline{\tau}$, its length being left implicit, to be determined by context; for instance, in the expression $F_i\overline{\tau}$ it is assumed that $k = \alpha_i$. We may index vecors, as in $\overline{\tau}_i$, which abbreviates $\tau_{i_1}, \ldots, \tau_{i_k}$. A sum whose j'th summand is $F\overline{\tau}$ is denoted $\sum[F\overline{\tau}]_j$.

Coercions

For every type constructor $F_i \in \mathcal{F}$, there is a positive, tagging coercion (injection), called $F_i^!$, and a negative, check-and-untag coercion (projection), called $F_i^?$, at each of the signatures

$F_i^! : F_i\overline{\tau} \rightsquigarrow \sum[F_i\overline{\tau}]_i$

$F_i^? : \sum[F_i\overline{\tau}]_i \rightsquigarrow F_i\overline{\tau}$

and *error-coercions* $\uparrow$, $\downarrow$ at each of the signatures

$\uparrow^{\tau} : \mathbf{E} \rightsquigarrow \tau$

$\downarrow_{\tau} : \tau \rightsquigarrow \mathbf{E}$

For every $F_i \in \mathcal{F}$ with $\alpha_i > 0$ there is a coercion constructor, also called F_i, of arity α_i. The set of coercions, ranged over by c, d, is given by

$$c \quad ::= \quad id \mid \uparrow \mid \downarrow \mid F_i^! \mid F_i^? \mid F_i(c_1, \dots, c_{\alpha_i}) \mid \textstyle\sum_{i=1}^{n} c_i$$

For every constructor F_i with $\alpha_i > 0$, signatures are assigned to co-variant constructors by the rule

$$\frac{c_i : \tau_i \rightsquigarrow \theta_i}{F_i(c_1, \dots, c_k) : F_i(\tau_1, \dots, \tau_k) \rightsquigarrow F_i(\theta_1, \dots, \theta_k)}$$

and contra-variantly, as usual, in the case of the function space constructor.

Coercion equality and conversion

The equational theory of Σ-coercions is given in Figure 4.2. As usual, there are two groups of equations, the "core" equations (called C) and a theory of $\phi\psi$-conversion. All equations are derivable in the categorical system and have already been explained in the previous section. Coercions are always considered *modulo* C, unless it is specifically said that we consider coercion *terms*, which, in that case, are identifiable only by syntactical identity. In case F is the function space constructor, the rule $[C8]$ must be read correctly, in that one must assume the constructor to work on "morphisms" from the "dual category". Alternatively, we can think of the rule as having the special case

$$(c_1 \rightarrow c_2) \circ (d_1 \rightarrow d_2) = (d_1 \circ c_1) \rightarrow (c_2 \circ d_2)$$

for the constructor $\rightarrow$.

Canonical coercions

A very important notion, not least from a pragmatic point of view, is that of canonical coercions, since it allows us to effectively choose a unique "optimal" coercion at a given signature. The canonical coercions of Σ are given in Figure 4.3, as a function κ from signatures (represented as a pair of types) to coercions, such that $\kappa(\tau, \theta) : \tau \rightsquigarrow \theta$. As can be easily verified, we have the following fundamental

Lemma 4.4.1 *At every signature $\tau \rightsquigarrow \theta$ there exists a canonical coercion $\kappa(\tau, \theta) : \tau \rightsquigarrow \theta$, which is unique* modulo C.

It is easy to see that we can modify our definition of canonicals in such a way that the canonical coercion at a given signature becomes *syntactically* unique (as in system **D**.) Doing this is left for the reader.

4.4.2 Reduction

We seek a good notion of reduction for Σ-coercions. The natural first attempt is to orient the $\phi\psi$-equations left-to-right, taken *modulo* the core equations in C, as usual. However, this results in a non-confluent system, as is shown in the example:

Example 4.4.2 Consider coercions $F^!$, $G^?$ with $F \not\equiv G$, then we have, on the one hand,

$$G^{!^\Sigma} \circ G^?_\Sigma \circ F^! \to_\psi F^!$$

and, on the other hand,

$$G^{!^\Sigma} \circ G^?_\Sigma \circ F^! \to_\phi G^! \circ \updownarrow \to_\phi \updownarrow$$

which is an unsolvable critical pair. □

This problem can be solved by performing a simple Knuth-Bendix completion ([KB67]) of the system oriented as just suggested. The result is shown in Figure 4.4.

We lift $\phi\psi$-reduction on coercions to Σ-completions, in complete analogy with the way this is done in $\mathbf{D}$. We denote these reductions $\Rightarrow_{\phi\psi}$ etc., as usual.

Discriminativity

Note the type dependency of rule $[\psi 1]$. In case we have $\Sigma \equiv \Sigma'$, we get back the "expected" rule (as in system $\mathbf{D}$):

$$F^{!^\Sigma} \circ F^?_\Sigma \to id_\Sigma$$

However, it is a consequence of the fine grained articulation of the type $\mathbf{D}$ performed in Σ, that we can have "anomalous" coercions, with $\Sigma \not\equiv \Sigma'$, as in, *e.g.*,

$$\mathtt{B!}^{\mathbf{B}+(\tau'\to\theta')} \circ \mathtt{B?}_{\mathbf{B}+(\tau\to\theta)}$$

with $\tau \not\equiv \tau'$, $\theta \not\equiv \theta'$. Therefore, one cannot just take over the $\mathbf{D}$-style ψ-reduction rule; it is not guaranteed to enjoy *subject reduction*, at the level of coercions. Note that this is *never* a problem with ϕ-reduction, so long as we require discriminativity.

If we imagine that we dropped the conditions of *discriminativity* we could have more radical situations, such as, *e.g.*,

$$\mathtt{F!}^{(\tau\to\tau')+(\theta\to\theta')} \circ \mathtt{F?}_{(\tau\to\tau')+(\eta\to\eta')}$$

or, in all generality

$$F^{!^{A+C}} \circ F^?_{A+B}$$

where B, C can be arbitrary with $B \neq C$. Among other things, it appears to be difficult to arrive at a good notion of canonical coercions (as a function of signatures) here. The desire to avoid such complications is a major motivation for working within the discriminative restriction. Whether the generalized categorical system is practically tractable or not, we do not know.

Conservativity

We conjecture that the ϕ-conversion of Σ is *complete* wrt. the general notion of "categorical" ϕ-conversion of Section 4.3; or, in other words, the categorical theory is a conservative extension of Σ. This should be taken modulo an obvious translation

$$\mathcal{I} : \Sigma \to \mathbf{Co}$$

with $\mathcal{I}[\![\pi_j]\!] \equiv \bigvee_i \varepsilon^i_j$ and $\mathcal{I}[\![\sum_i c_i]\!] \equiv \bigvee_i \mathcal{I}[\![c_i]\!] \circ \iota_i$ (we leave it for the reader to complete the definition.) More precisely, we *conjecture* that

$$\Sigma \vdash \mathcal{I}^{-1}[\![c]\!] =_\phi \mathcal{I}^{-1}[\![d]\!]$$

whenever $c, d \in \mathcal{I}(\Sigma)$ and

$$\mathbf{Co} \vdash c =_{\phi} d$$

We have so far no proof of this conjecture. Note that, by Section 4.3, we have the "soundness property"

$$\Sigma \vdash c =_{\phi} d \Rightarrow \mathbf{Co} \vdash \mathcal{I}[\![c]\!] =_{\phi} \mathcal{I}[\![d]\!]$$

Properties of reduction

A good sign that we have a reasonable notion of reduction is that we can prove the following fundamental Theorem 4.4.4. The whole matter is really concentrated in the following lemma:

Lemma 4.4.3 *For all Σ-types τ, θ, η,*

$$\Sigma(\phi\psi) \vdash \kappa(\tau, \theta) \circ \kappa(\eta, \tau) \to^{*} \kappa(\eta, \theta)$$

Proof The proof is essentially by induction on the generation of $\kappa(\tau, \theta)$, $\kappa(\eta, \tau)$. It is contained in Appendix B. □

Theorem 4.4.4 *For every Σ-coercion $c : \tau \rightsquigarrow \theta$,*

$$\Sigma(\phi\psi) \vdash c \to^{*} \kappa(\tau, \theta)$$

Moreover, this property fails *for any proper subset of the $\phi\psi$-rules.*

Proof The proof that every coercion c reduces to the canonical coercion at the signature is by induction on the coercion term c, using the previous lemma:

Base cases. The base case where c is primitive is clear.

Induction. In case $c \equiv F_i(c_1, \ldots, c_n)$, we first note that c must have a signature of the form $F_i\overline{\tau}_i \rightsquigarrow F_i\overline{\theta}_i$, with $c_i : \tau_{i_j} \rightsquigarrow \theta_{i_j}$. The result now follows by inductive hypothesis, which yields the coercion $F_i(\kappa_1, \ldots, \kappa_n)$, canonical at the signature of the above form.

In case $c \equiv \sum_i c_i$, we get by induction the coercion $\sum_i \kappa_i$, canonical at the appropriate signature. Note the case where all κ_i are trivial, in which case we use $[C9]$, $\sum_i id = id$, canonical.

In case $c \equiv c_1 \circ c_2$, we know by induction hypothesis that c_1 reduces to κ_1, canonical, and c_2 reduces to κ_2, also canonical. Hence, $c_1 \circ c_1$ reduces to $\kappa_1 \circ \kappa_2$. Now apply Lemma 4.4.3. □

Canonical coercions are extremely important for practical purposes, but they also play a useful part in theory, since they provide induction parameters for various properties. The theorem above is quite powerful, since it entails

Corollary 4.4.5 *In Σ, $\phi\psi$ reduction is weakly normalizing and confluent.*

Proof First consider the normalization property. It is easy to verify, by induction on the generation of canonical coercions, that every canonical is a $\phi\psi$-normal form. Now use the previous theorem.

Next, consider the confluence claim. Suppose we have two reducts c_1, c_2 of the same coercion c. Then, since reduction evidently preserves signatures, c_1 and c_2 must have the same signature; by the preceeding theorem they both reduce to the canonical coercion at their signature, which is unique up to C-equality. Since $\phi\psi$-reduction is *modulo C*, it follows that the canonical is a common reduct. □

We conjecture that $\phi\psi$-reduction on Σ-coercions is in fact strongly normalizing, but we shall not attempt to prove that. Of greater interest to us is that ϕ-reduction *is* strongly normalizing, as is easily seen:

Proposition 4.4.6 *In Σ, ϕ-reduction on coercions is strongly normalizing.*

PROOF The reduction decreases the number of non-trivial prime coercions. □

We may note that with

$$\updownarrow^{\theta}_{\tau} \rightarrow_{\psi} \kappa(\tau, \theta)$$

the orientation $\rightarrow_{\phi\psi}$ of the error-coercions will verify [ψ1], when the coercions are transported back into the categorical framework and the projections are taken as *derived* notions, since we have

$$\begin{array}{rcl} \iota_A^{A+B} \circ \pi^A_{A+B} & \equiv & \iota_A^{A+B} \circ [id_A, \updownarrow^A_B] \\ & = & [\iota \circ id, \iota \circ \updownarrow] \\ & \rightarrow_{\phi} & [\iota, \updownarrow^{A+B}_A] \\ & \rightarrow^*_{\psi} & [\iota, \iota] \\ & = & id \end{array}$$

where we use that the injection $\iota : A \rightsquigarrow A + B$ is canonical at the signature and hence reachable by ψ-reduction.

We have previously noted that, in the presence of the error-equations, it is trivial that the category has the coherence property. It is natural to ask whether this still holds, in the conversion theory of Σ, if we *remove* the error-coercions. After all, the corresponding property holds in **D**. Here we meet again a kind of "anomaly", which stems from the fact that we have "more" coercions in Σ than we have in **D**. Consider the Σ-coercions

$\mathtt{B?} \circ \mathtt{F!} \circ (c^{\tau}_{\theta} \rightarrow d^{\theta'}_{\tau'})$

$\mathtt{B?} \circ \mathtt{F?}$

both of which can be seen at the signature $(\tau \rightarrow \tau') \rightsquigarrow \mathbf{B}$. However, to identify these two coercions we need the error (ϕ) reductions.

4.4.3 Completions

The reductions on Σ-coercions can be lifted to Σ-completions in analogy with the way completion reduction is defined in **D** from reductions on **D**-coercions. The equational theory E makes sense for Σ-completions [6] *with the proviso* that we be explicit about the signatures of coercions in the non-linear rules. To see the significance of this, consider as an example the usual E-rule

$$[c \rightarrow d](\lambda x : \tau.M) = \lambda x : \tau'.[d]M\{x := [c]x\}$$

So far we have not cared to write the signatures of coercions explicitly, since it would automatically be the case that one side of this equation is well-typed, just in case the other side is. This

[6]Taking the coercions in the E-equations to range over all Σ-coercions, and taking the theory C (which is used to build E) to be the theory C of Σ, of course.

Remarks such as this one are often strictly necessary when "re-using" parts of a restricted framework in the definition of an extension; we shall not make such remarks explicitly any more, since it is really obvious how to understand the slightly sloppy, but less cumbersome, mode of expression where such remarks are left out.

no longer holds in Σ, where the signature of a coercion name, such as, *e.g.*, $F^!$, is not unique. Here the rule really must be read

$$[c^{\tau}_{\tau'} \to d^{\theta'}_{\theta}](\lambda x : \tau.M) = \lambda x : \tau'.[d^{\theta'}_{\theta}]M\{x := [c^{\tau}_{\tau'}]x\}$$

for reasons of type preservation.

We can consider $\phi\psi$-reduction on Σ-completions which, as usual, is just the coercion reduction in context, taken *modulo E*.

In the remainder of this subsection we use the results of Section 2.2 to establish that there is a confluent polarization of ϕ-reduction on Σ-completions.

First let us note that we can define a notion of *polarity* for Σ, in close analogy with system **D**. We extend positive and negative coercions inductively from the base cases:

$F^! : +$

$F^? : -$

$\uparrow : +, \downarrow : -$

using the rules for induced coercions:

$$\frac{c_i : \diamond \text{ for } i = 1 \dots k}{F(c_1, \dots, c_k) : \diamond}$$

$$\frac{c_i \diamond \ \text{ for } i = 1 \dots k}{\sum_i c_i : \diamond}$$

where $\diamond$ is fixed to be either $+$, $-$, and where polarities extend contravariantly (in the first argument) in case the functor F is the arrow. Note the *absence* of neutral coercions. They turn out to be unnecessary in the presence of the explicit error-coercions $\uparrow$, $\downarrow$, since we can prescribe a notion of reduction on compositions which were classified as neutral in **D**.

Let us also note that the notion of *prime factoring* is definable in Σ, also completely analogously to **D**. We note here that factorization of sum-coercions $\sum_i c_i$ can be obtained using rule $[C12]$ of the coercion equality in Σ. It can be seen without much difficulty that the factorization results of Section 2.1 for **D** can be transferred to Σ.

We can define *polarized* reductions for Σ-completions, as in Section 2.2.2 (see Figure 2.2) However, to get a well-behaved system it turns out to be necessary to impose a slight restriction on the reduction rules for error-coercions. To see what motivates this, consider what happens if we simply define the polarized reduction P_ϕ for system Σ to be the union of its ϕ-reduction on coercions and the polarized completion equality P of Figure 2.2:

Example 4.4.7 Let

$$M \equiv \lambda x : \mathbf{B} + \tau. \dots [\downarrow \circ \mathtt{B}?]x \dots [\mathtt{B}?]x \dots$$

Then we have the reductions

$$M \Rightarrow^*_{P_\phi} \lambda x : \mathbf{B} + \tau. \dots [\downarrow]x \dots [\mathtt{B}?]x \dots$$

and

$$M \Rightarrow^*_{P_\phi} [\mathtt{B}? \to id](\lambda x : \mathbf{B}. \dots [\downarrow]x \dots x \dots)$$

with no common reduct under $P_\phi*$ of the two reducts shown. □

Hence, the suggested polarization of ϕ reduction on *completions* is *not* confluent. However, this is solely due to the unrestricted abortive nature of error-coercions, and there is a very natural restriction of it which lives well with the polarized equality. Noting that the essence of ϕ-reduction in system **D** is just the elimination of positive-negative compositions $c^- \circ c^+$ (apart from the error-generating ones) it is natural to restrict the error-reductions in the same way, obtaining the polarized reduction for Σ shown Figure 4.5.

Remark 4.4.8 We speculated earlier (at the end of Section 4.3) that one could define a more general notion of "categorical" ϕ-reduction. However, it appears that is is difficult to obtain a natural notion of *polarity* of coercions in this general system. We have not investigated this in full depth, though. $\square$

4.4.4 Factorization and confluence

It is easy to see that the polarized notion of ϕ-reduction on coercions in Figure 4.5 is locally confluent and terminating, hence also confluent. Moreover, as already stated, the Σ-reduction P_ϕ does *not* include the equations for neutrals (which are present in the polarized reduction for **D**.) This simplification is possible because of the following Proposition 4.4.11, which shows that, due to the existence of error-coercions, Σ has a stronger form of coercion factorization under ϕ-reduction: Every Σ-coercion factors into a negative-positive composition under ϕ, $c \to_\phi^* c_1^+ \circ c_2^-$. In the remainder of this section, ϕ reduction on coercions is that of Figure 4.5. The main technical lemma follows.

Lemma 4.4.9 *If $d^- \circ c^+$ is a ϕ-normal form then there are a^+, a^- such that*

$$C \vdash d^- \circ c^+ = a^+ \circ a^-$$

Proof By structural induction on c^+.

$\boxed{c^+ \equiv p^+}$ We prove

$$C \vdash d^- \circ p^+ = a^+ \circ a^-$$

with a^+ primitive, by structural induction on d^-:

Case $d^- \equiv p^-$. Here d^- must be the identity (satisfying claim), since otherwise $p^- \circ p^+$ must be a ϕ-redex, contrary to assumption.

Case $d^- \equiv d_1^- \circ d_2^-$. By induction we have $d^- \circ c^+ = d_1^- \circ d_2^- \circ p^+ = d_1^- \circ (a^+ \circ a^-)$, with a^+ primitive. Then another application of the induction hypothesis, to $d_1^- \circ a^+$, yields the claim.

Case $d^- \equiv d_1^+ \to d_2^-$. In this case the domain type of d^- must be a function type, and so the range type of p^+ is a function type. There is only one primitive (non-trivial) positive coercion with a function type in its range, $p^+ \equiv \uparrow$. So, if d^- is not trivial, $d^- \circ c^+$ contains a ϕ-redex, contrary to assumption; if it is trivial, the claim is true.

Case $d^- \equiv \sum_i d_i^-$. In this case we must have $p^+ = F_j^!$ (unless p^+ trivial), by inspection of the form of signatures. Therefore,

$$\begin{aligned} d^- \circ c^+ &= (\textstyle\sum_i d_i^-) \circ F_j^! \\ &= F_j^! \circ d_j^- \\ &= p^+ \circ d_j^- \end{aligned}$$

which has the desired form. We used $[C10]$ in the derivation.

$\boxed{c^+ \equiv c_2^+ \circ c_1^+}$ Here we have $d^- \circ c^+ = (d^- \circ c_2^+) \circ c_1^+$, and by induction we get $d^- \circ c^+ = (a^+ \circ a^-) \circ c_1^+ = a^+ \circ (a^- \circ c_1^+)$. If a^- is trivial, we are finished. Otherwise, using induction hypothesis again we get $a^- \circ c_1^+ = b^+ \circ b^-$, and our claim is seen to hold.

$\boxed{c^+ \equiv c_1^- \to c_2^+}$ If d^- is trivial, we are finished. If d^- is not trivial, we reason as follows :

Suppose first that $d^- = d_1^+ \to d_2^-$. Then, by induction, we have

$$\begin{aligned} d^- \circ c^+ &= (d_1^+ \to d_2^-) \circ (c_1^- \to c_2^+) \\ &= (c_1^- \circ d_1^+) \to (d_2^- \circ c_2^+) \\ &= (a_1^+ \circ a_1^-) \to (a_2^+ \circ a_2^-) \\ &= (a_1^- \to a_2^+) \circ (a_1^+ \to a_2^-) \end{aligned}$$

and the claim is seen to be satisfied.

Next, suppose d^- is not equal to an arrow. Then, by signatures, we must have $d^- = \downarrow \circ d_1^-$ where d_1^- is either trivial or equal to an arrow. In these cases we have, as above

$$d_1^- \circ c^+ = (a_1^- \to a_2^+) \circ (a_1^+ \to a_2^-)$$

from which we get

$$\downarrow \circ d_1^- \circ c^+ = \downarrow \circ (a_1^- \to a_2^+) \circ (a_1^+ \to a_2^-)$$

The last coercion contains a ϕ-redex, contrary to our assumptions. Hence, the claim is vacuously true in this case.

$\boxed{c^+ \equiv \sum_i c_i^+}$ In case d^- is primitive, non-trivial, it must be of the form $d^- = F_j^?$, and hence

$$\begin{aligned} d^- \circ c^+ &= F_j^? \circ \textstyle\sum_i c_i^+ \\ &= c_j^+ \circ F_j^? \end{aligned}$$

with the desired form. We used equation [$C11$].

In case d^- is not primitive, it must (by inspection of the form of signatures) have the form $d^- = \sum_i d_i^-$, and so we have, using induction hypothesis,

$$\begin{aligned} d^- \circ c^+ &= (\textstyle\sum_i d_i^-) \circ (\sum_i c_i^+) \\ &= \textstyle\sum_i (d_i^- \circ c_i^+) \\ &= \textstyle\sum_i (a_i^+ \circ a_i^-) \\ &= (\textstyle\sum_i a_i^+) \circ (\sum_i a_i^-) \end{aligned}$$

which is again on the desired form. We used [$C12$] in the derivation.

□

Lemma 4.4.10 *For every ϕ normal form c there are a^+, a^- such that*

$$C \vdash c = a^+ \circ a^-$$

Proof By structural induction on c. For c primitive the matter is clear. For $c \equiv c_1 \circ c_2$ we have by induction that $c = c_1^+ \circ c_1^- \circ c_2^+ \circ c_2^-$. By Lemma 4.4.9 we have $c_1^- \circ c_2^+ = a^+ \circ a^-$, and so $c = c_1^+ \circ a^+ \circ a^- \circ c_2^-$ which satisfies the claim. For $c \equiv c_1 \to c_2$ we have by induction that $c = (c_1^+ \circ c_1^-) \to (c_2^+ \circ c_2^-) = (c_1^- \to c_2^+) \circ (c_1^+ \to c_2^-)$ which satisfies the claim. Finally, for $c \equiv \sum_i c_i$, we have by induction $c = \sum_i (a_i^+ \circ a_i^-) = (\sum_i a_i^+) \circ (\sum_i a_i^-)$ which satisfies the claim. □

Proposition 4.4.11 *($+ \circ -$ Factoring) For every coercion c there are a^+, a^- such that*

$$c \to_\phi^* a^+ \circ a^-$$

Proof Take the ϕ-normal form of c, and apply Lemma 4.4.10. □

Using our notion of polarity for Σ together with Proposition 4.4.11, we can carry out the whole confluence argument of Chapter 2 for polarized reduction in **D**. The development for Σ is completely parallel, with a single exception (where the argument for Σ is *simplified*): Recall from the argument of Chapter 2 that the neutral equations (of the relation P_ϕ in **D**) were used just to resolve critical pairs of the type *III* (2) as described in the proof of Proposition 2.2.2. With the factoring property of Proposition 4.4.11, we can see that these equations are no longer necessary, for the critical pairs (of the type mentioned) can now be resolved using our stronger ϕ-reduction, which does not include the neutrals. In fact, confluence of P_ϕ in Σ follows, by Newman's Lemma, from the simplified analysis of Proposition 2.2.2, since it turns out that P_ϕ is strongly normalizing in Σ (see Theorem 5.4.4.) We can confine ourselves to recording:

Theorem 4.4.12 *In Σ:*

1. *ϕ-reduction on Σ-coercions is canonical (confluent, terminating)*
2. *P_ϕ-reduction on Σ-completions is confluent over λI*

Remark 4.4.13 It is clear that we can define an analogous notion of ϕ-reduction including error-coercions for system **D**. In this case also, we can polarize ϕ-reduction to a confluent relation *without* using the equations for neutral coercions. □

4.4.5 Polymorphic coercions and maximal sums

Having developed the Σ-systems from a categorical basis we have uniformly assumed *type-indexed families* of *monomorphic* coercions. For instance, we have assumed a family of primitives

$F^! : F(A_1, \ldots, A_k) \rightsquigarrow \sum_i [F(A_1, \ldots, A_k)]$

$F^? : \sum_i [F(A_1, \ldots, A_k)] \rightsquigarrow F(A_1, \ldots, A_k)$

Such a family can be regarded, at the *meta-level* of the calculus, as emerging by instantiation into polymorphic type schemes

$F^! : \forall \alpha^{(k)} \beta . F(\alpha_1, \ldots, \alpha_k) \rightsquigarrow F(\alpha_1, \ldots, \alpha_k) + \beta$

$F^? : \forall \alpha^{(k)} \beta . F(\alpha_1, \ldots, \alpha_k) + \beta \rightsquigarrow F(\alpha_1, \ldots, \alpha_k)$

where the notation $\tau^{(k)}$ abbreviates the vector $\tau_1, \ldots, \tau_k$, with the tacit assumption that k equals the arity of the constructor applied to the type vector. The polymorphic view is subject to the condition:

- *It is tacitly assumed that only such instantiations shall be made which respect the discriminativity condition.*

We can *internalize* this meta-level polymorphism in the system itself by adding the usual Hindley-Milner style *type schemes* (ranged over by σ) to the system:

$$\sigma ::= \tau \mid \forall \alpha . \sigma$$

extending the typing rules for Σ-completions with standard rules for type abstraction and instantiation, regarding the coercions as *polymorphic constants*. We shall henceforth feel free to assume this for the Σ-systems. In the present chapter we have additionally assumed that every instance of a sum considered is *maximal* in the sense that it contains just n summands, where n is the size of the constructor alphabet. If n is held fixed over all programs considered, this amounts to degenerating the sum to a *free* operator, a *constructor*, as it were, of arity n.

Coercion equality (C)

$$
\begin{array}{lcll}
c \circ id & = & c & [C1] \\
id \circ c & = & c & [C2] \\
c \circ (c' \circ c'') & = & (c \circ c') \circ c'' & [C4] \\
\uparrow^{\mathbf{E}} & = & id_{\mathbf{E}} & [C5] \\
\downarrow_{\mathbf{E}} & = & id_{\mathbf{E}} & [C6] \\
id_{F_i(\tau_1,\ldots,\tau_k)} & = & F_i(id_{\tau_1},\ldots,id_{\tau_k}) & [C7] \\
F_i(c_1,\ldots,c_k) \circ F_i(d_1,\ldots,d_k) & = & F_i(c_1 \circ d_1,\ldots,c_k \circ d_k) & [C8] \\
\\
\sum id & = & id & [C9] \\
(\sum_i c_i) \circ F^!_j & = & F^!_j \circ c_j & [C10] \\
F^?_j \circ \sum_i c_i & = & c_j \circ F^?_j & [C11] \\
(\sum_i c_i) \circ (\sum_i d_i) & = & \sum_i (c_i \circ d_i) & [C12]
\end{array}
$$

Coercion conversion $(\phi\psi)$

$$
\begin{array}{lcll}
F^? \circ F^! & = & id & [\phi 1] \\
G^? \circ F^! & = & \updownarrow \text{ for } F \not\equiv G & [\phi 2] \\
c^\theta_\tau \circ \uparrow^\tau & = & \uparrow^\theta & [\phi 3] \\
\downarrow_\theta \circ c^\theta_\tau & = & \downarrow_\tau & [\phi 4] \\
\\
F^{!^\Sigma} \circ F^?_\Sigma & = & id_\Sigma & [\psi 1] \\
\updownarrow^\tau_\tau & = & id_\tau & [\psi 2]
\end{array}
$$

Figure 4.2: Coercion equality and conversion of system Σ

$$\begin{array}{lcll}
\kappa(\tau,\tau) & = & id_\tau & [\kappa1] \\
\kappa(F_i\overline{\tau}, F_j\overline{\theta}) & = & \updownarrow \text{ for } i \neq j & [\kappa2] \\
\kappa(F_i\overline{\tau}, F_i\overline{\theta}) & = & F_i(\kappa(\tau_1,\theta_1),\ldots,\kappa(\tau_k,\theta_k)) & [\kappa3] \\
\kappa(F_j\overline{\theta}, \sum_i F_i\overline{\tau}) & = & F_j^! \circ \kappa(F_j\overline{\theta}, F_j\overline{\tau}) & [\kappa4] \\
\kappa(\sum_i F_i\overline{\tau}, F_j\overline{\theta}) & = & \kappa(F_j\overline{\tau}, F_j\overline{\theta}) \circ F_j^? & [\kappa5] \\
\kappa(\sum_i F_i\overline{\tau}, \sum_i F_i\overline{\theta}) & = & \sum_i \kappa(F_i\overline{\tau}, F_i\overline{\theta}) & [\kappa6]
\end{array}$$

Figure 4.3: Canonical coercions of system Σ

ϕ reduction

$$\begin{array}{lcll}
F^? \circ F^! & \to & id & [\phi1] \\
G^? \circ F^! & \to & \updownarrow & [\phi2] \\
c_\tau^\theta \circ \uparrow^\tau & \to & \uparrow^\theta & [\phi3] \\
\downarrow_\tau \circ c_\theta^\tau & \to & \downarrow_\theta & [\phi4]
\end{array}$$

ψ reduction

$$\begin{array}{lcll}
F^{!^{\Sigma'}} \circ F_\Sigma^? & \to & \kappa(\Sigma,\Sigma') & [\psi1] \\
\updownarrow_\tau^\tau & \to & id_\tau & [\psi2] \\
\updownarrow_{F\overline{\tau}}^{F\overline{\theta}} & \to & F(\updownarrow_{\tau_1}^{\theta_1},\ldots,\updownarrow_{\tau_n}^{\theta_n}) & [\psi3] \\
\updownarrow_{F_j\overline{\tau}}^{\sum_i F_i\overline{\theta}} & \to & F_j^! \circ F_j(\updownarrow_{\tau_1}^{\theta_1},\ldots,\updownarrow_{\tau_n}^{\theta_n}) & [\psi4] \\
\updownarrow_{\sum_i F_i\overline{\tau}}^{F_j\overline{\theta}} & \to & F_j(\updownarrow_{\tau_1}^{\theta_1},\ldots,\updownarrow_{\tau_n}^{\theta_n}) \circ F_j^? & [\psi5] \\
\updownarrow_{\sum_i F_i\overline{\tau}}^{\sum_i F_i\overline{\theta}} & \to & \sum_i \updownarrow_{F_i\overline{\tau}}^{F_i\overline{\theta}} & [\psi6]
\end{array}$$

The reductions are *modulo* the equations in C.

Figure 4.4: Coercion reductions ($\phi\psi$) of system Σ

Polarized ϕ reduction (P_ϕ)

$$
\begin{array}{lcll}
F^? \circ F^! & \rightarrow & id & [\phi 1] \\
G^? \circ F^! & \rightarrow & \updownarrow & [\phi 2] \\
c^- \circ \uparrow & \rightarrow & \uparrow & [\phi 3] \\
\downarrow \circ c^+ & \rightarrow & \downarrow & [\phi 4]
\end{array}
$$

Side conditions:
In $[\phi 3]$ and $[\phi 4]$ c^- must be properly negative and c^+ must be properly positive.

The polarized ϕ reduction on Σ-completions is the union of ϕ as given above and the P-rules of Figure 2.2. The reduction ϕ on coercions is *modulo* C.

Figure 4.5: Polarized ϕ reduction for Σ

4.5 Recursive systems

In this section we define extensions of Σ which include *regular recursive types*. We consider two variants. The first system, which will be called Σ^μ, contains explicit coercions (unfold and fold) which coerce a recursive type to its syntactic unfolding and *vice-versa*, respectively. It gives rise to a completion calculus (by obvious extension of Σ) into which system **D** can be embedded. The second system, to be called Σ^∞, is stronger; it keeps type recursion implicit and is equivalent to an infinitary type system in which recursive types whose infinite unfoldings are equal can be considered equivalent. The resulting systems can be viewed as a *polymorphic generalization* of system **D**, which is strong enough to model a form of *soft typing*. They can be seen as a syntactic articulation of the semantic equation

$$\mathbb{D} \cong [\mathbb{D} \to \mathbb{D}] \oplus \mathbb{E} \oplus \mathbb{B} \ldots$$

underlying our understanding of **D**, in that they articulate type **D** explicitly into a sum-component and a recursion-component. The relation to soft type systems is considered in Chapter 6.

4.5.1 Regular recursive types

We add recursive types to Σ by extending the definition of Σ-types with *recursive type abstraction*

$$\tau ::= \ldots \mid \alpha \mid \mu\alpha.\tau$$

where α ranges over *type variables.*

Recursive type abstractions are finite notations for *regular trees*, meaning that the set of distinct subtrees of the complete (possibly infinite) unfolding of the type is finite. See [Cour83], [CC91] for all the details; we confine ourselves to a brief review of some main points here.

We consider the set T_R of (finite and infinite) *regular trees* over a given ranked alphabet of type constructors. The set T_R is organized as an ω-complete partial order, as follows. There is a special element, called Ω, which is the least element, representing empty information. The set of trees is partially ordered by extending $\Omega \leq \zeta$ (for every tree ζ) co-variantly over the constructors.

The (possibly infinite) regular tree $(\mu\alpha.\tau)^*$ associated with a recursive abstraction $\mu\alpha.\tau$, the *unfolding* of $\mu\alpha.\tau$, is the least fixed point, over the ordered set T_R of all regular trees, of the functional which sends a regular type ζ to the unfolding of τ by ζ :

$$(\mu\alpha.\tau)^* = \mathbf{fix}(\boldsymbol{\lambda}\zeta \in T_R.\tau^*\{\alpha := \zeta\})$$

If F denotes the functional associated with a recursive abstraction (as mentioned above), the least fixed point can be obtained as the limit of the sequence of n-fold iterations of F on Ω:

$$\mathbf{fix}\, F = \bigsqcup_{n=0}^{\infty} F(\Omega)$$

The full definition of $\bullet^*$ is as follows, by recursion in the structure of the type expression. It associates to each notation τ an element $(\tau)^* \in T_R$:

$$\begin{array}{rcl}
(\alpha)^* &=& \alpha \\
(tc(\tau_1, \ldots, \tau_k))^* &=& tc((\tau_1)^*, \ldots, (\tau_k)^*) \\
(\sum_i \tau_i)^* &=& \sum_i (\tau_i)^* \\
(\mu\alpha.\tau)^* &=& \mathbf{fix}(\boldsymbol{\lambda}\zeta \in T_R.(\tau)^*\{\alpha := \zeta\})
\end{array}$$

The type inference systems we shall be concerned with later in this work will only accept types whose unfolding carries information; one has for instance $(\mu\alpha.\alpha)^* = \Omega$, and no expression will be assigned this type.

4.5.2 System Σ^μ

System Σ^μ contains two new primitive coercions, called the *(type) recursion coercions*, **f** (fold) and **u** (unfold) with the signatures

$\mathbf{f} : \tau\{\alpha := \mu\alpha.\tau\} \rightsquigarrow \mu\alpha.\tau$

$\mathbf{u} : \mu\alpha.\tau \rightsquigarrow \tau\{\alpha := \mu\alpha.\tau\}$

The coercion language of Σ^μ arises by extending that of Σ with these new primitives as additional base cases of the definition of coercions. The resulting completion calculus can be seen as an explicit typing proof system corresponding to a type system with the rule

$$\frac{\Gamma \vdash M : \tau \quad \tau \sim \tau'}{\Gamma \vdash M : \tau'} \qquad [\sim]$$

where one defines $\sim$ to be the syntactic unfolding relation generated (by reflexive, symmetric, transitive, compatible closure) from

$$\mu\alpha.\tau \sim \tau\{\alpha := \mu\alpha.\tau\}$$

This is usually combined with the restriction that only *formally contractive* type expressions are allowed. Formal contractiveness is a syntactic criterion which ensures that only such type expressions can be built, which can be given a meaning in the ideal model of [MPS86]. This excludes non-sensical abstractions like $\mu\alpha.\alpha$. See [MPS86] for the details. See also further comments on (the weakness of) $\sim$ below, in Section 4.5.3.

We assign *polarities* to the recursion coercions, by the rules

$\mathbf{f} : +$

$\mathbf{u} : -$

These rules are to be taken in continuation of the polarity assignment of Σ. We define the μ-reductions on coercions:

$$\mathbf{u} \circ \mathbf{f} \to id \qquad [\mu\phi]$$

$$\mathbf{f} \circ \mathbf{u} \to id \qquad [\mu\psi]$$

which are obvious analogues of ϕ and ψ-reductions of Σ, respectively. The extended ϕ-reduction on Σ^μ-coercions is the union of the ϕ-rules of Σ with the rule $[\mu\phi]$. The extended reductions on Σ^μ-coercions are lifted to corresponding reductions on Σ^μ-completions in the obvious way. Furthermore, in the presence of an extended assignment of polarities to Σ^μ-coercions we can define *polarized* ϕ-reduction, also to be called P_ϕ (ambiguities in notation are resolved by context), on Σ^μ-completions.

Likewise, we can consider just the reduction relation induced by $[\mu\phi]$. Clearly, this relation can be lifted to completions and it can be polarized in the usual fashion; it is called $\mu\phi$-reduction. We refer to the polarized version as simply μ-reduction. These relations are obvious analogues to the usual ϕ-reductions, only they concern just the recursion coercions. Note that the possibility of creating *run-time errors*, known from the usual ϕ-reductions, is not present under $\mu\phi$- or μ-reduction. These reductions are interesting for a number of reasons. One is that, since Σ^μ factors into Σ and μ:

$$\Sigma^\mu(\phi) = \Sigma(\phi) \cup \mu\phi$$

we can analyse properties of Σ^μ in terms of properties of its natural two components, the sum-component (Σ) and the type recursion component (the μ-rules.) For instance, we shall be able (Chapter 5) to establish that the complexity of unrestricted ϕ-reduction in Σ^μ is precisely due to certain properties of the recursion rules (and not the sum-rules.) Moreover, as we shall see below, system **D** sits inside Σ^μ:

$$\mathbf{D} \hookrightarrow \Sigma^\mu$$

and in this sense system Σ^μ gives an analysis of system **D**. From this perspective, it can be seen formally that system **D** really contains two aspects that are logically quite distinct, namely one concerning sums and another concerning type recursion; only, in **D** these aspects are not explicitly articulated as they are in its generalization, system Σ^μ.

The reader who has followed our confluence arguments closely will have no difficulty in realizing that these arguments transfer to the large system Σ^μ. Hence we record

Theorem 4.5.1 *In Σ^μ:*

1. *ϕ-reduction on Σ^μ-coercions is canonical (confluent, terminating)*
2. *P_ϕ-reduction on Σ^μ-completions is confluent over λI*
3. *The recursion reductions $\mu\phi$ and μ on Σ^μ-completions are confluent over λI*

4.5.3 System Σ^∞

The recursive system Σ^∞ is obtained by adding to the type system of Σ the rule

$$\frac{\Gamma \vdash M : \tau \quad \tau \approx \theta}{\Gamma \vdash M : \theta} \qquad [\approx]$$

where the equivalence relation $\approx$ is defined by

$$\tau \approx \tau' \text{ if and only if } \tau^* = \theta^*$$

In other words, two types are equivalent if and only if their infinite unfoldings as regular trees are equal. Again, see [CC91] for details concerning this relation. It is, however, worth while to note the following:

- It has been demonstrated in [CC91] that the notion of *principal type* (see [Hin69], [DM82]) *fails* for the simply typed λ-calculus extended with recursive types using [$\sim$], whereas it *holds* for the same system under rule [$\approx$].
- The relation $\approx$ is strictly stronger than $\sim$, in particular, one has

  $$\mu\alpha.\tau \approx \tau\{\alpha := \mu\alpha.\tau\}$$

  The reader may wish to verify that taking, *e.g.*,

  $$\tau \equiv \mu\alpha.(\alpha \to \beta) \qquad \tau' \equiv \mu\alpha.((\alpha \to \beta) \to \beta)$$

  one has $\tau \approx \tau'$, but *not* $\tau \sim \tau'$. The reason is that τ and τ' become identical only in the limit, at infinity. Again, see [CC91] for details.

The first point is a *very* compelling reason for choosing $\approx$ over $\sim$. This is the more so, since $\approx$ is actually the relation decided by the usual algorithm adopted for systems with recursive types: One simply removes the *occurs check* from the unification algorithm (see, *e.g.*, [ASU86], section 6.7) Systems using $\approx$ are usually (as is Σ^∞) equivalent to an infinitary type system which has no special notation for recursive type abstractions, using instead the (possibly infinite) regular trees directly (and the typing rules remain unchanged.) Note that, since $\approx$ is stronger than $\sim$, Σ^μ embeds into Σ^∞.

Conversion and reduction

The distinctive feature of Σ^∞ as compared to Σ^μ is that, in the former system, the type recursion operations are *logical* or non-structural; in Σ^∞ we have no explicit type recursion coercions. Therefore, the equational theories (C, E, $\phi\psi$) of Σ^∞ are to be taken in union with the relation $\approx$ lifted to completions (by compatible closure), and we write $M \approx M'$, where we require that only such transformations under $\approx$ are allowed which preserve well typing. For instance, we have

$$\lambda x : \mu\alpha.(\alpha \to \alpha).x\,x \approx \lambda x : (\mu\alpha.(\alpha \to \alpha)) \to (\mu\alpha.(\alpha \to \alpha)).x\,x$$

Similarly, the completion reductions (ϕ and P_ϕ) are to be taken *modulo* $T \cup \approx$, whenever the corresponding relations in Σ^μ are taken *modulo* a theory T.

4.6 Embedding of D

It is straight-forward to show that $\mathbf{D}$ embeds into Σ^μ (and hence also into Σ^∞) All we have to realize is that a monomorphic universal type is definable in terms of summation and recursion.

Define the translation $\bullet^\mu$ on types, sending a $\mathbf{D}$-type to a Σ^μ-type, by :

$$\begin{array}{lll} \mathbf{D}^\mu & \equiv & \mu\alpha.\mathbf{B} + (\alpha \to \alpha) \\ \mathbf{B}^\mu & \equiv & \mathbf{B} \\ (\tau \to \sigma)^\mu & \equiv & \tau^\mu \to \sigma^\mu \end{array}$$

Then we have

Lemma 4.6.1 *In Σ^μ, one has*

$\mathbf{f} \circ \mathtt{F}! : (\mathbf{D}^\mu \to \mathbf{D}^\mu) \rightsquigarrow \mathbf{D}^\mu$

$\mathtt{F}? \circ \mathbf{u} : \mathbf{D}^\mu \rightsquigarrow (\mathbf{D}^\mu \to \mathbf{D}^\mu)$

$\mathbf{f} \circ \mathtt{B}! : \mathbf{B} \rightsquigarrow \mathbf{D}^\mu$

$\mathtt{B}? \circ \mathbf{u} : \mathbf{D}^\mu \rightsquigarrow \mathbf{B}$

PROOF Straight-forward. □

Hence, we can translate any $\mathbf{D}$-coercion c to a Σ^μ-coercion c^μ : We send the $\mathbf{D}$-coercion $\mathtt{F}!$ to the Σ^μ-coercion $\mathbf{f} \circ \mathtt{F}!$, and so forth for the other primitives; this translation on primitives is extended in obvious fashion to constructed $\mathbf{D}$-coercions (we leave the details for the reader) such that

Lemma 4.6.2 *For every coercion c : $\mathbf{D} \vdash c : \tau \rightsquigarrow \theta$ implies $\Sigma^\mu \vdash c^\mu : \tau^\mu \rightsquigarrow \theta^\mu$.*

PROOF By structural induction on c, using the previous lemma. $\square$

For an assumption set Γ assigning **D**-types to variables, we let $\Gamma^\mu = \{x : \tau^\mu \mid x : \tau \in \Gamma\}$. Then we immediately get the embedding property:

Proposition 4.6.3 *(Embedding)*
If $\Gamma \vdash_{\mathbf{D}} M : \tau$ *then* $\Gamma^\mu \vdash_{\Sigma^\mu} M : \tau^\mu$

PROOF By induction on the derivation of $\Gamma \vdash_{\mathbf{D}} M : \tau$, using the previous lemma. $\square$

In stating the proposition above we have been slightly sloppy, because we ought really to apply the $\bullet^\mu$ translation to all type annotations on the bound variables within the term M. For notational simplicity, we leave out explicit mention of this operation.

As a consequence of the embedding result, every **D**-completion can be regarded as a Σ^μ-completion. Naturally, the completion may well have a more interesting Σ^μ-type than just the type τ^μ, where τ is the type of the corresponding **D**-completion. The Σ^μ-typing of a **D**-completion will expectedly be much more informative in general, than the corresponding **D**-typing. Consider as a very simple example the **D**-completion

$$M \equiv \lambda x : \mathbf{D}.(\mathbf{if}\; \mathit{true}\, x\; ([\mathsf{B}!]\mathit{false}))$$

It has type $\mathbf{D} \to \mathbf{D}$ in the system **D**. Hence, knowing only the type of this object, we know nothing more than that it is a function, taking a tagged object to a tagged object. However, the same completion has the Σ^μ type

$$\forall \alpha.(\mathbf{B} + \alpha) \to (\mathbf{B} + \alpha)$$

whis is much more informative.

The embedding property shows that Σ^μ has the important property:

Property 4.6.4 (*Universality of* Σ^μ)
Every term has a well typed Σ^μ completion. $\square$

This follows directly, since we know that every term has a **D**-completion.
Apart from the fact that more information can be extracted from the Σ^μ typing of a **D** completion, we have many more well-typed completions in Σ^μ than in **D**. In the presence of recursive types, we have of course the property [7]

Property 4.6.5 Every term of the pure λ-calculus has a Σ^μ-completion whose only non-trivial primitives are **f** and **u**. Hence, every such term is a Σ^∞ completion of itself. $\square$

The surplus set of completions is of course not limited to the pure terms; *many* more completions type in Σ^μ, which is much more expressive than **D**. The polymorphism of Σ^μ is of great potential benefit. To see an important example of this, consider the following problem in the monomorphic system **D**. Here, the type **D** has a *cascading* effect on completions; for instance, a functional object such as $\lambda x.M$ can only be tagged as such in **D** if it is already completed at type $\mathbf{D} \to \mathbf{D}$. This, in turn, may force an arbitrary number of coercions (both negative and positive ones) to be inserted into $\lambda x.M$ (depending on how M looks; it could be an entire

[7] The reader who is not familiar with recursive types may like to do the following (easy) exercise: Show, by induction on λ terms, that every λ term has the type $\mu\alpha.\alpha \to \alpha$.

program in itself.) This is a major source of inefficiency in system **D**; the problem is overcome by the polymorphism of the Σ-systems, where we can perfectly well have a completion of the form

$$[\texttt{F!}](\lambda x : \tau.M)$$

where the abstraction has no coercions inside it.

Example 4.6.6 Consider the term M

$$M \equiv \textbf{if}\ \mathit{true}\ \mathit{true}\ (\lambda x.\textbf{if}\ \mathit{true}\ (x\ \mathit{false})\ (x\ \mathit{false}))$$

It has the Σ-completion

$$\textbf{if}\ \mathit{true}\ ([\texttt{B!}]\mathit{true})\ ([\texttt{F!}](\lambda x.\textbf{if}\ \mathit{true}\ (x\ \mathit{false})\ (x\ \mathit{false})))$$

at the type $((\mathbf{B} \to \alpha) \to \alpha) + \mathbf{B}$.

On the other hand, completing M in **D** results in cascading of the tagging coercion `F!` at the abstraction, so that the corresponding **D**-completion of M is:

$$\textbf{if}\ \mathit{true}\ ([\texttt{B!}]\mathit{true})\ ([\texttt{F!}](\lambda x.\textbf{if}\ \mathit{true}\ ([\texttt{F?}]x\ [\texttt{B!}]\mathit{false})\ ([\texttt{F?}]x\ [\texttt{B!}]\mathit{false})))$$

at the type **D**. □

It is quite easy to scale up this effect arbitrarily in larger terms, and the example suggests that the reduction of both checks and tags can be quite drastic when completing in Σ rather than in **D**. We shall return to this theme *Part II* of this report, where we deal with the problems of completion inference in systems based on the calculus Σ^∞.

Chapter 5

Normalization

This chapter investigates the normalization problem for ϕ-reduction over λI-completions. We immediately note that the problem is only interesting for λI, since the reduction clearly fails to terminate if this restriction is dropped (use equation $[E4]$ to "invent" coercions ad infinitum.) Despite a very considerable effort, we have not succeeded in deciding the question. We believe the conjecture of termination, proposed in [Hen94], to be true, but we have been unable to provide a proof of this. However, we are able to prove that for a natural and practically very interesting, restricted class of completions, we have existence and uniqueness of normal forms under P_ϕ-reduction within that class. The class, called $\mathcal{C}_{p*}$ is characterized by the use of *primitive* coercions only, but with no restrictions on how they are placed in the term (apart from typability restrictions, of course.) The main result of this chapter states that, for any λI-completion $M \in \mathcal{C}_{p*}$, there is a unique P_ϕ-reduct M' of M with $M' \in \mathcal{C}_{p*}$, such that M' has no proper reducts in $\mathcal{C}_{p*}$. To reach M' it may be necessary (and is allowed by the theorem) to go anywhere outside the class $\mathcal{C}_{p*}$ in the intermediary steps of the reduction.

Unfortunately, our proof is not very constructive, and it is not trivial, so far as we can tell, to turn it into an algorithmic technique for completion. However, since the reduction P_ϕ posesses a very high degree of determinism, it is not to be excluded that we can construct more powerful completion algorithms by analyzing that reduction. The present result shows that such an enterprise is fundamentally meaningful, since we now know at least that normal forms (within the restricted class) always exist, and furthermore, these are unique. The practical interest of the class is stimulated in the final two chapters of this report, since we there give several examples to show that the class is very powerful and in fact necessary for some of the goals we wish to pursue in our extended framework of polymorphic dynamic typing.

The chapter is organised as follows. The first section defines a very important partial order on types which can be lifted to a preorder on completions. It is then shown, in the second section, that P_ϕ-reduction is monotonic wrt. the order. From this we can conclude that P_ϕ-reduction is *acyclic*. The third section then uses that property as key ingredience in a combinatorial argument which shows that reduction always terminates within restricted completion classes, among these $\mathcal{C}_{p*}$. The remaining parts of the section argues uniqueness of such normal forms. The fourth section considers general reduction in Σ, without rules for recursion unfolding. It turns out (easily, using previous results) that P_ϕ-reduction always terminates in a system where the structure of type-recursion is held fixed (*i.e.*, where unfolding is not allowed.) This shows that (perhaps not surprisingly) the hardness of the termination problem is concentrated in type recursion. The last, fifth, section of the chapter has, admittedly, more of a purely academic interest. We show two things. First, that strong normalization for the general termination problem is essentially reducible to weak normalization. Secondly, that there is a possibly interesting connection between termination of ϕ-reduction on a completion and termination properties of the underlying pure term wrt. β-reduction.

In this chapter we restrict our attention to λI-terms. This will be tacitly assumed in the remainder of the chapter. The reduction P_ϕ is taken to be the polarized ϕ-reduction of Figure 4.5 for Σ; we also assume this reduction transferred to $\mathbf{D}$ (in the obvious way, see Remark 4.4.13) and P_ϕ also denotes this relation in $\mathbf{D}$ [1]

5.1 Partial order on types

In this section we introduce, following [Hen94], an extremely useful partial order on the set of type expressions, with variants for both $\mathbf{D}$ and Σ. In the following section we shall see that this order can be lifted to a pre-order on completions. The main significance of this is that P_ϕ reduction will turn out to be strictly monotonic wrt. this order relation. From this a number of important consequences can be drawn in the following sections of the present chapter.

Following [Hen94] we define an order relation $\leq$ on types, as follows:

Definition 5.1.1 (Order on types, Henglein)

$$\tau \leq \tau' \text{ if and only if there exists a positive coercion } c : \tau \rightsquigarrow \tau'$$

Note that this definition makes sense for both $\mathbf{D}$ and Σ. Under the conditions of the definition we call c a *witness* of $\tau \leq \tau'$, and we may write

$$\tau \leq_c \tau'$$

to indicate that c is a witness of the relation. We write

$$\tau < \tau' \text{ if and only if } \tau \leq_c \tau' \text{ with } c \textit{ properly} \text{ positive, for some } c$$

It can be seen (induction on c) that $\tau <_c \tau'$ holds if and only if $\tau \leq_c \tau'$ and $\tau \not\equiv \tau'$. □

We define the notion of the *formal inverse* of a coercion c, called $(c)^{-1}$, as follows, where $(\bullet)^{-1}$ is a function from coercion terms to coercion terms:

$$\begin{array}{rcl}
(id)^{-1} & \equiv & id \\
(\uparrow)^{-1} & \equiv & \downarrow \\
(\downarrow)^{-1} & \equiv & \uparrow \\
(F^!)^{-1} & \equiv & F^? \\
(F^?)^{-1} & \equiv & F^! \\
(c \circ d)^{-1} & \equiv & (d)^{-1} \circ (c)^{-1} \\
(c \rightarrow d)^{-1} & \equiv & (c)^{-1} \rightarrow (d)^{-1} \\
(\sum_i c_i)^{-1} & \equiv & \sum_i (c_i)^{-1}
\end{array}$$

If $\diamond$ is a polarity, $+, -$, we let $\neg\diamond$ denote $-, +$, respectively. Then we have

Lemma 5.1.2 *For all Σ-types τ, τ' and coercions c:*

1. $c : \tau \rightsquigarrow \tau'$ *if and only if* $(c)^{-1} : \tau' \rightsquigarrow \tau$

[1] Recall in particular, that P_ϕ does *not* include the equations for neutral coercions in $\mathbf{D}$.

2. *$c : \diamond$ if and only if $(c)^{-1} : \neg\diamond$, and this holds with $\diamond$ proper on one side if and only if it is proper on the other side.*
3. *$\Sigma(\phi\psi) \vdash c \circ (c)^{-1} = (c)^{-1} \circ c = id$*

Proof All claims are verified by easy induction on c. □

Lemma 5.1.3 *The order $\leq$ is a partial order on the set of Σ-type expressions.*

Proof We must verify that $\leq$ is reflexive, transitive and anti-symmetric. Reflexivity follows from the existence of an identity coercion at every signature $\tau \rightsquigarrow \tau$. Transitivity follows from the fact that $c \circ d$ is positive, whenever both c and d are. Anti-symmetry is argued thus:

Suppose $\tau \leq_c \tau'$ and $\tau' \leq_d \tau$. Then $c : +$ and $d : +$, hence, by the previous lemma, $(d)^{-1} : -$ with $c, (d)^{-1} : \tau \rightsquigarrow \tau'$. Now, by inspection of the definition of polarity we see that any coercion which is negative (positive) must be a $\phi\psi$-normal form. Since c and $(d)^{-1}$ are both $\phi\psi$-normal forms at the same signature, $\tau \rightsquigarrow \tau'$, it follows that we must have $C \vdash c = (d)^{-1} = \kappa(\tau, \tau')$. Since the equations in C preserve the polarity of positive (negative) coercions, it follows that c must be both positive and negative; this is only possible if we have $C \vdash c = id$, and hence $\tau \equiv \tau'$. □

Following [Hen94] we can define an order $\leq_t$, giving a characterization of the structure of types:

Definition 5.1.4 Let $\leq_t$ be the least relation over Σ-types, closed under the following rules:

$$\begin{array}{lll}
\mathbf{E} \leq_t \tau & & \\
\tau \leq_t \tau & & \\
F_j\bar{\tau}_j \leq_t \sum_i F_i\bar{\tau}_i & & \\
\tau \leq_t \tau', \tau' \leq_t \tau'' & \Rightarrow & \tau \leq_t \tau'' \\
\tau_1 \leq_t \tau_1', \tau_1 \leq_t \tau_2' & \Rightarrow & \tau_1 \to \tau_2 \leq_t \tau_1' \to \tau_2' \\
\tau_i \leq_t \tau_i', i = 1 \ldots n & \Rightarrow & \sum_i \tau_i \leq_t \sum_i \tau_i'
\end{array}$$

The conditions are subject to the restriction that every type expression in the condition is well formed (in particular, all sums must be discriminative.) We write

$$\tau <_t \tau' \text{ if and only if } \tau \leq_t \tau' \text{ and } \tau \not\equiv \tau'$$

□

An analogous definition can be given for **D**-types, see [Hen94]. The two orders, $\leq_c$ and $\leq_t$, coincide: [2]

Lemma 5.1.5 *For any Σ-types τ, τ':*

1. *There exists a (properly) positive $c^+ : \tau \rightsquigarrow \tau'$ if and only if there exists a (properly) negative $c^- : \tau' \rightsquigarrow \tau$.*
2. *$\tau \leq_c \tau'$ if and only if $\tau \leq_t \tau'$.*
3. *$\tau <_c \tau'$ if and only if $\tau <_t \tau'$.*

[2]This holds also for the orders in **D**; again, see [Hen94]

PROOF The first claim is contained in Lemma 5.1.2.

For the second claim, the inclusion $\leq_c \subseteq \leq_t$ is proven by induction on c; the inclusion $\leq_t \subseteq \leq_c$ is proven by induction on the definition of $\leq_t$.

The third claim follows from the second: Suppose $\tau <_c \tau'$, then $\tau \leq_c \tau'$ and $\tau \not\equiv \tau'$, hence $\tau \leq_t \tau'$ and $\tau \not\equiv \tau'$. The other inclusion is proved analogously. □

In the sequel we shall use one notation $\leq$ and $<$ for the coincident orders $\leq_t$ and $\leq_c$ (and similarly for $<$)

5.2 Monotonicity of reduction

In this section we lift the partial order $\leq$ on types to a pre-order on completions. Using this order relation we are able to prove

- *Monotonicity:* P_ϕ-reduction is strictly monotonic wrt. the order on completions, and (therefore)
- *Acyclicity:* P_ϕ reduction is *acyclic*, in Σ and in $\mathbf{D}$.

In order to lift $\leq$ to an order on completions we need a few technical notions. For a pure term M, let $\mathcal{C}(M)$ denote the set of all completions (be it in Σ or in $\mathbf{D}$) of M, and let $P(M)$ denote the set of *coercion points* in M. For a completion M, we let $P(M)$ be the coercion points of the underlying term. For any completion we can assume w.l.o.g. that there is exactly one coercion at each coercion point. Hence, any coercion point in a completion M has the form

$$[c_\tau^{\tau'}]N$$

where N is a subterm occurrence, and where $[c_\tau^{\tau'}]N$ has no coercion applied to it in the occurrence within M, and N is not a coercion application. Abstracting from the coercion and focussing only on its signature, we can percieve this on the form

$$\langle \tau, \tau' \rangle N$$

where we call τ the *outer type* and τ' the *inner type* at the coercion point. Further, for a completion M we let $B(M)$ be the set of binding occurrences of λ-bound variables in M. Let $\Theta(M) = P(M) \cup B(M)$. Then, for every pure term M there is a function

$$\mathcal{T} : \Theta(M) \to \mathcal{C}(M) \to TyExp$$

where $TyExp$ denotes the set of type expressions of Σ or $\mathbf{D}$, as appropriate. The function $\mathcal{T}$ is defined, for any $M' \in \mathcal{C}(M)$, as follows:

- for $p \in P(M)$, $\mathcal{T}(p)(M')$ is the *outer type* at p within M', and
- for $b \in B(M)$, $\mathcal{T}(b)(M')$ is the type-annotation of the variable occurrence b within M'

These notions organize $\mathcal{C}(M)$ as an ordered set, for every pure term M, as follows:

Definition 5.2.1 (Pre-order on completions)
Let M be given pure term. Define the order $\sqsubseteq$ on $\mathcal{C}(M)$, by defining for $M', M'' \in \mathcal{C}(M)$

$$M' \sqsubseteq M'' \text{ if and only if } \forall \xi \in \Theta(M).\mathcal{T}(\xi)(M') \leq \mathcal{T}(\xi)(M'')$$

We write $M' \sqsubset M''$ if and only if

$$M' \sqsubseteq M'' \text{ and } \exists \xi \in \Theta(M).\mathcal{T}(\xi)(M') < \mathcal{T}(\xi)(M'')$$

Clearly, $\sqsubseteq$ is a pre-order on completions, since $\leq$ is a partial order on coercions. □

Now we have the following simple but fundamental characterization of polarized reduction (without the neutral equations)

Lemma 5.2.2 *(Monotonicity)*
If $M \Rightarrow_P M'$ then $M \sqsubset M'$.

Proof The proof is simple inspection of each of the rules in P, where one uses that $c : \tau \rightsquigarrow \tau'$ with c properly negative implies $\tau < \tau'$, and c properly positive implies $\tau' < \tau$. For example, consider rule $[P1^-]$,

$$[d^+ \to c^-]\,(\lambda x : \tau.M) \quad \Rightarrow_P \quad \lambda x : \tau'.[c^-]M\{x := [d^+]x\} \qquad [P1^-]$$

Let $\mathcal{L}$, $\mathcal{R}$ be the left- and right-hand-side terms, respectively, under a rewrite step by the rule above. Since d^+ is positive, we have $\tau' \leq \tau$; since c^- is negative, the outer type τ_o' of the body of the rewritten abstraction in $\mathcal{R}$, and the corresponding outer type τ_o in $\mathcal{L}$ must satisfy $\tau_o' \leq \tau_o$. Moreover, by the side-condition of the rule, either c^- is properly negative, or d^+ is properly positive, hence either $\tau' < \tau$ or $\tau_o' < \tau_o$, and all other outer types and type-annotations on bound variables in the term, in which this rewrite step occurs, are left invariant; from this it follows that $\mathcal{L} \sqsubset \mathcal{R}$. All the other rules of P can be analysed similarly. □

The lemma shows that the polarized reduction has a *direction*, since it *decreases* types. A similar argument can evidently be given for system **D**. Also, it can be straight-forwardly extended to the recursive systems, Σ^μ and Σ^∞. The definition of $\leq_c$ makes sense for both of these systems unchanged; the corresponding order $\leq_t$ can be extended to system Σ^μ by adding the recursion case to the definition of $\leq$:

$$\tau\{\alpha := \mu\alpha.\tau\} \leq \mu\alpha.\tau$$

For Σ^∞, where reduction is *modulo* the relation $\approx$, we extend the definition of $\leq_t$ by passing to the set of regular trees T_R (cf. Section 4.5.1.) We define $\zeta \leq_t \zeta'$ for $\zeta, \zeta' \in T_R$ by repeating the axioms of Definition 5.1.4, taking the metavariable τ to range over T_R. We then transfer this order back to type expressions by the definition

$$\tau \leq_t \tau' \text{ iff } (\tau)^* \leq_t (\tau)^*$$

where $(\bullet)^*$ is as defined in Section 4.5.1. Note that the order is invariant under $\approx$.

Under these extensions, the Monotonicity Lemma still goes through, with the same argument. An important consequence, then, is the following

Proposition 5.2.3 *(Acyclicity)*
*In **D**, Σ^μ and Σ^∞, P_ϕ-reduction is not cyclic (*modulo *C).*

Proof We note that ϕ-reduction on coercions is not cyclic (it is strongly normalizing) and that ϕ-reduction on coercions does not affect the order $\sqsubseteq$, since it preserves type signatures. Also, note that the equations in C do not affect the order $\sqsubseteq$. Finally, note that $\sqsubset, \sqsubseteq$ are transitive relations, since $<, \leq$ are. From these observations it follows that a proper cycle $M' \Rightarrow^+_{P_\phi} M'$ must involve at least one P-step, which in turn would imply $M' \sqsubset M'$, which is impossible. □

Note that the monotonicity property is destroyed when we add the neutral equations; indeed, we have seen that $P*$ is in fact cyclic (recall Example 2.2.5.)

5.3 Normalization in primitive classes

We investigate the problem of existence and uniqueness of normal forms under polarized reduction P_ϕ in restricted completion classes. The restricted classes are defined by requiring that we use only *primitive* coercions in completion. The class $\mathcal{C}_{p*}$ where this is the only restriction is probably of considerable practical interest. For the restricted classes we get optimal results, showing that there is always a unique normal form reduct in the class of any completion in the class. A normal form, relative to a class, is a term which has no proper reduct within the class.

5.3.1 Completion classes

In [Hen92] a classification of the set of **D**-completions was introduced, according to the form of coercions and their positions within the completions. We recall that the completion class C_{pf} consists of the completions M satisfying

- *restriction to primitives:* only primitive non-trivial coercions occur in M, and
- *restriction of positions:* negative coercions occur only at destruction points, and positive coercions occur only at construction points, within M

The class C_{p*} (see [Hen92]) consists of the completions with only primitive coercions, but at arbitrary places. So, we have $C_{pf} \subseteq C_{p*}$. In general, we have the classification:

	fixed positions	anywhere
primitives only	C_{pf}	C_{p*}
induced coercions	C_{*f}	C_{**}

This classification is extended to Σ, in obvious fashion, yielding the corresponding classes:

$\Sigma_{\diamond\diamond}$ in Σ

$\Sigma_{\diamond\diamond}^{\diamond\diamond}$ in Σ^μ

$\Sigma_{\diamond\diamond}^{\infty}$ in Σ^∞

with $\diamond$ instantiated to either p, f or $*$, as appropriate, the superscripted tags of Σ^μ-classes indicating restrictions on the *recursion coercions* $\mathbf{f}, \mathbf{u}$, and the subscripted ones indicating restrictions on the other coercions. Not all (in fact, probably only a few) of these classes have practical interest. In this report we shall be concerned with only two types of classes, namely the *unrestricted* ones (having $*$ everywhere) and the so-called *primitive classes.* We use the notation $\mathcal{C}$ with subscripted class indicators (p, $*$) generically, for classes of **D** or Σ^∞. We use C specifically for **D** and Σ specifically for Σ^∞.

Both in the case of **D** and the Σ-systems, we call a class *primitive*, if it imposes the restriction to primitive coercions, on every (non-trivial) coercion whatsoever. If $\mathcal{C}$ is a completion class and R a reduction on completions, we call a completion M in $\mathcal{C}$ a $\mathcal{C}(R)$-*normal form*, if no proper R-reduct of M is in $\mathcal{C}$:

Definition 5.3.1 Let $\mathcal{C}$ be any completion class, let R be a reduction on completions and suppose $M \in \mathcal{C}$. Then M is called a $\mathcal{C}(R)$-normal form if and only if there exists *no* $M' \in \mathcal{C}$ such that $M \Rightarrow_R^+ M'$. □

If the reduction R is understood, we may drop it from the notation (*e.g.*, speaking just of a $\mathcal{C}$-normal form.)

We shall be particularly concerned with the existence of $\mathcal{C}$-normal foms under polarized ϕ-reduction (P_ϕ). Note that it will generally be the case, more often than not, that a $\mathcal{C}$-normal form of a $\mathcal{C}$ -completion M is only reachable from M via reduction chains containing elements that are *not* in $\mathcal{C}$. In other words, to compute normal forms in these classes, one will usually have to go *outside* the classes in some of the intermediary reduction steps, because the classes (except for the unrestricted ones) are generally not closed under reduction. Just think of, say, P_ϕ-reduction in C_{p*}, as illustrated in the following example:

Example 5.3.2 The C_{p*}-completion

$$M \equiv (\lambda x : \mathbf{D}.([\mathtt{F?}]x)x)\ [\mathtt{F!}](\lambda x : \mathbf{D}.([\mathtt{F?}]x)x)$$

of Ω has a $C_{p*}(P_\phi)$-normal form, namely

$$M_{nf} \equiv (\lambda x : \mathbf{D} \to \mathbf{D}.x([\mathtt{F!}]x))\ (\lambda x : \mathbf{D}.([\mathtt{F?}]x)x)$$

but M_{nf} is clearly only obtainable from M by a series of P_ϕ-reduction steps involving intermediate terms that are *not* in C_{p*}. $\Box$

The completely general problem of normalizability, under P_ϕ-reduction and ϕ-reduction, for the unrestricted classes C_{**} and Σ^{**}_{**}, are just the problems of whether the relations P_ϕ and ϕ are (strongly or weakly) normalizing. As already indicated, a *very* considerable effort has been made to solve these problems, but unfortunately with no decisive result. We believe that the unrestricted normalizability problem is deep and that it could well be very difficult.

However, the primitive classes have considerable practical interest, for a number of reasons:

- Completions in primitive classes are relatively simple to understand
- We know of efficient algorithms for completion inference within the primitive class $\mathcal{C}_{pf}$ (cf. [Hen92] and Chapter 6 of this report) [3])
- Experience tells us that the primitive classes generally contain very good completions; in fact, a $\mathcal{C}_{p*}$-normal form will often be a global normal form.

For these restricted classes (and some others) we have been able to prove very satisfactory (in fact, optimal) results concerning the polarized reductions P_ϕ, as the following sections will show. To us, this indicates that the proof theoretic framework of [Hen94] reveals that there is a surprising formal order in the process of dynamic type completion, also when we consider those restrictions which are likely to be of great practical importance.

5.3.2 Existence of normal forms

We consider the problem of existence of normal forms under P_ϕ-reduction in the primitive completion classes. We ask: If $\mathcal{C}$ is a primitive class, is it always the case that a $\mathcal{C}$-completion has a $\mathcal{C}(P_\phi)$-normal form? We prove in this section that the answer is yes, since for every primitive class $\mathcal{C}$ we can show:

[3] In general, the only qualification to this is the fact that primitive operations in the language may need to be coerced with induced coercions; these can be eliminated, however, by η-expansions on the primitive operator.

- Every $\mathcal{C}$-completion has a $\mathcal{C}(P_\phi)$-normal form

The existence of such normal forms turns out to be deducible from the fact that P_ϕ-reduction is *acyclic* combined with observations that show that the primitive classes are *essentially finite* in the sense that, although the class may be infinite, only a finite subset of the the class matters for the termination behaviour of reduction.

We begin with the simple case of system **D**. The arguments given for this system will then be transferred to Σ^μ and Σ^∞. We begin by stating some definitions, concerning the classes ccc_{pf} and ccc_{p*}. Let R be a reduction defined on coercions (we shall think here just of various forms of ϕ-reduction, in **D** and Σ, as appropriate.) A *local* $ccc_{p*}(R)$-normal form is a completion $M \in ccc_{p*}$ such that every coercion occurring in M is a normal form wrt. R-reduction on coercions. Note that a $ccc_{p*}(P_\phi)$-normal form is in particular a local one. Note also that every $\mathcal{C}_{p*}$-completion has a unique *local* $\mathcal{C}_{p*}$-normal form, which is again in $\mathcal{C}_{p*}$.

A *simple* coercion is one which contains no non-trivial induced coercions (*i.e.*, apart from trivial coercions, which can be arbitrary, it contains only compositions of primitives.) In both classes, ccc_{pf} and ccc_{p*}, only simple coercions are allowed.

We now attack the normalization question for C_{pf}. As we shall see, for this class the matter is particularly simple. The main observation needed is this: A C_{pf}-completion has only finitely many coercion points. This follows from the C_{pf}-condition concerning construction- and destruction points. In particular, it is impossible to have a local ψ-redex (on coercions) in a legal C_{pf}-completion, since for such a redex to be present, a negative coercion would have to sit at a non-destructive position (namely immediately under a tagging coercion.) Hence, only simple ϕ-redices of the form $\pi^- \circ \pi^+$ (with π ranging over primes) can be assembled at any given point. Since there are only finitely many primitive coercions, it follows by simple combinatorics that there is only a finite number of ways we can construct a C_{pf}-completion from any given term, which has only a finite number of coercion points.

Proposition 5.3.3 *In* **D***, every C_{pf}-completion has a $C_{pf}(P_\phi)$-normal form.*

Proof There are only finitely many distinct (*modulo C*) completions in C_{pf} of a given term. Therefore, if a C_{pf}-completion M had no C_{pf}-normal form, there would have to be a P_ϕ-reduction sequence starting from M containing $M' \Rightarrow^+_{P_\phi} M''$ with $C \vdash M = M'$ and $M', M'' \in C_{pf}$. But this entails that P_ϕ is cyclic, contradicting Proposition 5.2.3. □

In the case of C_{p*} the matter is slightly more complicated, but it turns out that essentially the same argument still applies. The complication is that C_{p*} may well contain infinitely many distinct (*modulo C*) completions of the same pure term, since we can have arbitrarily complex compositions of primitives in C_{p*}-completions. However, we observe that only the subset of *local* ϕ-normal forms are really interesting; and this subset must always be finite. These are the main points in the following proof for C_{p*}.

First, we establish that the set of local ϕ-normal forms are finite:

Lemma 5.3.4 *1. In* **D***, every simple coercion in ϕ-normal form has at most* 2 *non-trivial factors.*

2. The set of all local *$C_{p*}(\phi)$-normal form completions of a given term is finite* modulo C.

Proof The first part follows from inspection of cases: If we ever try to build a composite *simple* coercion more complex than as stated in the lemma, we shall be forced to introduce a ϕ-redex:

(1) If we begin with a negative, say F?, we have only two ways of continuing, namely as

(1.1) $\ldots$F? $\circ$ F! $\circ$ F? (our attempted construction grows to the left, as indicated by the dots) which contains a ϕ-redex, or

(1.2) we can continue with a negative error-coercions, as in $\downarrow \circ$ F?, but then any continuation to the left of the error-coercion must have a positive right-factor, which will then constitute a ϕ-redex by the error-rules.

(2) If we begin with a positive we have three possibilities for proceeding:

(2.1) we can create an ordinary ϕ-redex, as in $\ldots$F? $\circ$ F!, or else

(2.2) we are forced to create a ϕ-redex for the error-rules, as in $\ldots$B? $\circ$ F! , or

(2.3) we begin with a positive error-coercion and continue with a positive coercion, as in F! $\circ \uparrow$. If we try to add another coercion factor to the left of this coercion, we shall be forced to enter one of the two previous situations, introducing a ϕ-redex.

The second part follows from the first by simple combinatorics: Every term has at most finitely many coercion points, and (by the assumption together with the first part) at every such point there can be at most 2 primitive factors. Since there are only finitely many primitives, the claim must be true. $\square$

The main point in the following proof is that, although the set of C_{p*}-completions of a given term may be infinite (*modulo* C), only the finite subset of *local* ϕ-normal forms really matters.

Proposition 5.3.5 *In* $\mathbf{D}$*, every* C_{p*} *completion* P_ϕ*-reduces to a* $C_{p*}(P_\phi)$*-normal form.*

PROOF By contradiction. So suppose there were a C_{p*}-completion M_0 with no normal form. Clearly, M_0 has a *local* C_{p*}-normal form, call it M_{loc_0}, obtainable by ϕ-normalizing at every coercion point in M_0. Since M has no C_{p*}-normal form, neither has M_{loc_0}. Therefore, there must exist a C_{p*}-completion M_1 such that $M_{loc_0} \Rightarrow^{+}_{P_\phi} M_1$. Now, M_1 cannot have a C_{p*}-normal form, since this would imply that M_{loc_0} and hence also M_0 would have one, because we have

$$M_0 \Rightarrow^{*}_{P_\phi} M_{loc_0} \Rightarrow^{+}_{P_\phi} M_1$$

Clearly, this process can be continued indefinitely, leading to an infinite reduction sequence of the form

$$\begin{tikzcd}
M_0 \arrow[r, "+"] & M_1 \arrow[d, "*"] & & & \\
& M_{loc_0} \arrow[r, "+"] & M_2 \arrow[d, "*"] & & \\
& & M_{loc_1} \arrow[r, "+"] & M_3 \arrow[d, "*"] & \\
& & & M_{loc_2} \arrow[r, "+"] & \cdots
\end{tikzcd}$$

This determines an infinite "diagonal" sequence M_{loc_n} of *local* ϕ-normal forms such that $M_{loc_i} \Rightarrow^{+}_{P_\phi} M_{loc_{i+1}}$, for all i. Since there are only finitely many *local* $C_{p*}(\phi)$-normal forms, it follows that for some j, k with $k > j$ we have $C \vdash M_{loc_j} = M_{loc_k}$. Therefore there is a cycle in the P_ϕ-reduction sequence starting from M_0, which is impossible. $\square$

We now ask whether the methods just employed can be used to prove similar results for system Σ^μ. This turns out to be the case, but we will need to add some further twists to the argument. First, let us agree to orient the two equations $[C10]$ and $[C11]$, left-to-right, and let us denote by $\rightarrow$ the relation $[C10-11]^{\rightarrow}$ taken *modulo* the remaining equations in C. Then it is not difficult to see that

- $\rightarrow^*$ commutes with P and ϕ (on coercions), by inspection of polarities
- $\rightarrow$ is canonical

from which it follows that we can erect diagrams

$$\begin{tikzcd}
M \arrow[rr, equal, "C"] \arrow[dr, dash, "*"] & & M' \arrow[r, "P\cup\phi"] \arrow[dl, dash, "*"'] & M'' \arrow[dl, dash, "*"'] \\
& N \arrow[r, dashed, dash, "P\cup\phi"'] & N' &
\end{tikzcd}$$

From this we can infer that it is enough to establish termination (within the primitive classes) of the reduction [4] $P\phi/\rightarrow^*$.

We now consider the primitive completion classes $\mathcal{C}$, either Σ^{pf}_{pf} or Σ^{p*}_{p*}. Now, it is clear that the arguments employed above to prove termination in the primitive classes of $\mathbf{D}$ can be used to prove that, for any term M, there are only finitely many local ϕ-normal forms (*modulo* C) in $\mathcal{C}$, *provided that we do not distinguish between different instantiations of the same primitive.* The proviso just mentioned is necessary for the Σ-system, because a primitive, such as, *e.g.*, $\mathtt{F!}$, is now effectively a polymorphic constant with an infinity of different type instatiations. Therefore, there can be infinitely many *different* $\mathcal{C}$-completions (in local normal form) whose type-erasures are all identical; consider just the infinite series

$$[\mathtt{F!}^{\mathbf{B}+(\tau\rightarrow\tau)}_{\tau\rightarrow\tau}](\lambda x:\tau.x)$$

of different completions of the identity indexed by τ. The fact which rescues the argument here is the following:

- No reduction rule can change the instatiation of a non-trivial primitive. Moreover, no rule of P_ϕ, except for $[C10-11]$, performed on a λI-completion, can invent a new coercion (only, they can copy existing ones.) Therefore, for any λI-completion M, it is the case that every reduct (under $P\phi/\rightarrow^*$) of M contains only copies of coercion *instances* that are already in M. Hence, for any *instantiation* of a primitive p^θ_τ to be present in a reduct of M, at least one occurrence of that *same instantiation* of that primitive must already be present in M.

From this it follows that the set $\mathcal{R}(M)$ of $\mathcal{C}$-reducts of $M \in \mathcal{C}$,

$$\mathcal{R}(M) = \{M' \mid M \Rightarrow^* M'\} \cap \mathcal{C}$$

can be generated combinatorially by insertion of copies of only finitely many *instances* of non-trivial primitives. Using arguments similar to the ones employed earlier, we can then argue that the set of local ϕ-normal forms in $\mathcal{R}(M)$ must be finite. Using acyclicity of P_ϕ in Σ^μ we can therefore repeat the proofs given earlier, and we record our main result

Theorem 5.3.6 *In* $\mathbf{D}$ *and* Σ^∞*, every* $\mathcal{C}_{p*}$*-completion* P_ϕ*-reduces to a* $\mathcal{C}_{p*}$*-normal form.*

[4]The notation R/S, read "R over S", denotes $S^* \circ R \circ S^*$. See [BD86].

A similar statement can be made for Σ^μ.

Note that, unfortunately, a $ccc_{p*}(P_\phi)$-normal form will *not* necessarily also be a $ccc_{p*}(\phi)$-normal form. We could repeat the proof for ϕ-reduction if we could prove that ϕ is not cyclic. However, we have not proven this. We believe it holds, but we have not investigated this question in depth. It is likely to be the case that only the polaerized reductions will be of real practical use in algorithm construction, due to their high degree of determinism.

The proofs above are not very constructive. One could probably constructivize them, in theory, by noting that we could compute an upper bound on the number of local ϕ-normal form completions of a given term. This could be made the basis of a terminating search through the reduction tree of a term, since reduction is finitely branching *modulo* the equations. This is not a pragmatically very interesting idea, of course.

Experience leads us to believe that the classes $\mathcal{C}_{p*}$ is *very* powerful, indeed. Very often, a $\mathcal{C}_{p*}$-normal form will in fact be an unrestricted ϕ-normal form.

5.3.3 Uniqueness of normal forms

We have seen that, for every primitive class $\mathcal{C}$, every $\mathcal{C}$-completion has at least one $\mathcal{C}(P_\phi)$-normal form. The next major question now is, for $\mathcal{C}$ primitive:

- Are the $\mathcal{C}$-normal forms of a given completion *unique* (up to C-equivalence [5]) ?

We know, of course, that P_ϕ-reduction as such is confluent. However, since the classes $\mathcal{C}$ are not closed under reduction, we can by no means conclude anything concerning uniqueness of $\mathcal{C}$-normal forms. To spell this out completely, we shall always have confluence diagrams

$$\begin{tikzcd}
 & M \arrow[dl, "*"'] \arrow[dr, "*"] & \\
M_1 \in \mathcal{C} \arrow[dr, dashed, "*"] & & M_2 \in \mathcal{C} \arrow[dl, dashed, "*"'] \\
 & M' &
\end{tikzcd}$$

but due to the failure of $\mathcal{C}$ being closed under reduction, we cannot be certain right away that we shall always have $M' \in \mathcal{C}$. However, it turns out that, due to the almost orthogonal character of the P_ϕ-reduction rules, this *is* in fact the case: Of two divergent reducts in $\mathcal{C}$ of a common (arbitrary) term, it is always possible to find a common reduct which is again in $\mathcal{C}$. From this we can conclude that normal form not only always exist, but they are also unique. This shows that these completion classes exhibit a very considerable order wrt. P_ϕ-reduction. Our proof is a variation on themes that have already been played, in Chapter 2, only now we go into a very close analysis of confluence, keeping track of individual redices. We shall, however, draw heavily upon the insights gained in Chapter 2, which should be fresh in the mind of the reader of this section.

Definition 5.3.7 (Equivalent P-redices) Let ΔR and ΔS denote P-redex occurrences. We write

$$\Delta R \cong \Delta S$$

[5]Nothing stronger can be expected, since the relation P_ϕ is *modulo* C-equivalence; since this is not a very complicated equivalence, uniqueness up to C will be a very strong property.

if and only if ΔR and ΔS differ *only* wrt. coercion factorization under ϕ-reduction on coercions. Further, we write

$$M \overset{\Delta R}{\rightarrow} M'$$

if and only if M' arises from M be reduction of P-redex occurrence ΔR in M. □

Note that, since ϕ-reduction is *modulo C*, redices which differ only by different coercion factorizations under C-equality are equivalent. Clearly, $\cong$ is an equivalence relation. The intuitive meaning of the equivalence of redices is that, for $\Delta R \cong \Delta S$, reduction of any one of the two redices will *perform essentially the same construction or destruction of induced coercions.* Thus, suppose a reduction of ΔR, say, constructs an induced coercion, say $c \rightarrow d$, at some point p_0 in the term, by displacing coercions c and d from some points p_1, p_2 respectively, in the term; then, for $\Delta S \cong \Delta R$, a reduction of ΔS will construct a corresponding induced coercion, say $c' \rightarrow d'$, at the *same* point p_0 and by displacing c', d' from the *same* coercion points p_1, p_2, respectively; and moreover, the pairs c, c' and d, d' emerge by different ϕ- (including possibly just C-) factorizations of the same coercions. A concrete example:

Example 5.3.8 Let

$M \equiv \lambda x : \tau.[d^+]M$

$\Delta R \equiv \lambda x : \tau.[c_1^+]M'$

$\Delta S \equiv \lambda x : \tau.[c_2^+]M''$

where $C \vdash [d^+]M = [c_1^+]M'$ and also $C \vdash [d^+]M = [c_2^+]M''$. In this case we have $\Delta R \cong \Delta S$, two equivalent redices under the rule $[P1^+]$, leading to the reductions (respectively)

$\Delta R \Rightarrow_P [c_1^+ \rightarrow id](\lambda x : \tau.M')$

$\Delta S \Rightarrow_P [c_2^+ \rightarrow id](\lambda x : \tau.M'')$

□

Note that the critical pair defined in the example will be solvable by reduction of redices that are again equivalent (by the Factorization Theorem, 2.1.13) The following lemma uses these ideas. The reader should have the rewrite analysis in the proof of Proposition 2.2.2 fresh in mind:

Lemma 5.3.9 *For divergent P-reductions by redices ΔR, ΔS one has either*

$$\begin{tikzcd}
 & M \arrow[dl, "\Delta R"'] \arrow[dr, "\Delta S"] & \\
M_1 \arrow[d, dash, dashed, "\phi^*"'] & & M_2 \arrow[d, dash, dashed, "\phi^*"] \\
M_3 \arrow[dr, dash, dashed, "\Delta S'"'] & & M_4 \arrow[dl, dash, dashed, "\Delta R'"] \\
 & M_5 &
\end{tikzcd}$$

with $\Delta R \cong \Delta R'$, $\Delta S \cong \Delta S'$, or else one has

$$\begin{tikzcd}
 & M \arrow[dl, "\Delta R"'] \arrow[dr, "\Delta S"] & \\
M_1 \arrow[dr, dashed, "\phi^*"'] & & M_2 \arrow[dl, dashed, "\phi^*"] \\
 & M_3 &
\end{tikzcd}$$

where the ϕ-reductions eliminate the coercions involved in the redices ΔR and ΔS.

PROOF Recall from the analysis in the proof of Proposition 2.2.2 (*case III*) that divergent reductions obtained by overlapping P-rules fall in two sub-cases:

Case 1 where the same P-rule is applied at the same point, only with different negative right, respectively, positive left factors, and

Case 2 where two different P-rules are applied at the same point.

The first case is solved as stated after Example 5.3.8, by reducing redices $\Delta S'$, $\Delta R'$ with $\Delta S \cong \Delta S'$ and $\Delta R \cong \Delta R'$, using just empty ϕ-reductions.

The second case can again take two forms. There is the simple case where ϕ-reduction alone leads to a common reduct by eliminating the rewritten coercions completely, as in the following divergent situation:

$$M \equiv [\mathtt{F!} \to id](\lambda x : \tau . M\{x := [\mathtt{F?}]x\})$$

$$M_1 \equiv \lambda x : \tau' . M\{x := [\mathtt{F?} \circ \mathtt{F!}]x\}$$

$$M_2 \equiv [(\mathtt{F!} \to id) \circ (\mathtt{F?} \to id)](\lambda x : \tau' . M)$$

with $M \Rightarrow_P M_1$ and $M \Rightarrow_P M_2$. Here the critical pair is solved by reducing

$$M_1 \to_\phi \lambda x : \tau' . M$$

$$M_2 \to_\phi \lambda x : \tau' . M$$

In the more complicated case where ϕ-reduction alone is not sufficient, we obtain the common reduct by first doing a $+\circ-$ factoring, under ϕ-reduction (cf. Proposition 4.4.11), of the coercions involved in the critical pair, and then we can choose redices to arrive at the common reduct; this analysis is performed in the proof of Proposition 2.2.2 (*case III*) (2) (with the additional aspect of neutral coercions, which is absent here). It can be seen from the analysis made there that *equivalent* redices can always be chosen. $\Box$

The proof of the previous lemma (including the references to earlier proofs made there) gives a *standard* way of choosing redices in order to obtain a convergent reduction. We shall always assume that, when postulating the possibility of constructing such reductions by appealing to this lemma, the construction is made in the standard way; in particular, it is assumed that no reductions apart from those that are strictly necessary for convergence are performed in the constructed reductions. We shall not be explicit about this assumption in the sequel, since we believe the idea is clear enough, and explicitating everything would necessitate an intolerable degree of formalism.

Lemma 5.3.10 *ΔR-reduction commutes with ϕ^*:*

$$\begin{tikzcd}
 & M \arrow[dl, "\Delta R"'] \arrow[dr, "\phi^*"] & \\
M_1 \arrow[dr, dashed, "\phi^*"'] & & M_2 \arrow[dl, dashed, "\Delta R"] \\
 & M_3 &
\end{tikzcd}$$

Proof The essential analysis is contained in the proof of Proposition 2.2.2: There are no overlaps between ϕ-rules and P-rules, by inspection of polarities. □

Now, write

$$M \overset{\Delta^\phi R}{\rightarrow} M'$$

if and only if either

$$M \rightarrow_\phi^* M'' \overset{\Delta R}{\rightarrow} M''' \rightarrow_\phi^* M'$$

where ΔR is a P-redex occurrence, or else

$$M \rightarrow_\phi^* M'$$

Note in particular that a step of the form $M \overset{\Delta^\phi R}{\rightarrow} M'$ may be empty (so $M = M'$) We write

$$M \overset{\Delta^\phi R_i}{\rightarrow^*} M'$$

if and only if

$$M \overset{\Delta^\phi R_1}{\rightarrow} M_1 \overset{\Delta^\phi R_2}{\rightarrow}, \ldots, \overset{\Delta^\phi R_n}{\rightarrow} M'$$

for some n. Then we have

Lemma 5.3.11

$$\begin{tikzcd}
 & M \arrow[dl, "\Delta^\phi R"'] \arrow[dr, "\Delta^\phi S"] & \\
M_1 \arrow[dr, dashed, "\Delta^\phi S'"'] & & M_2 \arrow[dl, dashed, "\Delta^\phi R'"] \\
 & M_3 &
\end{tikzcd}$$

where it holds that

1. *if the reduction $\Delta^\phi R'$ contains a non-empty P-step, then so does the reduction $\Delta^\phi R$, and moreover $\Delta R \cong \Delta R'$; and if the reduction $\Delta^\phi R'$ contains no non-empty P-step, then the ϕ^*-reductions eliminate the coercions involved in the ΔR-redex*
2. *if the reduction $\Delta^\phi S'$ contains a non-empty P-step, then so does the reduction $\Delta^\phi S$, and moreover $\Delta S \cong \Delta S'$; and if the reduction $\Delta^\phi S'$ contains no non-empty P-step, then the ϕ^*-reductions eliminate the coercions involved in the ΔS-redex*

Proof By analysis of cases over $\Delta^{\phi}R$ and $\Delta^{\phi}S$, using Lemma 5.3.9, Lemma 5.3.10 and confluence of ϕ-reduction on coercions:

If both of the steps $\Delta^{\phi}R$ and $\Delta^{\phi}S$ are empty or just ϕ^* steps on coercions, the claim is true by confluence of ϕ-reduction on coercions. If either one of the steps is just a ϕ^*-step, the claim follows easily from Lemma 5.3.10.

If both $\Delta^{\phi}R$ and $\Delta^{\phi}S$ contain a non-empty P-step, then the claim follows from Lemma 5.3.9: In the first situation of that lemma, we can erect the diagram

$$\begin{tikzcd}
 & & & M \arrow[dl, "\phi^*"'] \arrow[dr, "\phi^*"] & & & \\
 & & M_1 \arrow[dl, "\Delta R"'] \arrow[dr, dashed, no head, "\phi^*"] & & M_2 \arrow[dl, dashed, no head, "\phi^*"'] \arrow[dr, "\Delta S"] & & \\
 & M_3 \arrow[dl, "\phi^*"'] \arrow[dr, dashed, no head, "\phi^*"] & & P_1 \arrow[dl, dashed, no head, "\Delta R"'] \arrow[dr, dashed, no head, "\Delta S"] & & M_4 \arrow[dl, dashed, no head, "\phi^*"'] \arrow[dr, "\phi^*"] & \\
M_5 \arrow[ddr, dashed, no head, "\phi^*"] & & P_2 \arrow[d, dashed, no head, "\phi^*"] & & P_3 \arrow[d, dashed, no head, "\phi^*"] & & M_6 \arrow[ddl, dashed, no head, "\phi^*"'] \\
 & & P_4 \arrow[dl, dashed, no head, "\phi^*"'] \arrow[dr, dashed, no head, "\Delta S'"] & & P_5 \arrow[dl, dashed, no head, "\Delta R'"'] \arrow[dr, dashed, no head, "\phi^*"] & & \\
 & P_6 \arrow[dr, dashed, no head, "\Delta S'"] & & P_8 \arrow[dl, dashed, no head, "\phi^*"'] \arrow[dr, dashed, no head, "\phi^*"] & & P_7 \arrow[dl, dashed, no head, "\Delta R'"'] & \\
 & & P_9 \arrow[dr, dashed, no head, "\phi^*"] & & P_{10} \arrow[dl, dashed, no head, "\phi^*"'] & & \\
 & & & M' & & &
\end{tikzcd}$$

with $\Delta R \cong \Delta R'$ and $\Delta S \cong \Delta S'$, using Lemma 5.3.10 and confluence of ϕ-reduction on coercions in addition to Lemma 5.3.9. In case the second possibility of Lemma 5.3.9 obtains, the lower part of the diagram above can be filled out with ϕ^*-reductions instead of the P-reduction steps, and the claim is still satisfied. □

On the basis of the previous lemma, we can prove extended simulation lemmas which show that simulating (lower dotted) reductions can be found for diverging (upper) reduction sequences of arbitrary length. These lemmas are given in Appendix B.3. They lead to the proposition

Proposition 5.3.12 *Let $\mathcal{C}$ be any one of the primitive completion classes. Suppose $M \Rightarrow^*_{P_\phi} M_1$ and $M \Rightarrow^*_{P_\phi} M_2$ with $M_1, M_2 \in \mathcal{C}$. Then there exists $M' \in \mathcal{C}$ such that $M_2 \Rightarrow^*_{P_\phi} M'$ and $M_2 \Rightarrow^*_{P_\phi} M'$:*

$$\begin{tikzcd}
 & M \arrow[dl, "(P_\phi)^*"'] \arrow[dr, "(P_\phi)^*"] & \\
M_1 \in \mathcal{C} \arrow[dr, dashed, "(P_\phi)^*"'] & & M_2 \in \mathcal{C} \arrow[dl, dashed, "(P_\phi)^*"] \\
 & M' \in \mathcal{C} &
\end{tikzcd}$$

PROOF Given in Appendix B.3, Proposition B.3.3. □

Combining this property with the results of Section 5.3.2 we get our main theorem on normalization:

Theorem 5.3.13 *(Existence and uniqueness of primitive normal forms) Let $\mathcal{C}$ be any of the primitive completion classes. Then every completion in $\mathcal{C}$ has a $\mathcal{C}(P_\phi)$-normal form which is unique up to C-equivalence.*

PROOF By the results of Section 5.3.2 we know that every completion in $\mathcal{C}$ has a normal form. Now, suppose, for some $M \in \mathcal{C}$, that both $M_1 \in \mathcal{C}$ and $M_2 \in \mathcal{C}$ are $\mathcal{C}(P_\phi)$-normal forms of M; then, by Proposition 5.3.12, there exists an $M' \in \mathcal{C}$ such that we have the commuting diagram

$$\begin{tikzcd}
 & M \arrow[dl, "(P_\phi)^*"'] \arrow[dr, "(P_\phi)^*"] & \\
M_1 \in \mathcal{C} \arrow[dr, dashed, "(P_\phi)^*"'] & & M_2 \in \mathcal{C} \arrow[dl, dashed, "(P_\phi)^*"] \\
 & M' \in \mathcal{C} &
\end{tikzcd}$$

But then $M_1 = M_2 = M'$ follows from the assumption that M_1, M_2 are normal forms. □

We round off this section with a continuation of Example 5.3.2. First, consider again the term M shown in that example; it is its own $\mathcal{C}_{pf}$-normal form, but it is not a $\mathcal{C}_{p*}$-normal form. The term M_{nf} in the example is clearly the unique $\mathcal{C}_{p*}(P_\phi)$-normal form of M. And it is in fact *much* more than that; it is even the unique ϕ-normal form of M, in the unrestricted class $\mathcal{C}_{**}$; and moreover, if we admit the recursion coercions of Σ^μ, we can see that the corresponding Σ^μ-completion

$$M^\mu \equiv (\lambda x : \tau.([\mathtt{F}? \circ \mathbf{u}]x)x)\ [\mathbf{f} \circ \mathtt{F}!](\lambda x : \tau.([\mathtt{F}? \circ \mathbf{u}]x)x)$$

reduces to the corresponding unique $\Sigma^{p*}_{p*}(P_\phi)$-normal form

$$M^\mu_{nf} \equiv (\lambda x : \tau'.x([\mathbf{f} \circ \mathtt{F}!]x))(\lambda x : \tau.([\mathtt{F}? \circ \mathbf{u}]x)\ x)$$

which is in fact also the unique $\Sigma^{**}_{**}(\phi)$-normal form of M^μ. This illustrates the idea (suggested by experience) that, for quite many terms, the $\mathcal{C}_{p*}$-normal form under P_ϕ will indeed be very good.

5.4 Normalization in Σ

In this section we prove that P_ϕ-reduction is strongly normalizing in system Σ. In conjunction with our confluence results of Chapter 4 and previous properties established in the present chapter this implies that

- Every Σ-completion has a P_ϕ-normal form which is unique up to $\mathcal{C}$.

Thus, we can always minimize completions in Σ_{**} uniquely, under P_ϕ-reduction. Another consequence of the normalization proof for Σ is that, in the recursive systems Σ^μ and Σ^∞, every completion has a unique P_ϕ-normal form *relative to any fixed structure of type recursion*, as explained below.

To prove the strong normalization result we define a measure ∂ on Σ-types:

$$\begin{array}{rcl}
\partial(\mathbf{E}) & = & 0 \\
\partial(A) & = & 1 \text{ , for } A \text{ atomic, } A \not\equiv \mathbf{E} \\
\partial(\tau \rightarrow \tau') & = & 1 + \partial(\tau) + \partial(\tau') \\
\partial(\sum_i \tau_i) & = & 1 + \sum_i \partial(\tau_i)
\end{array}$$

Then we have

Lemma 5.4.1 *If $\tau \leq \tau'$ then $\partial(\tau) \leq \partial(\tau')$, and if $\tau < \tau'$ then $\partial(\tau) < \partial(\tau')$.*

Proof By induction on the definition of $\leq_t$. □

Therefore, $<$ is well-founded on the set of Σ-types:

Proposition 5.4.2 *The set of Σ-types ordered by $<$ has no infinite descending chains.*

Proof There is only one type τ with $\partial(\tau) = 0$, namely $\tau \equiv \mathbf{E}$, and $\mathbf{E}$ has no type strictly below itself. For any type $\tau \not\equiv \mathbf{E}$, we have $\partial(\tau) > 0$. Therefore, the claim follows from the previous lemma. □

The proposition just stated marks *the crucial difference* between $\mathbf{D}$, Σ^μ, Σ^∞ on the one hand and Σ on the other (wrt. the problem of normalization), since the corresponding property *fails* for $\mathbf{D}$ and Σ^μ:

Example 5.4.3 In $\mathbf{D}$, it is easy to build infinite descending chains, *e.g.*

$$\mathbf{D} > \mathbf{D} \rightarrow \mathbf{D} > (\mathbf{D} \rightarrow \mathbf{D}) \rightarrow (\mathbf{D} \rightarrow \mathbf{D}) > \ldots$$

This can also be regarded as an infinite chain in Σ^μ, by the embedding of $\mathbf{D}$ into Σ^μ, and hence also a chain in Σ^∞. □

At this point it is a short step to

Theorem 5.4.4 *In Σ, P_ϕ-reduction is strongly normalizing.*

Proof We establish strong normalization for P_ϕ by contradiction. So, suppose there were an infinite P_ϕ-reduction path starting from some completion M in Σ. Since ϕ-reduction on coercions is strongly normalizing, this path must contain P-steps infinitely often. Each such step decreases the order $\sqsubset$, by Monotonicity (Lemma 5.2.2) and the ϕ-steps on coercions leaves this order invariant. Therefore, there must be an infinite decreasing chain

$$M \sqsupset M_1 \sqsupset \ldots$$

as a subsequence of the infinite reduction sequence. Since there are only finitely many coercion points and bound variables in all the terms of the sequence, this implies that there must exist an infinite decreasing chain of types

$$\tau > \tau_1 > \ldots$$

But this contradicts Proposition 5.4.2, and P_ϕ must be strongly normalizing. □

Combining Theorem 5.4.4 with Theorem 4.4.12 we have:

Corollary 5.4.5 *In Σ, P_ϕ-reduction is canonical, and in particular, every Σ-completion has a unique (up to C) P_ϕ-normal form.*

A comparison of Theorem 5.4.4 with Example 5.4.3 shows that the essence of the unrestricted termination problem for $\mathbf{D}$, Σ^μ and Σ^∞ lies just in the recursive type operations, and not in the operations on sums (in $\mathbf{D}$ one cannot distinguish the two kinds of operations, but the Σ-systems provide an analysis of $\mathbf{D}$ where the difference becomes clear.) The proof of the theorem also shows that, if we block the type recursion operations in the recursive systems by holding any given type recursion pattern fixed, [6] we get strong normalization. We therefore have

Corollary 5.4.6 *In Σ^μ and Σ^∞, every completion has a unique P_ϕ-normal form, relative to any fixed structure of type recursion.*

5.5 Normalization in D and in Σ^μ

This section is concerned with the general termination problem of P_ϕ and ϕ. The first subsection reduces (essentially, see the qualifications below) the problem of SN (strong normalization) to that of WN (weak normalization.) The second subsection is a somewhat technical development of a connection between ϕ-reduction on a completion and β-reduction in the underlying program. We show that ϕ terminates on every completion of a pure term with a β-normal form. Nothing else in this report requires reading of this section, and the reader may wish to skip (details of) the second subsection. We feel that the development given here adds to our confidence that the termination conjecture of [Hen94] is true.

5.5.1 Reduction of SN to WN

We show that, for the unrestricted P_ϕ-reduction of $\mathbf{D}$ and Σ^μ, weak normalization implies strong normalization (over λI, as always) for so-called *error-free* terms; these are completions with no P_ϕ-reduct containing the error-type $\mathbf{E}$ or any error-coercions. In other words, for the error-free terms, we reduce the problem of strong normalization to that of weak normalization. This implies in particular, that the recursion reduction μ is strongly normalizing if it is weakly normalizing.

The main ingredience in the reduction proof is a combination of *monotonicity* (Lemma 5.2.2) and *confluence* (Theorem 4.4.12 together with Remark 4.4.13)

Theorem 5.5.1 *(Reduction) In $\mathbf{D}$ and Σ^μ: If an error-free completion has a P_ϕ-normal form, then it is strongly normalizing. In particular, μ-reduction is strongly normalizing if it is weakly normalizing.*

PROOF We consider system Σ^μ. Note that, for any type τ not containing $\mathbf{E}$ there is only a finite number of Σ^μ-types above τ wrt. the partial order $\leq$ on types; *i.e.*, for such τ, there are no infinite *a*scending chains starting from τ.

The proof is by contradiction. So assume P_ϕ weakly normalizing, but not strongly normalizing, aiming for contradiction. Then there exists an error-free λI-completion M' of pure term M with an infinite P_ϕ-reduction path starting from it:

$$M' \Rightarrow_{P_\phi} M_1 \Rightarrow_{P_\phi} M_2 \Rightarrow_{P_\phi} \ldots$$

Since ϕ-reduction on coercions is strongly normalizing, it follows that P-reductions must occur infinitely often; P strictly decreases the order $\sqsubset$ on completions (see Definition 5.2.1 and the

[6]By "fixing" a pattern of type recursion we mean precisely, in the case of Σ^μ, that we hold the recursion coercions fixed, and, in the case of Σ^∞, that we remove the congruence $\approx$.

Monotonicity Lemma 5.2.2) and so it follows that we can find arbitrarily $\sqsubset$-small P_ϕ- reducts of M', in the sense that, for some $\xi_0 \in \Theta(M)$, the sequence (τ_i) given by

$$\tau_i \equiv \mathcal{T}_M(\xi_0)(M_i)$$

contains an infinite descending $<$-chain,

$$\tau_{i_1} < \tau_{i_2} < \dots$$

determined by the terms of the infinite reduction sequence starting from M'.

Now, by weak normalization, M' has a P_ϕ-normal form, N. Choose, according to the argument above, a P_ϕ-reduct Q_N of M' such that the type τ_Q given by

$$\tau_Q \equiv \mathcal{T}_M(\xi_0)(Q_N)$$

satisfies

$$\tau_N \not\leq \tau_Q$$

where $\tau_N \equiv \mathcal{T}_M(\xi_0)(N)$; this can be done by choosing $Q \equiv M_i$ for sufficiently large i, since M' was assumed to be error-free.

Since P_ϕ is confluent (Theorem 4.4.12), it follows that

$$Q_N \Rightarrow^+_{P_\phi} N$$

because N is a normal form; note that we must have a *non-empty* reduction from Q_N to N, since an empty one would specifically imply that $\tau_N \equiv \tau_Q$.

We have the following situation:

$$\begin{tikzcd} M \arrow[r, "P_\phi"] \arrow[dr, "(P_\phi)^*"'] & M_1 \arrow[r, "P_\phi"] & {} \arrow[r, dotted, no head] & Q_N \arrow[dll, "(P_\phi)^+"] \\ & N & & \end{tikzcd}$$

It then follows, by Monotonicity (Lemma 5.2.2), that we must have

$$N \sqsubseteq Q_N$$

But this implies specifically that

$$\tau_N \leq \tau_Q$$

Contradiction.

The case of reduction in system **D** can be handled analogously. □

5.5.2 Normalization and β-reduction

We close our investigation of the normalization problem with the proof of a possibly interesting connection between normalizability under ϕ-reduction of a completion and normalizability under β-reduction of the underlying pure term. We use the framework of **D** and we disregard the conditional construct, for simplicity. The proof is somewhat technical and, as is often the case in syntactical proofs of λ-calculus, a little lengthy. The property proven is not, however, used anywhere else in this report, and this section is, admittedly, somewhat academic. We feel, though, that it is of some interest, in its own right, to be aware of the connections shown here. However, if the reader is not interested in this topic, this section can be skipped.

That any connection between ϕ- and β-conversion as indicated above should persist is perhaps not so surprising, once we take the view of the equational theory E as a kind of static abstraction of the data-flow of the underlying term. This view was suggested in the Introduction to this report, and our development here can be seen as a technical underpinning of this view. The property which we ultimately want to prove here is this:

- If a pure λI-term M is weakly normalizing under β-reduction, then ϕ-reduction is strongly normalizing on any completion of M.

First, note that the assumption of weak normalization of M wrt. β is equivalent, by the conservation Theorem of λ-calculus ([Bar84]), to strong normalization. Thus, by assumption, every β-reduction sequence starting from M is finite.

This property may throw some light on the conjecture (made in [Hen94]) that ϕ-reduction always terminates, on λI-completions. The property tells us that, in case termination fails, this must be somehow due to non-termination of the underlying program. This makes it natural to think that non-termination should be caused by data-flow cycles in the program. The obvious idea here is that such cycles would cause ϕ-reduction to become cyclic in some cases. However, we know that the *oriented* version of ϕ (reduction P_ϕ) is not cyclic. And in spite of the fact that we have investigated several, highly cyclic combinators, prominent among which is Ω, we have found only terminating reductions. These considerations may perhaps strengthen our belief that ϕ is in fact normalizing.

Embedding of D into the simply typed λ-calculus

First we define the simply typed λ-calculus with type language generated from the atom $\mathbf{D}$, and including typed constants corresponding to the non-trivial primitive coercions.

Definition 5.5.2 (Simply typed λ-calculus $\lambda^{\rightarrow}(\mathbf{D})$)
The simply typed calculus $\lambda^{\rightarrow}(\mathbf{D})$ is generated by the type rules of $\lambda^{\rightarrow}$ using the type language

$$\tau ::= \mathbf{D} \mid \tau \rightarrow \tau$$

and adding the *typed constants* $\mathtt{F!}, \mathtt{F?}$ with

$\vdash_{\lambda^{\rightarrow}} \ \mathtt{F!} : (\mathbf{D} \rightarrow \mathbf{D}) \rightarrow \mathbf{D}$

$\vdash_{\lambda^{\rightarrow}} \ \mathtt{F?} : \mathbf{D} \rightarrow (\mathbf{D} \rightarrow \mathbf{D})$

We write derivability in $\lambda^{\rightarrow}(\mathbf{D})$ as $\Gamma \vdash_{\lambda^{\rightarrow}} M : \tau$. □

Now define the translation on coercions:

Definition 5.5.3 (Translation $\lceil\bullet\rceil$ on coercions)

$$\begin{array}{lll}
\lceil p \rceil & \equiv & p \qquad\qquad \text{for } p = \mathtt{F!}, \mathtt{F?} \\
\lceil id \rceil & \equiv & \lambda x.x \\
\lceil c \circ d \rceil & \equiv & \lambda x.\lceil c \rceil(\lceil d \rceil x) \\
\lceil c \rightarrow d \rceil & \equiv & \lambda f.\lambda x.\lceil d \rceil(f\ (\lceil c \rceil x))
\end{array}$$

The translation $\lceil\bullet\rceil$ is extended to arbitrary coercion contexts by stipulating $\lceil[]\rceil \equiv []$. This is sometimes convenient in proofs below. □

The translation on coercions is lifted to a translation on completions, as follows.

Definition 5.5.4 (Translation $[\![\bullet]\!]$ on completions)

$$\begin{array}{lcl} [\![x]\!] & \equiv & x \\ [\![\lambda x.M]\!] & \equiv & \lambda x.[\![M]\!] \\ [\![M\,N]\!] & \equiv & [\![M]\!]\,[\![N]\!] \\ [\![[c]M]\!] & \equiv & \lceil c\rceil\,[\![M]\!] \end{array}$$

□

This translation is an embedding of type systems. The point is that the coerce rule $[CO]$ of $\mathbf{D}$ can be simulated by the application rule $[\rightarrow E]$ in the simply typed calculus.

Lemma 5.5.5 *If* $c : \tau \rightsquigarrow \sigma$ *then* $\vdash_{\lambda^{\rightarrow}} \lceil c\rceil : \tau \rightarrow \sigma$

Proof Structural induction on c. □

Proposition 5.5.6 *(Embedding)*
If $\Gamma \vdash_{\mathbf{D}} M : \tau$ *then* $\Gamma \vdash_{\lambda^{\rightarrow}} [\![M]\!] : \tau$.

Proof From the previous lemma one immediately has

- $\Gamma \vdash_{\lambda^{\rightarrow}} M : \tau$ and $c : \tau \rightsquigarrow \sigma$ together imply $\Gamma \vdash_{\lambda^{\rightarrow}} \lceil c\rceil M : \sigma$.

Then one can prove the claim by induction on the derivation of $\Gamma \vdash_{\mathbf{D}} M : \tau$. In the case where $\Gamma \vdash_{\mathbf{D}} M : \tau$ is $\Gamma \vdash_{\mathbf{D}} [c]M' : \sigma$ with $c : \tau \rightsquigarrow \sigma$ and $\Gamma \vdash_{\mathbf{D}} M' : \tau$, we get by induction that $\Gamma \vdash_{\lambda^{\rightarrow}} [\![M']\!] : \tau$, and so by $(\bullet)$ $\Gamma \vdash_{\lambda^{\rightarrow}} \lceil c\rceil[\![M']\!] : \sigma$, which verifies the claim in this case, since $[\![M]\!] \equiv \lceil c\rceil[\![M']\!]$. The other cases are straight-forward and left to the reader. □

We consider the relation between ϕ-reduction on $\mathbf{D}$-completions and $\beta\eta\phi$-reduction on the translated simply typed terms. We establish correspondences between these reduction relations. As a consequence we can show that ϕ-reduction on a $\mathbf{D}$-completion of a pure λI-term having a β-normal form always terminates.

We consider the relation between ϕ and $\beta\eta$ on simply typed terms. Using lemmas proven in Appendix B.4, we can establish a simulation property for the translation $[\![\bullet]\!]$:

Proposition 5.5.7 *(Correspondence)*
Let M *be a* $\mathbf{D}$*-completion with*

$$M =_E M' \rightarrow_{\phi} N$$

Then

1. *There exist* $\lambda^{\rightarrow}(\mathbf{D})$*-terms* Q, S *such that* $[\![M]\!] \rightarrow^*_{\beta\eta} S$, $[\![M']\!] \rightarrow^*_{\beta\eta} S$, $S \rightarrow^*_{\beta\eta\phi} Q$, *and* $[\![N]\!] \rightarrow^*_{\beta\eta} Q$, *and in particular,*
2. *If* M *is a* λI*-completion, then one has in addition that* $S \rightarrow^+_{\beta\eta\phi} Q$.

$$\begin{tikzcd}
M \arrow[rr, equal] \arrow[d, dash] & & M' \arrow[r, "\phi"'] \arrow[d, dash] & N \arrow[d, dash] \\
{[\![M]\!]} \arrow[rr, equal, "\beta\eta"] \arrow[dr, dashed, dash, "(\beta\eta)^*"'] & & {[\![M']\!]} \arrow[r, "\eta\phi"'] \arrow[dl, dashed, dash, "(\beta\eta)^*"] & {[\![N]\!]} \arrow[d, dashed, dash, "(\beta\eta)^*"] \\
 & S \arrow[rr, dashed, dash, "(\beta\eta\phi)^{*/+}"'] & & Q
\end{tikzcd}$$

Proof Proof in Appendix B.4, Proposition B.4.10. □

By repeated application of the previous Proposition together with confluence of $\beta\eta$ (Lemma B.4.9), one immediately has that arbitrary $\Rightarrow_\phi$-reduction sequences on $\mathbf{D}$-completions of λI-terms are simulated in a lock-step fashion by the translated terms:

Corollary 5.5.8 *Let M be a $\mathbf{D}$-completion with*

$$M \Rightarrow_\phi^n N$$

Then

1. *There exist $\lambda^\rightarrow(\mathbf{D})$-terms Q, S such that $[\![M]\!] \rightarrow_{\beta\eta}^* S$, $[\![M']\!] \rightarrow_{\beta\eta}^* S$, $S \rightarrow_{\beta\eta\phi}^* Q$, and $[\![N]\!] \rightarrow_{(\beta\eta)}^* Q$, and in particular,*
2. *If M is a λI-completion, then one has in addition that $S \rightarrow_{\beta\eta\phi}^m Q$ with $m \geq n$.*

In particular, one has therefore

Corollary 5.5.9 *If M is a $\mathbf{D}$-completion of a λI-term, and there exists an infinite $\Rightarrow_\phi$-reduction sequence starting from M, then there exists an infinite $\beta\eta\phi$-reduction sequence starting from $[\![M]\!]$.*

We may note that it is perfectly possible to construct an infinite $\beta\eta\phi$-reduction sequence starting from a $\lambda^\rightarrow(\mathbf{D})$-term, as the following example shows.

Example 5.5.10 Let

$$\overline{\Omega} \equiv (\lambda x : \mathbf{D}.(\texttt{F?}\; x)x)\; (\texttt{F!}\; (\lambda x : \mathbf{D}.(\texttt{F?}\; x)x))$$

This is a well typed completion of $\Omega \equiv (\lambda x.xx)\; (\lambda x.xx)$ at $\mathbf{D}$. It is easy to verify that $\overline{\Omega}$ has no $\beta\eta\phi$-normal form. □

Some comments may be in place in relation to this example. The first is that one obviously cannot construct an infinite $\beta\eta$-reduction sequence starting from $\overline{\Omega}$, because this is a simply typed term. Thus, we get a very syntactical view of the strong normalization property for simply typed terms: For any term with a non-terminating $\beta\eta$-reduction to be typable in the simply typed λ-calculus it would have to be *coerced* in such a fashion that coercions would eventually accumulate at blocking positions in the term. For instance, with $\overline{\omega} \equiv \lambda x : \mathbf{D}.(\texttt{F?}\; x)x$ we have $\overline{\Omega} \rightarrow_\beta (\texttt{F?}(\texttt{F!}\overline{\omega}))(\texttt{F!}\overline{\omega})$. Here one sees that a ϕ-redex is accumulated by the β-contraction, blocking all further β-redices that sit in the underlying term.

Another comment we may make is the following

Observation 5.5.11 Whereas it is perfectly possible, as we have just seen, to have infinite $\beta\eta\phi$-reductions starting from terms of the form $[\![M]\!]$ where M is a $\mathbf{D}$-completion, *this is only possible if the underlying term $\tilde{M}$ already has an infinite $\beta\eta$-reduction sequence starting from it.* This follows from the observation that all abstractions introduced by the translation $[\![\bullet]\!]$ are *linear*. Hence, all $\beta\eta$-reductions involving only λ's originating from the translation must terminate. Since ϕ-reduction on coercions is strongly normalizing, it follows that any infinite $\beta\eta\phi$-reduction sequence starting from $[\![M]\!]$ must contain infinitely many $\beta\eta$-reductions of redices that are present in the underlying term $\tilde{M}$. It is not difficult to see that these reductions make up a reduction sequence of the underlying term. It follows that we can construct an infinite $\beta\eta$-reduction sequence starting from $\tilde{M}$. $\Box$

As an immediate consequence of this observation and our previous results in this section together with classical properties of the λ-calculus, we get the following partial termination result for $\Rightarrow_\phi$ reduction: It is guaranteed to terminate on all λI-completions of terms with a β-normal form. In details:

Corollary 5.5.12 *Suppose M is a $\mathbf{D}$-completion of a pure λI term $\tilde{M}$. If there is an infinite $\Rightarrow_\phi$ reduction starting from M, then $\tilde{M}$ has no β-normal form.*

Proof By assumption and Corollary 5.5.9, there is an infinite $\beta\eta\phi$-reduction starting from $[\![M]\!]$. By our observation above, this implies that there is an infinite $\beta\eta$-reduction starting from $\tilde{M}$. By η-postponement (see [Bar84], Cor. 15.1.5.), this implies that there is an infinite β-reduction starting from $\tilde{M}$. By the Conservation Theorem for λI (see [Bar84], 11.3.4. with Cor. 11.3.5., also Thm. 9.1.5.), it follows that $\tilde{M}$ has no β-normal form. $\Box$

Chapter 6

Simple polymorphic inference

This chapter takes first steps towards practical completion inference based on the calculus Σ^{∞} (Chapter 4) of polymorphic discriminative sums with regular recursive types.

We quickly introduce Hindley-Milner style polymorphism (first section) in order to fix notations.

We then (second section) introduce the full dynamically typed polymorphic system. The final, new feature of formal machinery of polymorphic dynamic typing is introduced here. This is the notion of *coercion parameterization*, as originally suggested in [Hen92]. This leads to a stratification of the extended Hindley-Milner types to comprise an additional class of so-called *coercive types*. The resulting system can be seen as a form of polymorphic *qualified type system* in the sense of [Jon92] [Jon94].

The third section describes a simple but very efficient constraint based completion inference algorithm, descending from Henglein's near-linear constraint-solving algorithm for higher-order binding time analysis ([Hen91]), which was the basis of the algorithm for **D** described in [Hen92]. In this specification we leave out the polymorphic **let**-construct, restricting the use of polymorphism to the coercion constants. We show that completion inference in our stronger framework *simplifies* constraint solving in certain respects. This has to do with the fact that the *cascading effect* of the monomorphic type **D** (cf. Example 4.6.6) is eliminated due to the presence of polymorphic coercions and recursive types.

The fourth section gives correctness proofs for the constraint solution method. Operational correctness is based on a simple relation between the behaviour of a completion and that of its underrlying program.

The fifth section describes an implementation of the algorithm.

The sixth section describes a simple-minded extension to incoporate **let**-polymorphism, essentially by copying constraint sets. The most important part of this section, which should be stressed here, is the discussion (first subsection of the sixth section) of the problem of *modularity and polymorphic safety.* This is the first of a series of recurrent discussions in the remainder of this report, concerning the fact that, for reasons of *operational safety*, the completion inference methods suggested for soft typing as well as the method suggested in earlier sections of this chapter, are *all non-modular* in the sense that they rely in an essential way on *global, non-compositional* analyses. The limitation of non-modularity is established for the system presented in this chapter; it is established for the soft type systems in section seven (see below.) We show by examples that the simple notion of operational correctness used in the fourth section is not adequate to describe soundness conditions for completeness inference in a modular setting. This is the main reason why the inference system presented in this chapter is called *simple.* The introduction of coercion parameterization is intended as a means of overcoming the limitation of non-modularity.

In section seven, we give examples of the how the comletion inference system works. In particular, we illustrate the limitations of the inference system in its present form. We also briefly illustrate the potential practical importance of the class $\mathcal{C}_{p*}$.

Finally, the last section contains a comparison of dynamic typing as developed in this chapter with systems of soft typing, with special emphasis on the Rice system, to which our system is most closely related. We show, by simple examples, that the systems of soft typing are non-modular. We point out that, in some respects, the present framework is more expressive in principle than the Rice-system of soft typing.

6.1 Polymorphic type systems

We shall consider polymorphic λ-calculi of the two varieties introduced and studied by, respectively, Curry and Hindley [Hin69], and Damas and Milner [Mil78], [DM82]; the latter is also known as the calculus of ML-*like polymorphism*, since it represents the core of the type system of ML ([MTH90], [MTH91]) and its relatives. In order to deal with both kinds of systems in one framework, we shall follow the presentation and classification of [Hen89].

λ-terms

The λ-terms of the polymorphic calculi are the usual λ-terms with the addition of the **let**-construct:

$$M ::= \ldots \mid b \mid \mathbf{let}\ x = M\ \mathbf{in}\ M$$

The construct is well known from ML, and it allows a restricted form of *second order polymorphism*(see [GLT89]). We assume also some set of *basic* constants (like numbers and truth values) ranged over by b.

Type expressions

In general, we shall consider types of two categories, *monotypes* and *type schemes*, ranged over by τ and σ, respectively (possibly primed, subscripted etc.):

$$\begin{array}{rcl} \tau & ::= & A \mid \alpha \mid \tau \to \tau \\ \\ \sigma & ::= & \tau \mid \forall\alpha.\tau \end{array}$$

Here A ranges over a set of *atomic types*, such as $\mathbf{B}$ (booleans), $\mathbf{Z}$ (integers), $\mathbf{R}$ (reals) etc.

The two type systems of interest are defined by the set of rules of Figure 6.1. We assume a function *Typeof* assigning types to basic constants.

$$\Gamma, x:\sigma \vdash x:\sigma \qquad [AX]$$

$$\Gamma \vdash b : \mathit{Typeof}(b) \qquad [bas]$$

$$\frac{\Gamma, x:\tau \vdash M:\tau'}{\Gamma \vdash \lambda x:\tau.M : \tau \to \tau'} \qquad [\to I]$$

$$\frac{\Gamma \vdash M:\tau \to \tau' \quad \Gamma \vdash N:\tau}{\Gamma \vdash M\,N : \tau'} \qquad [\to E]$$

$$\frac{\Gamma \vdash M:\mathbf{B} \quad \Gamma \vdash N:\tau \quad \Gamma \vdash N':\tau}{\Gamma \vdash (\mathbf{if}\ M\ N\ N') : \tau} \qquad [IF]$$

$$\frac{\Gamma \vdash M:\sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall\alpha.\sigma} \qquad [GEN]$$

$$\frac{\Gamma \vdash M:\forall\alpha.\sigma}{\Gamma \vdash M : \sigma\{\alpha := \tau\}} \qquad [INST]$$

$$\frac{\Gamma \vdash M:\tau \quad \Gamma, x:\tau \vdash N:\sigma'}{\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : \sigma'} \qquad [LET^M]$$

$$\frac{\Gamma \vdash M:\sigma \quad \Gamma, x:\sigma \vdash N:\sigma'}{\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : \sigma'} \qquad [LET^P]$$

Figure 6.1: Typing Rules

We now define the two basic polymorphic calculi we shall be concerned with:

Definition 6.1.1 (Curry-Hindley calculus, CH)
The *Curry-Hindley* calculus is obtained from the rules of Figure 6.1 by leaving out the rule $[LET^P]$. □

Definition 6.1.2 (Damas-Milner calculus, DM)
The *Damas-Milner* calculus is obtained from the rules of Figure 6.1 by leaving out the rule $[LET^M]$. □

The *Damas-Milner* system is also often referred to as the *Hindley-Milner* system in the literature. Note that the **let**-construct does not play any essential rôle in the system CH. The polymorphism present in CH amounts to a *top level polymorphism*, where different occurrences of a function may recieve different types. Thus, in CH, one does not have to write more than one identity function. This is no more than the substitution lemma for the simply typed λ-calculus. The **let**-construct adds nothing to this in CH, and one could regard it as definable in the usual way:

$$\mathbf{let}\ x\ =\ M\ \mathbf{in}\ N \equiv (\lambda x.N)\, M$$

This is *no longer* the case in DM, since the **let**-construct in that system may type where its translation does not.

To illustrate, we can type the expression $I\,I$ in CH ($I\ \equiv\ \lambda x.x$) but we cannot type the expression

$$\mathbf{let}\ I\ =\ \lambda x.x\ \mathbf{in}\ I\,I$$

in CH. The expression types in DM (as in ML), however. The expression

$$(\lambda I.I\,I)\,(\lambda x.x)$$

types in *neither* of the systems CH, DM, or ML.

Principal typings

We briefly recall the notion of principal typings. The *generic instance* preordering $\preceq$ on type expressions is given by

$$\forall\alpha_1 \ldots \alpha_n.\tau \preceq \forall\beta_1 \ldots \beta_n.\tau\{\alpha_1 \ldots \alpha_n := \tau_1 \ldots \tau_n\}$$

for any monotypes $\tau_1 \ldots \tau_n$, and where every β_i is not free in $\forall\alpha_1 \ldots \alpha_n.\tau$.

In any system S under consideration, we say that σ is a *principal type scheme* for expression M under the assumption set Γ in S, if and only if

1. $\Gamma\ \vdash_S\ M : \sigma$, and
2. whenever $\Gamma\ \vdash_S\ M : \sigma'$, one has $\sigma \preceq \sigma'$.

We say that S has the *principal typing property*, if and only if it holds for every M that, whenever $\Gamma\ \vdash_S\ M : \sigma$ for some Γ, σ, then M has a principal type in S under Γ; in short form: every term which has a type also has a principal one. It is well known (see, *e.g.*,[DM82], [Hen89]) that the systems CH and DM have the principal typing property. It is also well known that Milner's algorithm $\mathcal{W}$ ([Mil78], [DM82]) automatically infers the principal type of any DM-program having a type. Algorithm $\mathcal{W}$ takes an assumption set Γ and a term M as input, and when run on a pair (Γ, M) algorithm $\mathcal{W}$ either *fails* or it *succeeds* and outputs a pair (S, τ) consisting of a substitution S and a type τ. As shown by Damas [Dam85], one has the following properties of algorithm $\mathcal{W}$:

Syntactic soundness. If $\mathcal{W}(\Gamma, M)$ succeeds with (S, τ) then $S(\Gamma) \vdash_{\mathrm{DM}} M : \tau$.

Syntactic completeness. Given Γ, M and an instance Γ' of Γ and type scheme σ such that

$$\Gamma' \vdash_{\mathrm{DM}} M : \sigma$$

one has

1. $\mathcal{W}(\Gamma, M)$ succeeds with (S, τ) such that
2. For some substitution R,

$$\Gamma' = (R \circ S)(\Gamma) \quad \text{and} \quad R(\overline{S(\Gamma)}(\tau)) \preceq \sigma$$

where the notation $\overline{\Gamma}(\tau)$ denotes the type scheme $\forall \alpha_1 \dots \alpha_n . \tau$ with $\{\alpha_1, \dots, \alpha_n\} = FV(\tau) \setminus FV(\Gamma)$.

Completeness shows that not only does $\mathcal{W}$ determine the principal type of a typable term, but it also determines the most general specialization of a given context of type assumptions for which a typing exists.

6.2 Dynamically typed polymorphic systems

In this section we introduce *dynamically typed* versions of the polymorphic calculi, CH and DM. The dynamic versions, called $\mathrm{CH}^{\leadsto}_{\mu,+}$ and $\mathrm{DM}^{\leadsto}_{\mu,+}$, both arise from adding *discriminative sum types*, *regular recursive types* and *polymorphic coercions*, as defined in Chapter 4, to the systems CH and DM; in the case of $\mathrm{DM}^{\leadsto}_{\mu,+}$ we introduce in addition the notions of *coercion parameters* and *coercive types*, leading to a form of polymorphic *qualified* type system.

We assume the general framework of Chapter 4, in particular, a ranked alphabet $\mathcal{F}$ of type constructors, ranged over by tc, indexed by $I = \{1, \dots, n\}$, and a corresponding set of type expressions

$$\tau ::= \dots \mid tc(\tau_1, \dots, \tau_{\alpha(tc)}) \mid \tau + \tau \mid \mu\alpha.\tau$$

where a non-empty set of atomic types is assumed (in case $\alpha(tc) = 0$.) We also assume a system of *polymorphic type coercions* generated from this type language, including the primitives `tc!` (positive, tagging coercions), `tc?` (negative, checking coercions) and, for $\alpha(tc) > 0$, the *induced coercions* $\mathtt{tc}(c_1, \dots, c_{\alpha(tc)})$.

System $\mathrm{CH}^{\leadsto}_{\mu,+}$

The dynamical Curry-Hindley style system $\mathrm{CH}^{\leadsto}_{\mu,+}$ arises from CH by addition of the typing rules $[CO]$ and $[\mu]$ of Figure 6.2. The type language and coercion language of $\mathrm{CH}^{\leadsto}_{\mu,+}$ are as indicated above. The relation $\approx$ (used in rule $[\mu]$ of Figure 6.2) is defined in Chapter 4. Coercions are assigned polymorphic signatures according to the rules of Figure 6.3.

System $\mathrm{DM}^{\leadsto}_{\mu,+}$

The dynamical Damas-Milner system $\mathrm{DM}^{\leadsto}_{\mu,+}$ arises from adding *all* af the rules of Figure 6.2 and Figure 6.3 to DM. Here we assume a further stratification of the type system, with the additional class, ranged over by ω, of so-called *coercive types*, in order to accomodate for *polymorphic coercion parameters*, ranged over by κ, as explained below.

The full *type language* of $\mathrm{DM}^{\leadsto}_{\mu,+}$ is given by:

$$\begin{array}{lcl} \tau & ::= & A \mid \alpha \mid tc(\tau_1, \ldots, \tau_{\alpha(tc)}) \mid \tau + \tau \mid \mu\alpha.\tau \\ \omega & ::= & \tau \mid (\tau \leadsto \tau) \Rightarrow \omega \\ \sigma & ::= & \omega \mid \forall\alpha.\sigma \end{array}$$

where τ ranges over *monotypes*, ω ranges over *coercive types* and σ ranges over *type schemes.* Note that, as in the Damas-Milner discipline, only top level type quantification is possible. We sometimes use s to range over coercion signatures of the form $\tau \leadsto \tau'$, and a coercive type may therefore have the form

$$s \Rightarrow \omega$$

The full language of coercions, ranged over by c, d of $\mathrm{DM}^{\leadsto}_{\mu,+}$ is given by

$$c ::= \kappa \mid id \mid \mathtt{tc!} \mid \mathtt{tc?} \mid \mathtt{tc}(c_1, \ldots, c_{\alpha(tc)}) \mid c \circ d$$

where κ ranges over an infinite supply of *coercion parameters* (coercion variables) to be used as a vehicle for *coercion abstraction* and *coercion instantiation.* The set of *completions* of $\mathrm{DM}^{\leadsto}_{\mu,+}$ is given by continuing the usual definition of terms by

$$M ::= \ldots \mid [c]M \mid \Lambda\kappa : s.M \mid M\{c\}$$

introducing, respectively, *coercions application*, *coercion abstraction* and *coercion instantiation.*

Note that our coercive types (ω) are a special form of *qualified types*, as introduced by M.P. Jones [Jon92], [Jon94].

Coercion calculus

The coercion calculus Σ^{∞} can be regarded as a calculus of $\mathrm{CH}^{\leadsto}_{\mu,+}$-completions by the addition of obvious rules for the **let**-construct: to the equational theory E we add the rule

$$[d](\textbf{let}\ x\ =\ [c]N\ \textbf{in}\ M) = (\textbf{let}\ x\ =\ N\ \textbf{in}\ [d]M\{x := [c]x\})$$

The polarized reduction systems also extend in obvious way to $\mathrm{CH}^{\leadsto}_{\mu,+}$.

Principal typing

The *principal typing property* is an essential property of a real-life type system. In the presence of recursive types one needs to modify the notion of principality as given in Section 6.1 in order to take the relation $\approx$ into account. This modification has been made by Coppo and Cardone in [CC91], to which the reader is referred for the definition. Using this definition, we have by previous work:

Property 6.2.1 (*Principality*)
Let S be any of the systems CH, DM. Then each of the systems $S^{\leadsto}_{\mu,+}$ has the principal typing property. Moreover, the principal typing can be automatically inferred by extension of Milner's algorithm $\mathcal{W}$. □

Let S_μ be the systems obtained by the addition of just the typing rule $[\mu]$, in each case of S. The truth of the statement above follows, in case of the systems S_μ, from previous results; Rémy [Rem89] proves that *soundness* and *completeness* of $\mathcal{W}$ still holds when performed over regular recursive types under circular unification (see also [Fag90] for discussion of this result. Also, Cardone and Coppo [CC91] prove the principality property for CH extended with regular recursive types. The principality property is easily seen to be extensible to a system with a *maximal* sum construcor (see Section 4.4.5 for the notion.)

Note that the property above says nothing about the problem of *completion inference*, although it does say that, once we *have* a completion with a type, we can infer a principal type for it, in any of the systems $S^{\leadsto}_{\mu,+}$.

Additional typing rules for $\mathrm{CH}^{\leadsto}_{\mu,+}$, $\mathrm{DM}^{\leadsto}_{\mu,+}$ and polymorphic signatures

$$\frac{\Gamma \vdash M : \tau \quad c : \tau \leadsto \sigma}{\Gamma \vdash [c]M : \sigma} \qquad [CO]$$

$$\frac{\Gamma \vdash M : \tau \quad \tau \approx \sigma}{\Gamma \vdash M : \sigma} \qquad [\mu]$$

$$\frac{\Gamma, \kappa : s \vdash M : \omega}{\Gamma \vdash \Lambda\kappa : s.M : s \Rightarrow \omega} \qquad [\Rightarrow I]$$

$$\frac{\Gamma \vdash M : s \Rightarrow \omega \quad \vdash c : s}{\Gamma \vdash M\{c\} : \omega} \qquad [\Rightarrow E]$$

Figure 6.2: Additional typing rules for $\mathrm{DM}^{\leadsto}_{\mu,+}$

$\mathtt{tc!} : \forall \alpha^{(k)}\beta.tc(\alpha_1, \ldots, \alpha_k) \leadsto tc(\alpha_1, \ldots, \alpha_k) + \beta$

$\mathtt{tc?} : \forall \alpha^{(k)}\beta.tc(\alpha_1, \ldots, \alpha_k) + \beta \leadsto tc(\alpha_1, \ldots, \alpha_k)$

$$\frac{c : \tau \leadsto \tau' \quad d : \theta \leadsto \theta'}{c \to d : (\tau' \to \theta) \leadsto (\tau \to \theta')}$$

$$\frac{c_i : \tau_i \leadsto \tau_i' \quad tc \not\equiv \to}{\mathtt{tc}(c_1, \ldots, c_k) : tc(\tau_1, \ldots, \tau_k) \leadsto tc(\tau_1', \ldots, \tau_k')}$$

Figure 6.3: Polymorphic coercion signatures

6.3 Simple polymorphic completion inference

By the embedding property of Section 4.6 we know that every **D**-completion can be regarded as a $\mathrm{CH}^{\leadsto}_{\mu,+}$-completion. One can therefore infer $\mathrm{CH}^{\leadsto}_{\mu,+}$-completions if one can infer **D**-completions. In [Hen92], Henglein gave a near-linear completion inference algorithm for **D**-completions. By the embedding property, this algorithm can trivially be transferred to infer completions of $\mathrm{CH}^{\leadsto}_{\mu,+}$. However, doing so is not as good as we should wish. Clearly, we would want an inference mechanism to exploit more aggressively the greater expressive power of system $\mathrm{CH}^{\leadsto}_{\mu,+}$, as was already hinted at in Section 4.6 (cf. Example 4.6.6) In particular, we should wish an inference algorithm to be *conservative* over CH_{μ} (we denote by S_{μ} the system S with the addition of regular recursive types together with the recursive typing rule $[\mu]$), in the sense that, for any given source program which already types in CH_{μ}, the algorithm should give back the *principal typing* of that completion, inserting no coercions at all. Henglein's algorithm is conservative over the simply typed λ-calculus with *Curry-Hindley* style polymorphism.

6.3.1 Henglein's algorithm

The completion inference algorithm of [Hen92] works by solving *type constraints* over the monomorphic type language of the simply typed λ-calculus generated from the base types **D** and **B**. It infers completions in **D**, and in the sequel this algorithm is sometimes referred to as algorithm $\mathcal{A}_{\mathbf{D}}$ for short. It can be specified as a three-step process, to be explained in more detail below:

1. *Constraint extraction*: For a given source term M, extract a multiset $C(M)$ of *type constraints* by a recursive-descent pass over the term

2. *Constraint solving*: Rewrite the constraint set $C(M)$ to a *normal form*, and read off the solution from the normal form constraint set.

3. *Completion construction*: From the original constraint set $C(M)$ together with its solution, construct the completion of M

The extracted constraint set consists of *formal type equalities and inequalities*,

$$C(M) = \{\tau =^{?} \tau', \tau \leq^{?} \tau'\}$$

The constraint normalization phase reduces the set [1] to one which *can be solved equationally*, *i.e.*, by a *most general unifying substitution* S, such that $S(\tau) = S(\tau')$ for every $\tau =^{?} \tau'$ and $\tau \leq^{?} \tau'$ in the normalized set. Here S is a type substitution, mapping type variables to type expressions. During the normalization phase, a type substitution is gradually built up (somewhat as a side effect of rewriting); the complete solution of the set is a type substitution obtained by composing the substitution created during normalization with the substitution created by the final equational solution step.

The algorithm runs in time $\mathcal{O}(N\alpha(N,N))$ where N is the size of the input program, and α is an inverse of Ackermann's function (α is equivalent to a very small constant for all practical purposes.) This complexity is derived from the possibility of implementing the constraint solution using an implementation of unification based on *union-find* structures (see, *e.g.*, [Sed88] 441 ff., [Tar83].)

We now define the constraint extraction function. For a λ-term M we associate a fresh type variable α_x with every λ-bound and free variable x, assuming w.l.o.g. that they are distinct;

[1] In this section, when talking about constraint sets, we mean by the word "set" a multiset.

with every subterm occurrence M' of M we associate a fresh type variable $\alpha_{M'}$. The contstraint extraction function C is defined in Figure 6.4. We refer the reader to [Hen92] for further information about the solution algorithm for **D**.

$$
\begin{array}{lllll}
\text{If } M \equiv x & \text{then } C[\![M]\!] & = & \{\alpha_M =^? \alpha_x\} \\
\text{If } M \equiv \mathit{true}, \mathit{false} & \text{then } C[\![M]\!] & = & \{\mathbf{B} \leq^? \alpha_M\} \\
\text{If } M \equiv \lambda x.N & \text{then } C[\![M]\!] & = & \{\alpha_x \to \alpha_N \leq^? \alpha_M\} \cup C[\![M]\!] \\
\text{If } M \equiv N\ P & \text{then } C[\![M]\!] & = & \{\alpha_P \to \alpha_M \leq^? \alpha_N\} \cup C[\![N]\!] \cup C[\![P]\!] \\
\text{If } M \equiv (\mathbf{if}\ N\ P\ Q) & \text{then } C[\![M]\!] & = & \{\mathbf{B} \leq^? \alpha_N, \alpha_P =^? \alpha_Q, \alpha_M =^? \alpha_P\} \cup \\
 & & & C[\![N]\!] \cup C[\![P]\!] \cup C[\![Q]\!]
\end{array}
$$

Figure 6.4: Constraint extraction function

6.3.2 A near-linear completion inference algorithm for $\mathrm{CH}^{\leadsto}_{\mu,+}$

In this section we consider our algorithm for $\mathrm{CH}^{\leadsto}_{\mu,+}$. The development is completely parallel to that of [Hen92], and in fact, the algorithm to be described arises by fairly simple modification of Henglein's algorithm $\mathcal{A}_{\mathbf{D}}$. Our algorithm uses the same constraint formalism as Henglein's, and the constraint extraction function remains the same [2] Only the process of solution is altered, and to this we turn now.

Following [Hen92] the overall development will be as follows:

- We specify a constraint normalization process by rewrite rules for constraint sets. This process is the core of completion inference.
- We characterize the syntactic form of constraints occurring in any set extraxted from a term (by the function $C[\![\bullet]\!]$); we then show that the normalization process preserves this form; we use this to give a characterization of constraints in normal form constraint sets.
- We observe that, by inspection of the form of normal form constraint sets, it is immediate that such sets can be *solved equationally*, using circular unification. Hence, every extracted set can be solved by first rewriting to normal form, followed by equational solution.
- Finally, we note that a solution to a constraint set uniquely determines a completion.

The main idea for a modification of the monomorphic algorithm $\mathcal{A}_{\mathbf{D}}$ is to make use of the strengthened type language of $\mathrm{CH}^{\leadsto}_{\mu,+}$ over that of **D**. As noted Henglein [Hen94], his algorithm actually has to *work hard in order to throw away information* during constraint solving. This is necessary due to the limited expressive power of **D**. In particular, there are two sources of information loss in $\mathcal{A}_{\mathbf{D}}$, which should not have the same effect for an inference mechanism for $\mathrm{CH}^{\leadsto}_{\mu,+}$:

[2] We leave out explicit mention of the **let**-construct; it presents no problems in CH, and specifying constraint extraction for this construct is left to the reader.

1. *Circular constraints.* In the absence of recursive types, circular constraints (see [Hen92] for a precise definition) must be eliminated. In $\mathcal{A}_{\mathbf{D}}$ this is accomplished by injecting and projecting to and from the type $\mathbf{D}$, using `F!` and `F?` as witnesses of the syntactic recursion equation $\mathbf{D} \cong \mathbf{D} \rightarrow \mathbf{D}$. Technically, $\mathcal{A}_{\mathbf{D}}$ imposes this discipline by introducing the equational constraint $\alpha =^? \mathbf{D}$ whenever a variable α appears on a cycle in the constraint set. An example of a term generating cyclic constraints is $\lambda x.xx$ which is completed to $\lambda x : \mathbf{D}.([\mathtt{F?}]x)x$ by $\mathcal{A}_{\mathbf{D}}$. In the presence of recursive types, this work is not only not necessary but of course also undesirable.

2. *Cascading constraints.* A completion inference algorithm must obey the restriction that the solutions it produces can be *realized* in the completion language. For this to be the case, the types assigned to subterms by the algorithm must be such that coercions with appropriate signatures exist. For instance, if at some point we discover that a function $\lambda x.M$ must be tagged with `F!` in $\mathbf{D}$, then this demand *cascades* into the further demand that $\lambda x.M$ be completed at $\mathbf{D} \rightarrow \mathbf{D}$ (recall Example 4.6.6) Technically, this is implemented in $\mathcal{A}_{\mathbf{D}}$ by breaking down any constraint of the form $\alpha \rightarrow \alpha' \leq^? \mathbf{D}$ into $\{\alpha =^? \mathbf{D}, \alpha' =^? \mathbf{D}\}$. To an inference algorithm for $\mathrm{CH}^{\rightsquigarrow}_{\mu,+}$, this should not always be necessary, since we can tag *any* functionally typed object with `F!`, due to the polymorphism.

We now present our solution. We first give definitions to characterize the form of extracted constraints.

Admissible constraint sets and solutions

Define the restricted class of $\mathrm{CH}^{\rightsquigarrow}_{\mu,+}$-type expressions ranged over by τ, θ as [3]

$$\tau ::= \alpha \mid tc(\tau_1, \ldots, \tau_{\alpha(tc)}) \mid \sum_{tc \in \mathcal{F}} tc(\overline{\tau})$$

Let

$$\sum_{tc} tc^*(\overline{\tau})$$

denote a sum-type where tc ranges through the constructor set $\mathcal{F}$ in the given, fixed order which is always assumed, and such that

$$tc^*(\overline{\tau})$$

is *either* $tc(\overline{\tau})$ *or* just a *type variable.* Then, define the subclass Δ by

$$\Delta ::= \alpha \mid \sum_{tc} tc^*(\overline{\Delta})$$

A set of constraints $C = \{\tau =^? \tau', \tau \leq^? \tau'\}$ is called *admissible*, if and only if it contains *only* constraints of the forms

$$\begin{array}{rcl} tc(\overline{\Delta}) & \leq^? & \Delta' \\ \\ \Delta & =^? & \Delta' \end{array}$$

A *solution* to an admissible constraint set C is a type substitution S such that

[3]The restriction on the class ranged over by τ, θ is local to the discussion of constraint normalization of the present section. Elsewhere, τ, θ range over the whole class of $\mathrm{CH}^{\rightsquigarrow}_{\mu,+}$-type expressions, unless anything else is explicitly stated.

- for $tc(\overline{\Delta}) \leq^? \Delta'$ in C there exists a *primitive positive* coercion p^+ such that

$$p^+ : tc(S(\overline{\Delta})) \leq S(\Delta')$$

- for $\Delta =^? \Delta'$ in C we have

$$S(\Delta) = S(\Delta')$$

- S is the identity on all variables not occurring in C

The reader who wishes to compare with [Hen92] may note that the class Δ corresponds to Henglein's class $\{\alpha, \mathbf{dyn}\}$, which is used to characterize the constraint sets of special interest there; there is evidently a close correspondence between the type $\mathbf{D}$ (called **dyn** in [Hen92]) and sums $\sum_i \tau_i$, as was explained in detail in Section 4.6.

Slack variables and maximal sums

Our sums above use *slack variables*, in the cases where tc^* is a variable. Slack variables allow restricted forms of subtyping relations to be expressed equationally. The idea is that an inclusion condition such as

$$X \subseteq Y$$

is reformulated as

$$\exists Z.X = Y \cup Z$$

Here Z is a slack variable, representing $Y \setminus X$. The idea of using slack variables is introduced in the context of *soft typing* in [Fag90], to which the reader is referred for more information. Slack variables are simply placeholders for as yet unknown summands; they may become instantiated during the process of unification. This strategy leads to a simplification, where the sum in effect degenerates to an ordinary type constructor (see Section 4.4.5.) So, for instance, assuming just three type constructors, say,

$$\mathcal{F} = \{\mathbf{B}, \rightarrow, *\}$$

consisting of base type $\mathbf{B}$ of booleans, the function space constructor $\rightarrow$ and a binary pairing $*$, with the ordering $\rightarrow < * < \mathbf{B}$, we shall encode a sum such as, *e.g.*,

$$\tau \rightarrow \tau' + \mathbf{B}$$

as

$$\tau \rightarrow \tau' + \alpha + \mathbf{B}$$

where α is a fresh type variable, acting as placeholder for the (as yet unknown) pair-component of the sum.

In the opposite extreme, in an "ideal" system, summation would be regarded as a commutative, associative, idempotent operator. However, as discussed in detail by Fagan [Fag90], this is probably not tractable from a computational viewpoint.

Notation

In order to write the constraint normalization rules, we fix some further notation. An expression of the form

$$\sum [\tau]_i$$

shall (as in Chapter 4) denote a sum whose i'th summand is τ. As always, it is implicitly assumed that τ occupies the correct position wrt. the ordering on type constructors. The expression

$$\sum_{\alpha} \{\alpha_{tc} := tc(\overline{\tau})\}$$

denotes the sum

$$\alpha_1 + \ldots + \alpha_{k-1} + tc(\overline{\tau}) + \alpha_{k+1} + \ldots + \alpha_n$$

where tc is the k'th constructor wrt. the ordering on constructors, and where the α_i are assumed to be fresh variables. This notation is extended to include more than one non-variable type, as in

$$\sum_{\alpha} \{\alpha_{tc}, \alpha_{tc'} := tc(\overline{\tau}), tc'(\overline{\tau}')\}$$

for $tc \not\equiv tc'$, where $tc(\overline{\tau})$ and $tc'(\overline{\tau}')$ are understood to be plugged in at the appropriate positions in a sum which otherwise consists of fresh type variables.

Constraint normalization rules

In order to define our generalized constraint normalization we define an auxiliary function $\mathcal{E}$ which sends a pair of types to a set of equational constraints, as follows:

$$\begin{array}{lcl}
\mathcal{E}(tc(\overline{\tau}), tc(\overline{\tau}')) & = & \{\tau_1 =^? \tau'_1, \ldots, \tau_n =^? \tau'_n\} \\
\mathcal{E}(\alpha, tc(\overline{\tau})) & = & \{\alpha =^? tc(\overline{\tau})\} \\
\mathcal{E}(tc(\overline{\tau}), \alpha) & = & \{\alpha =^? tc(\overline{\tau})\}
\end{array}$$

In Figure 6.5 we give the rules for constraint normalization. They are to be considered as *multiset rewrite rules.* The rules define the core of an inference algorithm which infers completions in $\mathrm{CH}^{\rightsquigarrow}_{\mu,+}$. It is sometimes referred to as algorithm $\mathcal{A}_{\mathrm{CH}}$ in the sequel. In rule $[N7]$ of Figure 6.5 below, the substitutions S should be thought of as being accumulated as a side effect of rewriting.

$$
\begin{array}{lll}
[N1] & C \cup \left\{ \begin{array}{lll} tc(\overline{\tau}) & \leq^? & \alpha \\ tc(\overline{\tau}') & \leq^? & \alpha \end{array} \right\} & \Rightarrow\ C \cup \left\{ \begin{array}{c} tc(\overline{\tau}) \leq^? \alpha \\ \mathcal{E}(tc(\overline{\tau}), tc(\overline{\tau}')) \end{array} \right\} \\[2ex]
[N2] & C \cup \left\{ \begin{array}{lll} tc(\overline{\tau}) & \leq^? & \alpha \\ tc'(\overline{\tau}') & \leq^? & \alpha \end{array} \right\} & \Rightarrow\ C \cup \{\alpha =^? \sum_\alpha \{\alpha_{tc}, \alpha_{tc'} := tc(\overline{\tau}), tc'(\overline{\tau}')\}\} \\[2ex]
[N3] & C \cup \{tc_i(\overline{\tau}) \leq^? \sum[\tau']_i\} & \Rightarrow\ C \cup \mathcal{E}(tc_i(\overline{\tau}), \tau') \\[2ex]
[N4] & C \cup \{c =^? c\} & \Rightarrow\ C \\[2ex]
[N5] & C \cup \{\sum_i \tau_i =^? \sum_i \tau_i'\} & \Rightarrow\ C \cup \bigcup_i \mathcal{E}(\tau_i, \tau_i') \\[2ex]
[N6] & C \cup \{\tau =^? \alpha\} & \Rightarrow\ C \cup \{\alpha =^? \tau\} \quad \text{if } \tau \text{ is not a variable} \\[2ex]
[N7] & C \cup \{\alpha =^? \tau\} & \overset{S}{\Rightarrow}\ S(C) \quad \text{where } S = \{\alpha := \tau\}
\end{array}
$$

Side conditions :

In rule $[N2]$, $tc \not\equiv tc'$.

In rule $[N3]$, τ' is either a variable, or of the form $\tau' \equiv tc_i(\overline{\tau}'')$.

In rule $[N4]$, c ranges over atomic types (type constructors of arity 0)

Figure 6.5: Constraint normalization rules for $\mathcal{A}_{\text{CH}}$

6.4 Correctness

We consider correctness properties of the constraint normalization procedure outlined above. The first subsection establishes that every extracted set of constraints has a minimal solution and that the method is sound and complete for the class $\mathcal{C}_{pf}$. The second subsection establishes operational soundness of inferred completions. The soundness conditions are a simple relation between the operational behaviour of a completion and that of the underlying program.

6.4.1 Soundness and completeness of normalization

We first characterize extracted constraint sets and the constraint normalization process.

Lemma 6.4.1 *(admissibility)*

1. *For every term M, the set $C[\![M]\!]$ is admissible.*
2. *If C is admissible, and $C \Rightarrow C'$ then C' is admissible.*
3. *The relation $\Rightarrow$ is strongly normalizing*
4. *For every term M, the set $C[\![M]\!]$ has an admissible normal form under $\Rightarrow$*

Proof The first property is easily verified by inspection of the definition of admissible sets and the definition of $C[\![\bullet]\!]$.

For the second property, note that the class Δ is closed under substitutions $S = \{\alpha := \Delta'\}$: If $\tau \in \Delta$, $\tau' \in \Delta$ then $\tau\{\alpha := \tau'\} \in \Delta$. This is easily established by structural induction on $\tau \in \Delta$. Hence, the substitution generated by the substitution rule $\overset{S}{\Rightarrow}$ preserves admissibility. It is easy to verify that the remaining rules also preserve admissibility.

For strong normalization, note that all inequality rules eliminate an inequality sign from the constraint set. It is easy to verify that an equational set can only be rewritten finitely.

The last claim follows from the previous ones. □

From this Lemma we have that any extracted constraint set reduces to an admissible normal form. From such a set the following characterization can be read off immediately:

Lemma 6.4.2 *(Form of normal form constraint sets)*
Any normal form C_n of an admissible constraint set satisfies the following properties:

1. *For every inequality constraint $\tau \leq^? \theta$ in C_n, the right-hand side θ is a type variable.*
2. *There is at most one constraint of the form $\tau \leq^? \alpha$ in C_n for every type variable α.*
3. *No constraints of the form $\Delta =^? \Delta'$ are in C_n.*

Proof It is easy to verify, by inspection of the constraint normalization rules, that any admissible set which does not satisfy one or more of the conditions mentioned must contain a redex for the constraint normalization rules of Figure 6.5. □

By the form of normalized constraint sets, it is immediate that such sets can be solved *equationally*, using circular unification. Composing this solution with the substitutions assembled during normalization rewriting yields the output of constraint solution. We call this the *minimal solution*, since it resolves inequalities by equalities, wherever possible.

In more detail, write

$$C \overset{id}{\Rightarrow} C'$$

if C rewrites to C' in one step using a rule other than $[N7]$ of Figure 6.5, and let

$$C \overset{S_i}{\Rightarrow}_k C_n$$

be a reduction of length k to normal form of the constraint set C defining the substitution S_i at the i'th rewrite step; furthermore, let S_E denote the equational solution substitution to C_n. Then the solution substitution S to C induced by the normalization is defined to be

$$S = S_E \circ S_k \circ \ldots \circ S_1$$

Remark 6.4.3 The previous Lemma is parallel to Henglein's normal form characterization in [Hen92] with the only exception that Henglein's normal forms cannot contain *cycles* which may arise as a result of cyclic unification. See [Hen92] for the definition of a cyclic constraint set. The restriction is left out here, since we have recursive types in our type language. □

The following Lemma ensures that the constraint normalization process looses no solutions. It corresponds to Henglein's Theorem 5 in [Hen92].

Lemma 6.4.4 *(Completeness of constraint normalization)*
If $C \overset{S}{\Rightarrow} C'$ and $\mathcal{U}$ is the set of solutions of C' then $\{U \circ S \mid U \in \mathcal{U}\}$ is the set of solutions of C.

Proof The claim is verified by inspection of each rule of Figure 6.5. In case the rule used is not $[N7]$, the substitution S is just the identity. For instance, for rule $[N1]$, taking the case $tc \equiv \rightarrow$, we can reason as follows: If $\theta \equiv \tau \rightarrow \tau'$, or $\theta \equiv \tau'' \rightarrow \tau'''$, then clearly the constraint can only be satisfied with the identity coercion as witness, and in case the right-hand side of the rule is also satisfied. In the remaining cases where θ is not one of the types $\tau \rightarrow \tau'$ or $\tau'' \rightarrow \tau'''$, any solution must have $\theta \equiv (\tau \rightarrow \tau') + \tau''''$, and the solution must be witnessed by `F!`, since `F!` is the only coercion with a function type in the domain of its signature. Since this type is preserved in its range type, it follows that the double constraint on the left-hand side of rule $[N1]$ can only be solved, if the constraints on the right-hand side have a solution. The remaining rules are dealt with similarly. See also [Hen92], Theorem 5. □

Recall from Section 5.3 the notion of *completion classes*, in particluar the definition of the class C_{pf}. Recall also the notion of a *local ϕ-normal form.*

Theorem 6.4.5 *(Soundness and completeness) The solutions of $C[\![M]\!]$ are in a one-to-one correspondence with the set of local ϕ-normal form completions of M in C_{pf}.*

Proof As in [Hen92] (Theorem 4) which uses the argument given in [Hen91] for Theorem 1 of that reference. □

The theorem entails that we can obtain a completion determined by the output of constraint solution.

It follows from well known results in type inference (see [Wan87]) that the type of a well typed CH-term can be inferred by purely *equational* reasoning over type constraints, using unification [4] It is also sufficiently clear from [Wan87] that leaving out the rules $[N2] - [N5]$ of our constraint normalization procedure gives back the standard type inference algorithm for CH. Hence we have

[4]In the presence of recursive types, circular unification.

Property 6.4.6 (*Conservativity*)
If M is a well typed CH$_\mu$-term, then $\mathcal{A}_{\text{CH}}\,(M) \equiv M$, and the solution substitution of $\mathcal{A}_{\text{CH}}$ determines the principal CH$_\mu$-type of M. □

6.4.2 Correspondence and operational soundness

In this section we introduce the notions of *operational correspondence* and *operational soundness.* They describe a relation between the operational behaviour of a completion and its underlying pure term. As we shall argue below, the relation considered here is very simple, and it only makes sense for inference systems which are restricted in a sense we shall make clear. The completion inference as developed in the present chapter satisfies the soundness property, as does also the system of *soft typing* developed at Rice. Later, in Chapter 7, we shall consider a more complicated inference system for which the notion of soundness given here is not appropriate.

In general, suppose we have a way of translating a pure term M to a (well typed) completion $\overline{M}$ of M. One example of such a translation we have already seen, namely the translation to so-called canonical completions. We should like to lay down some principles that are sufficient to guarantee that such a translation *preserves operational semantics* in some sense; we generally refer to properties of (operational) semantics preservation as *operational soundness.*

Intuitively, soundness is a matter of *correspondence* between the operational behaviour of any translated completion and its underlying term: The completion should in all cases behave sufficiently similarly to the pure term. In order to make these ideas precise we must be able to talk about the *pure semantics* of a pure term. Recall, therefore, the notion of pure semantics as given in Definition 1.6.4. With this definition in place we can define the following simple notion of soundness and correspondence. It essentially says that the only difference in operational behaviour of a completed term as compared to that of the source term is that the former may yield a run-time type error (represented by the error-element $\boldsymbol{\varepsilon}$) where the evaluation of the latter gets *stuck* (represented as the wrongness element $\boldsymbol{\omega}$.)

Definition 6.4.7 (Correspondence, soundness)
Let $\overline{\bullet}$ be a translation from pure terms to well typed completions. The the translation $\overline{\bullet}$ is called *sound* wrt. the pure semantics $\Downarrow^{*p}$, if and only if the following correspondence conditions are all satisfied for every pure term M:

1. $M \Downarrow^{*p} V_t$ if and only if $\overline{M} \Downarrow^{*} V'$ with $V_t \cong V'$
2. M diverges if and only if $\overline{M}$ diverges
3. $M \Downarrow^{*p} \boldsymbol{\omega}$ if and only if $\overline{M} \Downarrow^{*} \boldsymbol{\varepsilon}$

□

Similar correctness conditions are defined in [WrCa94] and [Wri94], where it is shown that softly typed programs always correspond, in this sense, to their source programs. By the properties shown in the previous section, we can regard the completion process as determining a translation function, $\mathcal{A}_{\text{CH}}\,(\bullet)$, which takes a pure term to its completion under algorithm $\mathcal{A}_{\text{CH}}$. Now, by standard techniques, *e.g.*, syntactic arguments based on *subject reduction* (cf., *e.g.*, [WF91] and [Wri94]) we can show that the *type soundness* property is satisfied in our system (see Property 1.6.8. Moreover, we can verify, by structural induction on pure term M together with inspection of the rules for constraint normalization, that

Lemma 6.4.8 *For every pure term M, $\mathcal{A}_{\text{CH}}\,(M)$ is in $\sum_{pf}^{\infty}$.*

Note that this characterization already makes it intuitively quite clear that we have correspondence, because any $\sum_{pf}^{\infty}$-completion defers run-time type checks as far as possible, placing only negative primitive coercions at destruction points. More formally, the lemma just stated can be used in a correspondence proof based on type soundness, in close analogy with the proof in [Wri94]. We therefore leave out a detailed proof, and we confine ourselves to recording

Theorem 6.4.9 *The translation $\mathcal{A}_{\mathrm{CH}}\ (\bullet)$ is operationally sound.*

In fact, one can prove that *any* $\sum_{pf}^{\infty}$-completion is sound wrt. *any* of the semantics (CBN or CBV) considered in this report.

The soundness conditions used in this section are in one sense too weak and in another sense too strong to deal meaningfully with more complicated systems of inference, we might wish to consider. As an instance of the notions' being too strong, we note that, if we move to a system which produces completions with *induced coercions*, then we can no longer expect the conditions to be satisfied, and they would not be meaningful correctness criteria for such a system. To see why this is so, recall our operational semantics of coercion evaluation in Figure 1.4, and consider the rule $[C{\rightarrow}]$ for the induced function space coercion. This rule introduces an η-expansion on the underlying term, and hence the relation $\cong$ is too weak to relate a good completion to its underlying term. More complicated notions of correctness would be required for such a system. For instance, it might be pertinent to consider correctness with observation of the operational behaviour at *ground types* only. This, in turn, would lead to the need for stronger proof methods in a correctness proof; for a notion of correctness with reference to ground values one would typically have to employ a *logical relation* (see [Wyn94]) or a notion of *computability* (see [Gun92].)

As an instance of the notions' being too weak, we note that they only relate the operational behaviour of completions to that of their underlying term, considered *independent of any further program context*. This reflects an important *limitation* of the inference mechanism as developed in the present chapter, as well as of extant systems of *soft typing*. We shall have much more to say about this later, where a more thorough discussion of soft type systems will also be found (see Section 6.6.1, Section 6.8 and Chapter 7.) There it is argued that these systems are all inherently *global* or *non-modular*, since they do not take into account the operational behaviour of the completed term within any larger program context; in particular, the techniques of inference are *unsafe* for modular inference, as is explained in the sections referred to above. Later in this report, in Chapter 7, we consider the problem of safe modular inference, and we shall see that this will require non-trivial extensions of the inference framework developed in the present chapter.

6.5 Implementation

The implementation of constraint solution to be described is very much inspired by Henglein's constraint based algorithm for efficient higher order binding time analysis in [Hen91]; in fact, it can be derived from that algorithm by relatively minor extensions and modifications in principle. It turns out that our algorithm is also very closely related to the algorithm employed in the basic flow analysis described by Bondorf and Jørgensen in [BJ93a] and [BJ93b], which can be regarded as another descendant of [Hen91].

The basic machinery consists of a *union-find* based [5] implementation of unification. The operations `union` and `find` (we assume the reader is familiar with these, and we shall not go

[5] See [Sed88], [Tar83]

into any further details about them) are brought to work on a representation of type expressions along the line [6] of the following ML declarations:

```
datatype Ty         = tyvar of variable
                    | tc of Tyref list
                    | sum of Tyref list

and      Tyval      = None
                    | Some of {Tyexp : Ty, Eq : Tyref , Leq : Tyref}

withtype Tyref      = Tyval ref

datatype Constraint = EQ  of Tyref * Tyref  (* equality constraint *)
                    | LEQ of Tyref * Tyref  (* inequality constraint *)

val      worklist   = ref ([] : Constraint list)
```

Here `tc` should be taken to range through the type constructor alphabet. Type expressions are manipulated in the form of `Tyref`'s which, in addition to the type information itself, contain information fields `Eq` and `Leq`. Of these, the reference `Eq` is used, in the standard manner, as a reference to the *union-find* equivalence class representative; an element may for instance be designated a representative (a canonical element) if and only if its `Eq`-reference refers directly to the element itself. In addition to the fields shown, a concrete implementation will contain further information (not shown here), such as, *e.g.*, a field `rank` for bookkeeping in connection with *union* by rank. The second information field shown, called `Leq`, is an *inequality reference* which keeps track of inequality constraints encountered during processing of the constraint set. It leads to an efficient implementation of the (expensive looking) multiset pattern matching which is used in the specification of Figure 6.5, on the left hand sides of the rules $[N1]$ and $[N2]$. The idea is to scan through the constraint set, represented in a global, updatable variable `worklist`; an inequality $\tau \leq^? \alpha$ results in τ being recorded under the inequality reference of the canonical element[7] associated with α, where it stays until another inequality $\tau' \leq^? [\alpha]$ is encountered ($[\alpha]$ denoting the canonical element associated with α.)

An implementation based on these ideas is shown in Figure 6.6. It uses a number of auxiliary functions, which behave as follows:

- `get_constraint()` destructively removes an element from the worklist and returns the element
- `is_sum x` is true iff `x` is a sum-type
- `leqref x` is the inequality reference of `x`
- `leqdefined x` is true iff `leqref x <> nil`
- `constr x` returns the top-level type constructor of `x`

[6]The representation used here is, of course, by no means the only one possible. We have chosen what we believe is a conceptually fairly simple way of doing it; for a somewhat more refined representation, see the works by Bondorf and Jørgensen cited above.

[7]Note that information about an element should be associated with the canonical element, since otherwise information may easily be lost under unification.

- `E(x,y)` adds the equality constraints

  $$\bigcup_i \mathcal{E}(tc(\overline{\tau}), tc(\overline{\tau}'))$$

  to the worklist, if `x` is $tc(\overline{\tau})$ and `y` is $tc(\overline{\tau}')$; otherwise, if `x` is $\sum_i \tau_i$ and `y` is $\sum_i \tau_i'$, then the equality constraints

  $$\bigcup_i \mathcal{E}(\tau_i, \tau_i')$$

  are added to the worklist

- `do_sum(x,s)` adds the equality constraints

  $$\mathcal{E}(tc_i(\overline{\tau}), \tau')$$

  to the worklist, when `x` is $tc_i(\overline{\tau})$ and `s` is $\sum_i [\tau']_i$

- `do_clash(x,y,z)` adds the equality constraint

  $$\alpha =^? \sum_\alpha \{\alpha_{tc}, \alpha_{tc'} := tc(\overline{\tau}), tc'(\overline{\tau}')\}$$

  to the worklist, when `x` is $tc(\overline{\tau})$, `y` is $tc'(\overline{\tau}')$ and `z` is α; furthermore, the inequality reference of `z` is set to `nil`

- `do_tyvar(x,y,u)` is implemented by:

  ```
  let val other = if isrep x then y else x
  in
      if (leqdefined other) then
         add_to_worklist(LEQ(find(leqref other), u))
      else
         skip
  end
  ```

  so that $\tau \leq^?$ `u` is added to the worklist, where `u` is the union of `x` and `y`, and $\tau \leq^?$ `other` is an inequality associated with that element (`other`) of `x` and `y` which is not an equivalence class representative. Note that *exactly* one of `x` and `y` must be an equivalence class representative in any context of use of the function `do_tyvar`, since `x` and `y` are always unioned immediately prior to a call to that function.

The algorithm works its way through the worklist, initialized with the set of extracted constraints $C[\![M]\!]$ for the subject program M. At each iteration, a constraint `c` is picked from the worklist, and actions are taken according to the form of the constraint, in close correspondence with the constraint normalization rules of Figure 6.5, as indicated in the commentaries in the code. New constraints may be added to the worklist during processing. Note that the case `(tc, tc)` under the analysis of an equational constraint is only there for the final equational solution step, after normalization.

```
while (! worklist) <> nil do (
      let val c = get_constraint()
      in
          case c of
            LEQ(l,r) => let val l' = find l
                            val r' = find r
                        in
                            if is_sum r' then
                               do_sum(l',r') (* [N3] *)
                            else (* l' < r' *)
                               if not (leqdefined r') then
                                  (* record inequality *)
                                  (leqref r') := ! l'
                               else (* inequality already found *)
                                  let val l'' = find (leqref r')
                                  in  (* l', l'' < r' *)
                                      if (constr l') = (constr l'') then
                                         E(l',l'')  (* [N1] *)
                                      else
                                         (* [N2] *)
                                         do_clash(l',l'',r')
                                  end
                        end
          | EQ(l,r)  => let val l' = find l
                            val r' = find r
                        in
                            if (! l') = (! r') then
                               skip
                            else
                               case (constr l',constr r') of
                                 (sum, sum) => E(l',r') (* [N5] *)
                               | (tc, tc)   => E(l',r')
                                               (* for equational
                                                  solving *)
                               | (tyvar, _) => let val u = union(l',r')
                                               in  (* [N6], [N7] *)
                                                   do_tyvar(l',r',u)
                                               end
                               | (_, tyvar) => let val u = union(l',r')
                                               in  (* [N6], [N7] *)
                                                   do_tyvar(l',r',u)
                                               end
                               | (_, _)     => skip (* [N4] *)
                        end
      end (* let val c = get_constraint() *)
    ) (* while *)
```

Figure 6.6: Constraint solution algorithm $\mathcal{A}_{\text{CH}}$

The characterizations of our inference mechanism given in the previous section were based on, among other things, the observation that extracted constraint sets are *admissible*. This restriction on the constraint extraction function $C[\![\bullet]\!]$ will be satisfied for standard constructors. For instance, for a pairing constructor **cons**, with destructors **car**, **cdr** one would have

$$\begin{array}{llcl}
\text{If } M \equiv (\mathbf{cons}\, N\, P) & \text{then } C[\![M]\!] & = & \{(cons\, \alpha_N\, \alpha_P) \leq^? \alpha_M\} \cup C[\![N]\!] \cup C[\![P]\!] \\
\text{If } M \equiv (\mathbf{car}\, N) & \text{then } C[\![M]\!] & = & \{(cons\, \alpha_M\, \beta) \leq^? \alpha_N\} \cup C[\![N]\!] \\
\text{If } M \equiv (\mathbf{cdr}\, N) & \text{then } C[\![M]\!] & = & \{(cons\, \beta\, \alpha_M) \leq^? \alpha_N\} \cup C[\![N]\!]
\end{array}$$

corresponding to the following static typing rules [8]

$$\frac{\Gamma \vdash N : \tau \qquad \Gamma \vdash P : \tau'}{\Gamma \vdash (\mathbf{cons}\, N\, P) : (cons\, \tau\, \tau')}$$

$$\vdash \mathbf{car} : \forall \alpha\beta.\, (cons\, \alpha\, \beta) \to \alpha$$

$$\vdash \mathbf{cdr} : \forall \alpha\beta.\, (cons\, \alpha\, \beta) \to \beta$$

However, this scheme is somewhat over-simplified wrt. coping with basic functions in a higher order functional language; for instance, the function **cons** of, say, *Scheme*, is not a constructor of the kind above, but rather it is a function which can be treated as a *first class* object, and it is more accurately described by the static type [9]

$$\vdash \mathbf{cons} : \forall \alpha\beta.\alpha \to \beta \to (cons\, \alpha\, \beta)$$

Hence, it is not sufficient to prescribe type constraints corresponding to *applications* of **cons**, since now **cons** is a first class object which may have to be completed just as such, as in, *e.g.*,

$$(\lambda f.M)\, \mathbf{cons}$$

Here we encounter the problem that we cannot start working with constraints like

$$\alpha \to \beta \to (cons\, \alpha\, \beta) \leq \alpha_{\mathbf{cons}}$$

within the framework developed so far, because the constraint just shown is *not* admissible. However, this problem can be solved if we pass (conceptually, at least) to consideration of the η-expanded version [10]

$$\mathbf{cons} =_\eta \lambda\eta_1.\lambda\eta_2.@_1(@_2(\mathbf{cons}, \eta_1), \eta_2)$$

[8]Note that the typings shown here are to be regarded as part of a *static* type system, we may impose on programs of a dynamically typed language; this is in distinction to the language's own (dynamic) type system, although the two systems are naturally closely related. Compare with Section 1.3.

[9]Strictly speaking, the type shown is still not quite accurate for exactly the **cons** of *Scheme*. Here we wish to illustrate a point of principle and we do not go into specifics about *Scheme* at this point. See Chapter 7 for further discussion.

[10]Note that general η-expansion is not semantically sound in a call-by-value language; but it can safely be assumed to be sound on the basic functions of the language. In any case, we *need* not think of the η-expansions as being *actually* performed on the source program.

where we have written the application $@(\bullet,\bullet)$ explicitly, using indices to distinguish occurrences. For this version, it is straight-forward to extract the constraints (constraints for **car**, **cdr** obtained in analogous fashion)

$$C[\![\textbf{cons}]\!] \;=\; \left\{ \begin{array}{c} (\mathit{cons}\ \alpha_{\eta_1}\ \alpha_{\eta_2}) \leq \alpha_{@_1} \\ \alpha_{\eta_2} \to \alpha_{@_1} \leq \alpha_{@_2} \\ \alpha_{\eta_1} \to \alpha_{@_2} \leq \alpha_{\textbf{cons}} \end{array} \right\}$$

$$C[\![\textbf{car}]\!] \;=\; \left\{ \begin{array}{l} (\mathit{cons}\ \alpha_{@}\ \beta) \leq \alpha_{\eta} \\ \alpha_{\eta} \to \alpha_{@} \leq \alpha_{\textbf{car}} \end{array} \right\}$$

$$C[\![\textbf{cdr}]\!] \;=\; \left\{ \begin{array}{l} (\mathit{cons}\ \beta\ \alpha_{@}) \leq \alpha_{\eta} \\ \alpha_{\eta} \to \alpha_{@} \leq \alpha_{\textbf{cdr}} \end{array} \right\}$$

As is readily seen, these constraints are all admissible. We have implemented a prototype completion inference system based on the ideas described in the present chapter. The prototype handles the following simple *Scheme*-like higher order functional subset:

$$\begin{array}{lcl} \mathtt{M} & ::= & \mathtt{x} \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{nil} \mid \mathtt{cons} \mid \mathtt{car} \mid \mathtt{cdr} \mid \mathtt{null?} \\ & \mid & (\mathtt{lambda}\,(\mathtt{x})\,\mathtt{M}) \mid (\mathtt{M}\,\mathtt{M}') \mid (\mathtt{if}\,\mathtt{M}\,\mathtt{M}'\,\mathtt{M}'') \mid (\mathtt{define}\,\mathtt{x}\,\mathtt{M}) \end{array}$$

where `cons`, `car`, `cdr` are treated as first class functions, as described above. The `define`-construct allows recursive definitions. In Section 6.7 below we give some examples of the workings of this inference system.

6.6 A simple extension to $\mathrm{DM}^{\leadsto}_{\mu,+}$

In this section we discuss issues arising in an attempt to extend completion inference to Damas-Milner systems with **let**-polymorphism ($\mathrm{DM}^{\leadsto}_{\mu,+}$) The most important part of the development of this section is found in Section 6.6.1. Here we introduce the important issue concerning safety and modularity of polymorphic completion inference. The second part, Section 6.6.2, outlines a simple-minded extension of algorithm $\mathrm{CH}^{\leadsto}_{\mu,+}$ to deal with **let**-polymorphism. The algorithm is probably not one we would prefer to use in practice; rather, we present it mostly for the reason that it is conceptually simple, and it will allow us to agree that the inference problem can be solved for $\mathrm{DM}^{\leadsto}_{\mu,+}$, without having to go into complicated problems at this point.

6.6.1 The problem of modularity and polymorphic safety

Before we outline the extension, we must now discuss a topic which is a major concern in the remainder of this report. This is the problem of *modularity* and *polymorphic safety* of completion inference. We have briefly mentioned this topic in passing, in the semantic discussion in Section 6.4.2, and we shall have occasion to discuss it on several occasions later (see the references given in Section 6.4.2.) The completion inference as described so far, embodied in algorithm $\mathcal{A}_{\mathrm{CH}}$, is *global* or *non-modular*, because it rests on the following assumption:

- *Assumption of globality: The object of analysis is an entire program.*

The reason why this assumption is necessary for algorithm $\mathcal{A}_{\mathrm{CH}}$ is explained in the following example:

Example 6.6.1

```
let f = (lambda (x) (if true x (cons 1 2)))
in
    (+ (f 1) 2)
end
```

The little program

$$M \equiv (\texttt{lambda}\,(\texttt{x})\,(\texttt{if}\,\texttt{true}\,\texttt{x}\,(\texttt{cons}\,1\,2)))$$

bound to f in the `let`-binding is already statically typable, at the type

$$num * num \rightarrow num * num$$

and completing M with $\mathcal{A}_{\text{CH}}$ will give back just M itself, $\mathcal{A}_{\text{CH}}\,(M) \equiv M$, inserting no coercions at all. However, this completion (*i.e.*, M itself) of M is not *safe* for the context of use, $C \equiv (+\,([]\,1)\,2)$, since a completion of $C[M]$ built from this completion (*i.e.*, M itself) of M would have to coerce the type *num* to the type *num* ∗ *num*, and *vice-versa*, as in

```
(+ ([num?][cons!](M ([cons?][num!]1))
   2)
```

This completion of $C[M]$ will generate a run-time error, even though there exists another completion of $C[M]$ which does *not* generate an error, to wit:

```
(+ [num?](M' ([num!]1)
   2)
```

where `M`$'$ is the completion of M:

```
(lambda (x) (if true ([cons!](cons 1 2))))
```

at the type

$$\forall\alpha.(num * num) + \alpha \rightarrow (num * num) + \alpha$$

Hence, the first completion of M is *unsafe* for the context C. □

In other words, the minimization performed by $\mathcal{A}_{\text{CH}}$ is *too aggressive* to allow modularity of completion while still satisfying the indispensable property of safety. Safe completion under $\mathcal{A}_{\text{CH}}$ is a *non-compositional* process:

- *Non-modularity: One cannot safely infer a completion of $C[M]$ by inserting an inferred completion of M into a completion of the context C.*

We learn from this example that minimility and safety are aims that must be kept in balance: from the point of view of minimality, the first completion `M` with no coercions is appealing, but unfortunately it is unsafe. Now, there are of course other contexts for which the completion `M` without any coercions is perfect, for instance

$$C' \equiv (\texttt{car}\,([]\,(\texttt{cons}\,3\,4)))$$

where insertion of completion `M`$'$, on the other hand, would be unfortunate, because it would force the completion of $C[\texttt{M}']$ to be

```
(car [cons?](M' ([cons!](cons 3 4))))
```

with several unnecessary coercions.

These observations naturally lead to the problem of contexts which impose *conflicting* demands. We can combine the phrases considered so far to pose the problem:

Example 6.6.2 Consider the program:

```
let f = (lambda (x) (if true x (cons 1 2)))
in
    (if tst (+ (f 1) 2) (car (f (cons 3 4))))
end
```

Here it appears that we are forced to complete the `let`-bound function in a way which introduces unnecessary coercions into the **else**-branch of the conditional in the body of the `let`-binding. □

This is where the idea, originally proposed by Henglein in [Hen92], of using *coercion parameterization* becomes important. Coercion parameterization is an important means of ensuring safety while still utilizing context specific information at each context of use. With coercion parameters we can reach a compromise between the conflicting demands posed by the contexts C and C' above, as indicated in this example:

Example 6.6.3 We can complete the program from Example 6.6.2 as

```
let f = (Lambda (k: (num * num) --> 'a)
                (lambda (x: 'a)
                        (if true x ([k](cons 1 2)))))
in
    (if tst
        (+ [num?](f{cons!} ([num!]1))
           2)
        (car (f{id} (cons 3 4))))
end
```

at the type *num*, where

```
(Lambda (k: (num * num) --> 'a) ...)
```

is the coercion abstraction

$$\Lambda\kappa : (num * num) \rightsquigarrow \alpha. \ldots$$

and the coercions `cons!`, `num!`, `num?`, `id` are taken at the signature instances

$\texttt{cons!} : num * num \rightsquigarrow (num * num) + num$

$\texttt{num!} : num \rightsquigarrow (num * num) + num$

$\texttt{num?} : (num * num) + num \rightsquigarrow num$

$\texttt{id} : num * num \rightsquigarrow num * num$

□

6.6.2 Algorithm

To infer a $\mathrm{DM}^{\rightsquigarrow}_{\mu,+}$-completion we can proceed as described in Figure 6.7. The algorithm will be referred to as $\mathcal{A}_{\mathrm{DM}}$. It is simple-minded, because it works by brute-forcely copying the constraint set associated with the **let**-bound term, producing one copy for each occurrence of the **let**-bound variable within the body of the binding. This is also how the **let**-construct is handled in the soft typing system of [AWL94][11]

The method described in Figure 6.7 is equivalent in power to completion inference by syntactic **let**-unfolding; only we copy constraint sets instead of actually unfolding the term syntacally. Copying is potentially more tractable than unfolding, since the latter may lead to an exponential blow-up of the size of the term without a combinatorial explosion taking place under copying. Since typability under **let**-polymorphism is equivalent to typability without **let**-polymorphism under **let**-unfolding (see [Mai92]), we have:

Property 6.6.4 (Conservativity)
If M is a well typed DM_μ-term, then $\mathcal{A}_{\mathrm{DM}}\,(M) \equiv M$. □

The algorithm outlined will, for example, produce the completion shown in Example 6.6.3.

[11]This is to judge by the specification in [AWL94] as well as in [AW93]. However, the exact implementation technique used in their realization of the specifications contained in these references is not known to us.

Input: A pure term M

Output: A $\mathrm{DM}^{\rightsquigarrow}_{\mu,+}$-completion of M

Method: For every expression

$$\mathbf{let}\ x = P\ \mathbf{in}\ N$$

of the original source program M with $n > 0$ occurrences of x in N, denoted $x_{(1)}, \dots, x_{(n)}$, we assume by induction that the body N can be completed to $\overline{N}$ with a fresh type variable α_i assumed for each occurrence of x in N. As a result of this, we obtain a substitution S, and we complete (by induction) n copies $P_1, \dots, P_n$ of P; here the completion of P_i is performed at the type $\alpha_{P_i} = S(\alpha_i)$, by adding this constraint to the constraint set extracted for P_i. This yields a set of completions $\{\overline{P}_i\}$ of P.

We now consult the set $\overline{P}_1, \dots, \overline{P}_n$ of completions of P to construct a parameterized completion of P, to be called $\overline{P}^{\kappa}$, as follows. For every internal coercion point p of the source expression P there are three possibilities:

1. None of the $\overline{P}_1, \dots, \overline{P}_n$ have a coercion at p
2. All of the $\overline{P}_1, \dots, \overline{P}_n$ have the same primitive (non-trivial) coercion c at p
3. Neither of the above hold

We then stipulate:

- In case 1. $\overline{P}^{\kappa}$ gets no coercion parameter and no coercion at point p
- In case 2. $\overline{P}^{\kappa}$ gets the coercion c at p
- In case 3. $\overline{P}^{\kappa}$ gets a fresh coercion parameter κ_j at p

We let $\overline{N}^{\kappa}$ arise from $\overline{N}$ by exchanging each occurrence $\overline{P}_i$ by $x\{c_{i_1}\} \dots \{c_{i_{k_i}}\}$ where the coercions $c_{i_1}, \dots c_{i_{k_i}}$ are chosen appropriately according to whether $\overline{P}_i$ has the coercion c_{i_j} in place of parameter κ_{i_j} at a coercion point p compared to $\overline{P}^{\kappa}$. In the abnormal case where $n = 0$ we simply complete P using $\mathcal{A}_{\mathrm{CH}}$. To obtain the final $\mathrm{DM}^{\rightsquigarrow}_{\mu,+}$-completion, we abstract on all the coercion parameters introduced in $\overline{P}^{\kappa}$, to get

$$\Lambda\kappa_1 : s_1. \dots \Lambda\kappa_k : s_k.\overline{P}^{\kappa}$$

Signatures s_i are guaranteed to exist such that the resulting term is well-typed; in the extreme case one can choose $s_i \equiv \alpha \rightsquigarrow \beta$.

Figure 6.7: Algorithm $\mathcal{A}_{\mathrm{DM}}$

6.7 Examples

We give some examples to show the workings of completion inference. The first example considered shows, in some detail, how constraint solving works on a concrete term. The remaining examples are in many respects focussed at giving an impression of the *limitations* of the simple completion inference methods described previously in the present chapter. The final example given also illustrates the potential power of the class $\mathcal{C}_{p*}$.

6.7.1 Constraint normalization

The first example illustrates, on a very simple term, how constraint sets are transformed during constraint solving by $\mathcal{A}_{\text{CH}}$. For this example we assume that our language is that of Σ^∞, the type language including just the boolean type and the function space. The term was discussed earlier, in Example 4.6.6.

Example 6.7.1 Consider the term M (substcripts to indicate occurrences)

$$M \equiv \mathbf{if}_0\ \mathit{true}_0\ \mathit{true}_1\ (\lambda x.\mathbf{if}_1\ \mathit{true}_2\ (x_0\ \mathit{false}_0)\ (x_1\ \mathit{false}_1))$$

The constraints extracted from M are:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathbf{B} & \leq^? & \alpha_{\mathit{true}_0} & \alpha_{\mathit{true}_1} & =^? & \alpha_\lambda \\
\alpha_{\mathbf{if}_0} & =^? & \alpha_{\mathit{true}_1} & \mathbf{B} & \leq^? & \alpha_{\mathit{true}_1} \\
\alpha_x \to \alpha_{\mathbf{if}_1} & \leq^? & \alpha_\lambda & \alpha_x & =^? & \alpha_{x_0} \\
\alpha_x & =^? & \alpha_{x_1} & \mathbf{B} & \leq^? & \alpha_{\mathit{true}_2} \\
\alpha_{x_0\mathit{false}_0} & =^? & \alpha_{x_1\mathit{false}_1} & \alpha_{\mathbf{if}_1} & =^? & \alpha_{x_0\mathit{false}_0} \\
\alpha_{\mathit{false}_0} \to \alpha_{x_0\mathit{false}_0} & \leq^? & \alpha_{x_0} & \alpha_{\mathit{false}_1} \to \alpha_{x_1\mathit{false}_1} & \leq^? & \alpha_{x_1} \\
\mathbf{B} & \leq^? & \alpha_{\mathit{false}_0} & \mathbf{B} & \leq^? & \alpha_{\mathit{false}_1}
\end{array}$$

A number of applications of rule $[N7]$ result in the substitution

$$S_1 = \left\{ \begin{array}{rcl}
\alpha_\lambda & \mapsto & \alpha_{\mathit{true}_0}, \\
\alpha_{\mathbf{if}_0} & \mapsto & \alpha_{\mathit{true}_1}, \\
\alpha_{x_0} & \mapsto & \alpha_x, \\
\alpha_{x_1} & \mapsto & \alpha_x, \\
\alpha_{x_1\mathit{false}_1} & \mapsto & \alpha_{x_0\mathit{false}_0}, \\
\alpha_{\mathbf{if}_1} & \mapsto & \alpha_{x_0\mathit{false}_0}
\end{array} \right\}$$

Using $[N1]$ and again $[N7]$, yielding $S_2 = \{\alpha_{\mathit{false}_0} \mapsto \alpha_{\mathit{false}_1}\}$, one can obtain the constraint set

$$\begin{array}{rcl}
\mathbf{B} & \leq^? & \alpha_{\mathit{true}_0} \\
\mathbf{B} & \leq^? & \alpha_{\mathit{true}_1} \\
\alpha_x \to \alpha_{x_0\mathit{false}_0} & \leq^? & \alpha_{\mathit{true}_1} \\
\mathbf{B} & \leq^? & \alpha_{\mathit{true}_2} \\
\alpha_{\mathit{false}_1} \to \alpha_{x_0\mathit{false}_0} & \leq^? & \alpha_x \\
\mathbf{B} & \leq^? & \alpha_{\mathit{false}_1} \\
\mathbf{B} & \leq^? & \alpha_{\mathit{false}_1}
\end{array}$$

Now one can apply rule $[N2]$ to the second and third constraints, to get the set

$$\begin{array}{rcl} \mathbf{B} & \leq^? & \alpha_{true_0} \\ \alpha_{true_1} & =^? & (\alpha_x \to \alpha_{x0 false_0}) + \mathbf{B} \\ \mathbf{B} & \leq^? & \alpha_{true_2} \\ \alpha_{false_1} \to \alpha_{x0 false_0} & \leq^? & \alpha_x \\ \mathbf{B} & \leq^? & \alpha_{false_1} \\ \mathbf{B} & \leq^? & \alpha_{false_1} \end{array}$$

which is in normal form. Its equational solution substitution is

$$S_{eq} = \left\{ \begin{array}{rcl} \alpha_{true_0} & \mapsto & \mathbf{B}, \\ \alpha_{true_2} & \mapsto & \mathbf{B}, \\ \alpha_{false_1} & \mapsto & \mathbf{B}, \\ \alpha_x & \mapsto & \alpha_{false_1} \to \alpha_{x0 false_0}, \\ \alpha_{true_1} & \mapsto & (\alpha_x \to \alpha_{x0 false_0}) + \mathbf{B} \end{array} \right\}$$

From this we see that the complete, accumulated solution substitution S_{sol} satisfies

$$S_{sol} \supseteq \left\{ \begin{array}{rcl} \alpha_{true_0} & \mapsto & \mathbf{B}, \\ \alpha_{true_2} & \mapsto & \mathbf{B}, \\ \alpha_{false_1} & \mapsto & \mathbf{B}, \\ \alpha_{true_1} & \mapsto & (\alpha_x \to \alpha_{x0 false_0}) + \mathbf{B} \\ \alpha_{\lambda} & \mapsto & (\alpha_x \to \alpha_{x0 false_0}) + \mathbf{B} \end{array} \right\}$$

from which we get the $\mathcal{A}_{\text{CH}}$ -completion

$$\mathcal{A}_{\text{CH}}\,(M) \equiv \textbf{if}\ \mathit{true}\,([\texttt{B!}]\mathit{true})\ ([\texttt{F!}](\lambda x.\textbf{if}\ \mathit{true}\,(x\,\mathit{false})\ (x\,\mathit{false})))$$

at the type $((\mathbf{B} \to \alpha) \to \alpha) + \mathbf{B}$. This is the completion shown in Example 4.6.6. On the other hand, using $\mathcal{A}_{\mathbf{D}}$ on M results in cascading of the tagging coercion F! at the abstraction, so that the $\mathcal{A}_{\mathbf{D}}$ -completion of M is

$$\mathcal{A}_{\mathbf{D}}\,(M) \equiv \textbf{if}\ \mathit{true}\,([\texttt{B!}]\mathit{true})\ ([\texttt{F!}](\lambda x.\textbf{if}\ \mathit{true}\,([\texttt{F?}]x\ [\texttt{B!}]\mathit{false})\ ([\texttt{F?}]x\ [\texttt{B!}]\mathit{false})))$$

at the type $\mathbf{D}$. The example suggests that the reduction of both checks and tags can be quite drastic when using $\mathcal{A}_{\text{CH}}$ instead of $\mathcal{A}_{\mathbf{D}}$, as was suggested in the discussion of Example 4.6.6.

6.7.2 Map

Our next example comes a little closer to "real" programs. We consider the `map`-function, defined by

```
(define map (lambda (f)
              (lambda (l)
                (if (null? l)
                    '()
                    (cons (f
                            (car l)
```

```
                                    (map f (cdr l))
                                 )
                         )
                    )
            )))
```

The algorithm produces the completion `Cmap` (we give it a new name, to distinguish)

```
(define Cmap (lambda (f)
                (lambda (l)
                        (if (null? l)
                            [nil!]'()
                            [pair!](cons (f
                                            (car [pair?]l)
                                            (Cmap f (cdr [pair?]l))
                                        )
                                    )
                        )
                )))
```

This output is produced under the assumption that `null?` has the type

$$\texttt{null?} : \forall\alpha.\alpha \to \alpha \to \texttt{bool}$$

Another possibility would be to consider the typing

$$\texttt{null?} : \forall\alpha^{(k)}\beta^{(k)}.\sum_i \alpha_i \to \sum_i \beta_i \to \texttt{bool}$$

This matter is discussed rather extensively in Chapter 7.

The example illustrates a number og points. Firstly, let us consider the inferred type for `Cmap`. One version of our implementation gives the following type information upon completing `map`:

```
****************
TOP TYPE :
LETREC
a48 = (a40->a21)
a51 = (nil+(cons a40 a51))
a50 = (nil+(cons a40 (nil+(cons a40 a51))))
a53 = (nil+(cons a21 a53))
a52 = (nil+(cons a21 (nil+(cons a21 a53))))
a49 = ((nil+(cons a40 a51))->(nil+(cons a21 a53)))
a47 = ((a40->a21)->(a50->a52))
IN
(a48->a49)
****************
```

This is not a very sophisticated presentation of the type of the completion. A good deal of work must typically be put into the presentation of types in a setting with recursive types and sums, and this has not recieved much attention in our prototype implementation. However, the type can be seen to represent the more elegant

$$\texttt{Cmap} : (\alpha \rightarrow \beta) \rightarrow (\mu\gamma.\texttt{nil} + \alpha * \gamma) \rightarrow (\mu\delta.\texttt{nil} + \beta * \delta)$$

which, under the abbreviation

$$\tau\,\texttt{list} \equiv \mu\alpha.\texttt{nil} + \tau * \alpha$$

is just

$$\texttt{Cmap} : (\alpha \rightarrow \beta) \rightarrow \alpha\,\texttt{list} \rightarrow \beta\,\texttt{list}$$

which is the "expected" type of `map`. However, this output has been produced by a version of our prototype which eliminates uninstantiated slack-variables. Without such processing of the types, the output becomes a type of the form

$$(\alpha \rightarrow \beta) \rightarrow \mu\gamma.(\ldots, \alpha, \gamma, \ldots)\texttt{dyn} \rightarrow \mu\delta.(\ldots, \beta, \delta, \ldots)\texttt{dyn}$$

where the notation $(\tau^{(k)})\texttt{dyn}$ abbreviates a maximal sum with summands $\tau_1, \ldots, \tau_k$. Dots have been placed in the types above to indicate slack variables, and the filled-in positions are, respectively, the types of the first and second component of the pair-component of the sum. The latter type is less precise, but more general. Under this typing, `(Cmap f)` can be applied to any tagged object respecting the demand that the domain type of `f` be identical to the type of the first component of the pair-component of the sum. This needs not at all be a list (but if not, then a run-time error will be the result.) This raises the question, *what do the types mean* in a framework of dynamic (or soft) typing.

6.7.3 Append

From the examples given so far, one might perhaps be inclined to think that, in general, we can choose to give, either, a very *precise* typing (like the first typing of `Cmap`), or a *universal* typing, which allows a function, say, to be used on any argument whatsoever, provided the argument is tagged. However, this is not so, since the seeming "universality" (of the second typing given to `Cmap`) is due to the "coincidence" that `map` happens, internally, to impose certain demands on its second argument. Moreover, both typings impose a specific demand on the first argument, that it should be a function of a certain type. This non-uniform character of inferred types is also present in systems of soft typing. The reason may be said to be that, for systems such as the present one and soft type systems, which type every program, there are *too few constraints* on what a type should be like. The only constraints taken into account by these systems is the *internal constraints* imposed by the program under analysis. This is just a facet of the *globality* assumption, as described in Section 6.6.1 above. The case of soft typing is discussed more extensively in Section 6.8. In the present, simple system, this can result in some quite curious effects, such as we find in the following inferred completion for the function `append`:

```
(define Cappend
        (lambda (l1)
        (lambda (l2)
                (if (null? l1)
                     l2
                    (cons (car [pair?]l1)
                          (Cappend (cdr [pair?]l1)
                                    l2))))))
```

at the type

$$\alpha\,\texttt{list} \rightarrow (\mu\beta.\alpha * \beta) \rightarrow (\mu\beta.\alpha * \beta)$$

This type is curious because the second argument is required to be a *non-empty list.* The reason is that, since `append` does not *consume* its second argument, the only constraint on the type of that argument comes from typing imprecision in the conditional; in other words, the conditional forces unification with the type of the constructed output, which will indeed always be a non-empty list. The `append` function is discussed again, in this report, in Chapter 7.

On the basis of these examples we must conclude that the *type expression* assigned to a program will not, in general, be very useful. The practical interest in a system such as the present would be to inspect the inferred *completions* for, say, debugging purposes of a single, entire program (cf. the assumption of globality, referred to above.)

6.7.4 Control flow

In the capacity as a debugging tool, the system would benefit from being extended with some form of *control flow analysis* which would detect, *e.g.*, in the `map` example, that no check `pair?` is necessary, because it can be learned from the test with `null?` that the argument (`l`) is non-empty. We might consider adapting the technique employed in soft typing [Wri94], [WrCa94] to solve this, which is to introduce pattern-matching with suitable typing rules. The negative coercions *pair*? correspond exactly to the destruction of the pair under the pattern-matching in

```
(define map (lambda (f)
              (lambda (l)
                      (match l
                         ['() => '()]
                         [(x . y) => (f x . (map f y))]))))
```

6.7.5 $\mathcal{C}_{pf}$ *vs* $\mathcal{C}_{p*}$

The algorithms described in this chapter infer completions in the class $\mathcal{C}_{pf}$ (see Section 6.4.) We round off this section by giving a small example of the relative power of the completion class $\mathcal{C}_{p*}$ as compared with $\mathcal{C}_{pf}$. A tiny but striking example is the following phrase:

```
(car (if tst true (cons false false)))
```

The $\mathcal{C}_{pf}$-completion inferred for this phrase is this

```
(car [pair?](if tst ([bool!]true) ([pair!](cons false false))))
```

This is in fact a $\mathcal{C}_{pf}(\phi)$-normal form (recall our definition of completion classes and restricted normal forms, from Chapter 5); note that the checking coercion cannot be moved into the conditional without violating the $\mathcal{C}_{pf}$-condition, since the branches of the conditional are not destruction points.

In contrast, the $\mathcal{C}_{p*}(P_\phi)$-normal form of this completion (which is also a global ϕ-normal form) is this:

```
(car (if tst ([error]true) (cons false false)))
```

The latter completion is evidently much more satisfactory that the former; or to put it more firmly, the former is almost unbearable to look at, whereas the latter is just what we want. It is not difficult to see that one can scale up this kind of example. In contrast, it appears to be very difficult to produce examples of $\mathcal{C}_{p*}$-normal forms which so clearly call for improvement. In fact, it appears difficult even to find natural examples of $\mathcal{C}_{p*}$-normal forms which are not also global ϕ-normal forms. Our experiments in this direction seem to suggest that $\mathcal{C}_{p*}$ is indeed a *very* powerful completion class (compare with our comments about this in Section 5.3.2.) Furthermore, we shall see later that the class will be important for extensions of the present framework to *modular* completion inference (see the examples considered in Section 7.1.)

6.8 Related systems. Dynamic and soft typing

This section compares our system as developed so far to the two systems of soft typing developed, respectively, by Cartwright, Fagan, Wright [Fag90], [CF91], [CF92] and Aiken, Wimmers, Lakshman [AWL94]. The discussion in this section is confined to issues which can be raised at the *core* of a dynamically typed, higher order programming language. Broader issues are discussed in Chapter 7.

6.8.1 Soft typing

Soft type inference for higher order functional programming languages has been studied and developed mainly by two teams, by Cartwright, Fagan, Wright and by Aiken, Wimmers, Lakshman. The term *soft typing* was coined at Rice by Cartwright and Fagan ([Fag90], [CF91], [CF92]) to describe a *static* type system suitable for type inference for *dynamically typed* languages, with specific reference to the language *Scheme*. While this may sound like a contradiction in terms, the overall idea, as was sketched in the introductory Chapter 1 of this report, is to devise a type inference system with the following key properties:

1. *Universality*, which means that *no syntactically correct source program is outrighly rejected* by type analysis.
   However, since an important goal is to ensure *type safety*, and since this property is clearly not decidable, a soft type inference system cannot, by logical necessity, accurately classify a program as being either type safe or type unsafe; rather, it must compute an approximation to the set of type safe programs, and any program falling in this class will be assigned a type and accepted as it is. For the rest of the programs, the soft type inference system will *insert run-time type operations* into the source program in such a way that type safety will provably be enforced. The insertion of run-time operations is clearly reminiscent of the *completion* process of dynamic typing, since we can regard a run-time operation as a *type coercion*, and the insertion of operations is performed in such a way that typability is *restored* for the "completed" program.
2. *Minimality*, which means that *run-time type operations are inserted "only" where necessary.* Clearly, the property, as stated, should be taken with a grain of salt (and hence the quotes) because, for the reasons of undecidability mentioned above, some programs must of necessity recieve run-time checks which are in fact (in the mathematical sense) not necessary, and so the statement merely expresses a property, the *approximation* of which is an important goal of soft typing.
3. *Expressiveness*, which means that *the underlying static type system must be highly expressive.* This property is interlocked with the ones already mentioned, because these other goals will expectedly be achieved proportionally, in some sense, to the expressiveness of the underlying static type system. Expressiveness must be enhanced with a view to practical programming, *i.e.*, certain forms of programming, which tend to violate ordinary static type systems, and which are regarded as especially important for practical programming, should be supported. Here, general use of non-heterogeneous structures, including recursion over such, tends to be particularly in focus; hence, the extension of Hindley-Milner polymorphism with some form of *union types* and *regular recursive types* tends to be an integral part of the desired system.

The term "soft typing" is sometimes used here in a generic sense to refer to any system which meets these goals, to an interesting degree, rather than to the specific realization of these ideas in the particular system of Cartwright, Fagan and Wright.

The Rice line of work has recently been continued by Wright ([Wri94], [WrCa94]) who takes the project from the prototype stage into a practical system which handles essentially all of IEEE *Scheme* ([CR91].) The overall technical characteristics of the resulting system fall in three categories, namely the static type system employed, the inference methods employed, and the method of completion. The main technical feature on the side of the type system is the combination of Hindley-Milner style polymorphism with regular recursive types and a form of Rémy-encoded ([Rem89]) subtyping based on discriminative union types. The encoding allows an implementation via algortihm $\mathcal{W}$ for standard Hindley-Milner type inference. The method of completion is the insertion of run-time *type checking operations* which adhere to primitive operations of the language, in the following sense: for every primitive operation f of the source language, there is an additional, checked version **check-**f; only the latter carries run-time type checks. Only when forced by, essentially, unification break-down under type inference will the system exchange an unchecked primitive by the corresponding checked one.

The soft type inference system of Aiken, Wimmers, Lakshman [AWL94] for the functional programming language FL is based on methods for solving systems of *set inclusion constraints* ([AW92], [AW93], see also [Aik94], [HeJa94]) These methods support a very impressive and rich type language, including recursive types, unrestricted (*i.e.*, non-discriminative) union types, intersection types and so-called conditional types which are used to incorporate information about control flow dependencies. Failure of typability is detected as inconsistency of constraint sets. As in the Rice-system, failure of typability leads to the insertion of run-time type checks, in the form of so-called *narrowers*. An operation $Check_\tau$, a narrower, is assumed at every monotype τ, such that $Check_\tau$ is the identity on τ and yields a run-time type error (represented by $\perp$ in the type system) elsewhere; the type of $Check_\tau$ is

$$\forall\alpha.\alpha \to (\alpha \cap \tau)$$

Comparing the Rice-system to the system of Aiken, Wimmers and Lakshman, one may say that the latter system is inherently more powerful but potentially less practicable. Set constraint solving is in general computationally very expensive. The general problem of solving systems of set constraints is complete for NEXPTIME (see [BGW93], [AKVW93]), although important restrictions, such as are used in so-called *set based analysis* [Hei94], run in polynomial time[12]; however, the set constraint solution algorithm employed by Aiken, Wimmers and Lakshman is reported (in [AWL94]) to run in worst case exponential time, and experimental evidence suggests that the method is also more expensive in practice than the Rice-system (see [WrCa94] for an assessment.) Moreover, the type expressions assigned to programs in the Aiken, Wimmers, Lakshman system can be very complicated, so that the result of analysis may be difficult to understand. Finally, the methods haven't (as yet) been carried beyond a pure functional language. On the positive side, apart from the greater formal power of the framework, the system of Aiken, Wimmers and Lakshman is "cleaner" than the Rice-system, because no type encodings are used.

[12]set based analysis runs in cubic time.

6.8.2 Similarities of dynamic and soft typing

The approach of the present work was very much inspired by the Rice-system; in fact, it could be said that we have so far attempted to unify Rice-style soft typing with a generalization of the methods of dynamic typing employed by Henglein in [Hen91], [Hen92], [Hen92a] and [Hen94]. In particular, we have incorporated the "Rice-elements" of discriminative sums and regular recursion into a polymorphic generalization of Henglein's monomorphic system of dynamic typing. The basic idea of this unification, if it may be so called, is really very simple, and in some sense quite obvious. The fundamental step is to articulate the universal type **D** as a polymorphic sum over regular recursive types.

Given this rather close connection to the Rice-system together with our desire to remain within relatively simple and computationally efficient techniques, we shall pay more attention to the Rice system than to the Aiken, Wimmers, Lakshman system in the remainder of this report. However, we begin with a general comparison of soft and dynamic typing.

It is part of the aim of the present work to bring the techniques of dynamic typing to fulfill, hopefully to an interesting extent, the overall goals of soft typing, as stated in Section 6.8.1. Thus, these goals are shared, and if we think they can be met within a framework of dynamic typing, then that framework can be seen as containing one possible way of realizing the idea of soft typing.

The fundamental, common preoccupation with the idea of blurring the distinction between static and dynamically typed languages may, potentially, be of benefit in at least the following respects:

1. *Static debugging.* Under this view, the run-time type checks (type coercions) of a completion may be interpreted as *warnings*, where a checked subterm indicates a *potential* run-time error, in case the piece of code in question gets executed. The presence of an error-coercion may indicate that, if ever executed, the piece of code in question will *definitely* generate a run-time error. Related to this view is the idea of using completions for *type error recovery* in a stacically typed language. As we have discussed earlier (Chapter 1) one may consider relaxing the interpretation of coercions here, such that they are not interpreted as safe claims about the run-time behaviour of (parts of) the program, but rather as indicators of formal type violations.

2. *Optimization.* Under this view, the analysis performed by soft and dynamic systems allow certain run-time type operations to be deemed superfluous in the sense that they can be safely eliminated. This results in optimization of target code.

3. *Bringing compilation technology developed for statically typed languages to bear on dynamically typed languages.* A completion inferred by a soft or dynamic system may in principle be regarded as a statically typed program. Taking this view, we may consider the idea of regarding source programs of a dynamically typed language as *incomplete program phrases of a statically typed language.* Completing the program can be seen as part of the compilation process, which now takes place as if within a statically typed framework. Under this view, we can consider utilizing compilation technology developed for statically typed languages, where much work is being done to take advantage of the presence of type information at compile time.

4. *High-level compilation from dynamically typed languages to statically typed languages.* In continuation of the point just made, we can go as far as to actually compile a dynamically typed language into a statically typed language. In fact, the process of completion can, in

accordance with the point mentioned above, be regarded as a special case of this. However, we can consider taking this idea further; an example of this is our development in Chapter 7, where we consider translating *Scheme* to *ML*. This appears to be an application for which dynamic typing is better fit than the extant frameworks of soft typing (see below.)

6.8.3 Non-modularity

A very important, common *limitation* of both of the soft type systems as well as of the dynamic type inference system as developed in the present chapter, is that they are all *global, non-modular* techniques in the sense made precise in Section 6.6.1 above. The reason why the assumption of globality is necessary for the Rice system can be seen from a very simple example.

Example 6.8.1 Consider the application combinator `apply`, given by

```
(define apply (lambda (f) (lambda (x) (f x))))
```

The Rice system will type this program inserting *no* checks at all, because the system is not *forced* to do so by the program itself. Our simple inference system will type the program at

$$\forall\alpha\beta.(\alpha \to \beta) \to \alpha \to \beta$$

inserting no coercions at all, since that system is conservative over Hindley-Milner typing. However, there exist contexts C for which this completion does not work in the Rice-system; this could be for the reason that the chosen *presentation type* ([WrCa94]) cannot type C[`apply`], or it could be because it would violate *type safety.* For instance, insertion of `apply` into the context $C \equiv$ ([] `1`) would lead to a *stuck state*, because the procedure application in `apply` is unchecked. To work for context C with preservation of type safety, the Rice-completion of `apply` would have to be

```
(lambda (f) (lambda (x) CHECK-ap(f, x)))
```

where `CHECK-ap` is the run-time type checked version of procedure application. □

Note that the examples shown in Section 6.6.1 will not drive home the point for the Rice-system; those examples showed specifically that insertion of *tagging* operations may be necessary to achieve safety for certain contexts; however, the Rice-system assumes all objects to be tagged at run-time (see below), and a program such as

```
(define f (lambda (x) (if P x #f)))
```

is completed (in the Rice-system) at the type [13]

$$\forall\alpha\beta.((false^{+} \cup \alpha) \to^{+} (false^{+} \cup \alpha)) \cup \beta$$

The reason for the presence of the unions with the trailing variable α here is that constants are assumed to have "universal types" of the form $\forall\alpha.\langle \mathit{basetype}\rangle \cup \alpha$, for instance, the boolean constant `#f` has the type $\forall\alpha.\mathtt{bool} \cup \alpha$.

[13]The superscripts + are so-called *flags* used to simulate a form of subtyping. We refer the reader to [WrCa94], from which the example is taken.

Example 6.8.1 shows the point for Aiken, Wimmers, Lakshman system as well. Since that system of is conservative over Hindley-Milner typing (Lemma 4.6 of [AWL94]), the system will infer the type. [14]

```
(g true)
```

In [WrCa94] the problem of modular soft type inference is proposed to be one of the major problems for future work. This problem is addressed, within the framework of dynamic typing, in Chapter 7 of this report. We argue there that our dynamic framework could be better suited than the Rice soft type inference systems to achieve the necessary balance between minimality and safety required for modularity.

6.8.4 Differences of dynamic and soft typing

At least the following features distinguish our generalized dynamic typing from both of the two soft type inference systems considered:

d1. Our system analyses both tagging and check/untagging operations and deals with optimization of both kinds of run-time type operations as dual aspects, in the style of Henglein's dynamic typing. This is in contrast to the other systems mentioned, which assume that all objects are *tagged* at run-time, and they only deal with optimization of run-time *checking* operations.

d2. Our system can be made to serve as the basis of a *Scheme*-to-*ML* translation, as is outlined in Chapter 7.

d3. Our system is modelled by a completion calculus, Σ^{∞}, in which it is possible to reason about various aspects of completion. In addition, it is possible to define extensions of the present framework on the basis of the syntactical theory of dynamic typing developed in Chapter 4. In particular, there is in principle a well-behaved notion of completion within the class $\mathcal{C}_{P*}$ of completions with unrestricted use of primitive coercions (Chapter 5).

d4. Our system allows smooth integration with *coercion paramterization and instantiation* based on a form of qualified type system.

Ad d1. With respect to the aspect of *optimization* (Section 6.8.2 above), dynamic typing is in one respect more powerful than the systems of soft typing, because it deals with the *tagging* operations in addition to the *checking* operations, as natural dual aspects. However, one must be aware here that we have so far adopted an *abstraction principle* to the effect that

- Dynamic type analysis tells us only that certain operations are not necessary *from the point of view of run-time type safety.*

[14]The reader with a detailed memory of [AWL94] may think that there is a discrepancy with what is said above about non-modularity of the soft type inference system described in that reference, because the authors write, comparing their methods to those developed within the framework of abstract interpretation:

"Our type inference algorithm, on the other hand, is a compositional, bottom-up algorithm. This makes it amenable to use in an environment that supports seperately compiled modules."

However, this should definitely just be taken to mean that the inference algorithm is compositional, and *not* that it will produce safe completions in a modular way as it stands. It might, however, be *amenable* to work for a soft type inference system which is modular. The authors have no discussion of the safety issue in [AWL94].

Regarding run-time tagging operations, it must be said that there may be *other* motivations for compiling something like type tags into the target code than those having to do with run-time type safety. An important example of this is that certain techniques for automatic *garbage collection* require the presence of type tags at run-time.

However, from the general perspective of *bringing compilation technology developed for statically typed languages to bear on dynamically typed languages*, the aspect of tagging optimization *could* be of potential benefit, since a notable amount of work is currently being done to devise efficient garbage collection techniques for languages with strong, static typing. Some of this work aims directly at designing *tag-free garbage collection* for such languages, by making the garbage collector rely on type information at run time. Such ideas have been proposed recently by Appel, [App89] and have been taken up by several others. See [Tol94] for a very recent contribution, which contains further references. Related ideas proposed by Harper and Morrisett [HM95] suggest using run-time type information to compile polymorphic languages, with the aim of choosing (at compile time) efficient data-representations; this is an issue in the presence of polymorphism which requires representation independence and which is, in the extreme, realized by completely uniform ("boxed") data-representation. See also [Ler92] [HJ94] for related analyses of "boxing". Another example of a line of research, which may make the idea of using "static techniques" in the implementation of dynamic languages intriguing, is that pursued by Tofte and Talpin about garbage collection based on *region inference* (see [TT93], [TT94].)

The idea of integrating such techniques with dynamic type analysis remains at the speculative level in this report. Further discussion of these aspects are found in Chapter 7 also. The "abstraction principle" mentioned above is discussed in more depth in Section 7.5.

Ad d2. The *Scheme*-to-*ML*-translation to be outlined in Chapter 7 is based in an essential way on the explicit modelling of both of the dual run-time type operations. We refer to Chapter 7 for further discussion of this point.

Ad d3. The coercion calculi have proven to be important in several respects. One notable instance of this is that we have been able to prove the existence and uniqueness of normal form completions in the class $\mathcal{C}_{p*}$. This result may be of considerable practical interest. We have already seen indications of this, in Section 6.7.5. In contrast, the Rice-system appears to be inherently limited to the equivalent of $\mathcal{C}_{pf}$-completions, since all run-time type checks adhere to applications of primitive operations (corresponding to primitive destruction points.) The example considered there would have to be typed with a run-time type check in the Rice system, yielding

```
    (CHECK-car (if tst true (cons false false)))
```

Note that no equivalence to the error-coercion of the $\mathcal{C}_{p*}$-completion shown in Section 6.7.5 can be introduced in the Rice-style typing, even though the Rice-system contains so-called *error-checks* (see [Wri94].) Like any other check, such checks can only be placed as a qualification of a primitive operation (`car` in the example above.) This, in turn, can only be done safely if the system can prove that the occurrence of the primitive will never be applied to a valid argument. This condition is not satisfied in the example above.

Another point, which is harder to document, yet significant, is the fact that insights obtained from consideration of the calculus have proven to be invaluable in the practical development and understanding of completion inference.

Ad d4. The introduction of coercion abstraction and instantiation is an integral part of dynamic typing for capturing **let**-polymorphism (cf. Section 6.6 above.) Moreover, it will turn out to be a very important ingredient when, in Chapter 7, we come to consider the problem of *modular* completion inference and *polymorphic safety*, which appears to require completion in the class $\mathcal{C}_{p*}$. We refer to Chapter 7 for further discussion of this point.

There appears to only one respect in which the framework developed here is less expressive than the Rice-system. This is the simulation of subtyping via so-called *flag-variables*, which has no counterpart in our system as developed here. Something like it could probably be introduced in the present framework. We do not know at this time how important the absence of this feature is. In any case, it is a somewhat vulnerable technique since it is based on the equational analysis performed by algorithm $\mathcal{W}$. For instance, in a term of the form $\lambda x.M$, the effect will be impeded at the type of x, since all occurrences of x in M must have the same type.

Chapter 7

Dynamically Typed Scheme and ML-Scheme

This chapter attempts to do two things, more or less at the same time. On the one hand, we outline how the framework of completion inference in Σ^{∞} could be extended to deal with a real dynamically typed language such as *Scheme*. This results in the definition of a completion language called *Dynamically Typed Scheme*, *DTS* for short. The intention is that this completion language could serve as a static debugging tool for the underlying dynamically typed language.

On the other hand, we outline how dynamic type analysis can be made the basis of a high-level compilation of a *Scheme*-like language to *ML*. The method is by translating the *Scheme*-language into a richer language of *Scheme*-completions, called *ML-Scheme*, *MLS* for short.

The first section continues the discussion, introduced in Chapter 6, of safe modular completion inference. We argue that a full integration of static and dynamic typing must achieve modularity and is therefore bound to work with a notion of *polymorphic safety* which is stronger than the safety conditions with respect to which systems of soft typing or the dynamic system described in Chapter 6 are correct. We define what we take to be the appropriate stronger notion of safety, and we point out that, relative to this notion, the use of *coercion parameterization* together with completion in the class $\mathcal{C}_{p*}$ are important means for achieving a satisfactory balance of minimality and safety.

The second section begins the outline of a translation from the core of *Scheme* into *SML*. It is focussed on problems of translating of type recursion, which can be raised already at this early stage. This is continued in the third section, where the language *Core-Scheme* is defined, and a translation schema for the language *Core-Scheme* is proposed. The use of this schema is dependent upon a dynamic type analysis which is not specified in this section.

The fourth section defines the completion language *Dynamically Typed Scheme*. This is intended to serve as the target language for dynamic type analysis of *Core-Scheme*. The section identifies and discusses what we believe are the major problems posed at the core level. These are assignments, polymorphism in the presence of assignments and the handling of lists. We propose to adopt Wright's *imperative polymorphism*, at least for a prototyping stage of the project. We introduce the problem of *lists* which will be discussed recurrently throughout the chapter. We propose to use so-called *box*-types to handle assignments correctly. Finally, we introduce the topic of *universally defined operations* in *Scheme*, using the example of *Scheme*'s conditional construct, which is present in the core. This topic is also discussed recurrently throughout the chapter.

The fifth section introduces the *abstraction principle*. This was briefly referred to earlier, Section 6.8.4; it is more thoroughly defined and motivated in the section here. In principle, one could imagine both enterprises of dynamic type analysis and *ML*-translation as taking place within one and the same framework (using only one completion language.) This question is the topic of several discussions in this chapter. For reasons of simplicity, we adopt here the abstraction principle methodologically. This principle states that dynamic type analysis should model only the aspects of run-time type safety. The main question raised is whether or not run-time type tags should be used to implement non-parametric, generic *Scheme*-functions, of which the universally defined operations are a family of examples. This leads to the strategy of separating, in principle, the enterprise of dynamic type analysis and high-level translation and leads to the use of two different, though very closely related, completion languages.

The decisions in section five are reflected by the definition, in the sixth section, of the completion language *ML-Scheme*. At the core level, this language differs from *DTS* only in its typing of the conditional. However, the main importance lies in the general principle underlying this difference. We discuss a major alternative strategy, which is to rely on run-time type information rather than coercions. The alternative may well turn out to be more attractive in the end, but for the sake of simplicity we do not rely on it here. The problems of lists and boxes are briefly discussed wrt. translation.

The seventh section discusses how the ideas presented in the chapter could be extended to handle something like the full *Scheme*-language. This section reads as a problem-list, since we focus on what we take to be the main obstacles, which a full analysis and translation will meet. We identify features of *Scheme* which we believe cannot in a satisfactory way or should not be handled at all.

An overall outcome of our development is that, while a translation to *ML* appears to be possible in principle for a substantial part of *Scheme*, more sophisticated methods than those uniformly applied here will be necessary for an interesting translation in practice.

7.1 Polymorphism, safety and modularity

A dynamically typed language like *Scheme* offers the possibility of defining highly *generic* functions. A *Scheme*-procedure will work on any input whatsoever, unless it generates a run-time error. Such errors are only generated where absolutely necessary. *Scheme*-procedures can therefore be written which work on a wide range of different objects. This feature is useful for *code reuse*, since a single piece of multi-purpose code can be stuck away in a program library and called upon in a potentially unlimited number of contexts of use. This is in strong contrast to some statically typed languages, whose type discipline is so tight that code reuse is impeded. In *Pascal*, for instance, we have to declare an identity function at each type, even though they all perform the same abstract *operation*, uniformly over all inputs. The advent of *polymorphism* has challenged the idea that genericity must be bargained for static type safety. The image of a polymorphically typed function stuck away in a library captures much of the essence of what is important about polymorphism: once written, the function is there, and we never have to write code sufficiently “like it” again, (very) ideally like a theorem of mathematics. Given its type scheme, we may often allow ourselves to forget about the internals of the function, since its type specifies its interface to the outside world, determining the set of possible contexts of use.

When we move to “mixed” systems, like soft type systems or polymorphic dynamic typing, the nature of polymorphism changes. In its purest form, as is found in the second order λ-calculus, system **F** [GLT89], polymorphism is *parametric* ([Rey83].) One aspect of this deep property is that a parametric function is invariant under representation shifts and works *uni-*

formly over all objects for which it is type correctly defined. In the presence of dynamic type coercions, general parametricity brakes down. One indication of this is that we can define a fixed point combinator (see [Wad89].)

We have seen, in Section 6.6.1 and Section 6.8, that, due to restrictions of *safety*, *modularity* holds neither for the systems of soft typing nor for the dynamic completion inference described in Chapter 6. This is a serious drawback, since it runs counter to the ideal of polymorphism outlined above. In a non-modular setting, we cannot infer a (soft or dynamic) type for a function and put it away in a library. This also conflicts with the spirit of a dynamically typed language, which by its very nature supports a maximum of modularity. In a sense, therefore, if we cannot achieve modularity of completion inference, then we cannot really claim to have unified dynamic and static typing in a single framework.

The cause of failure of modularity being restrictions of safety (see Section 6.6.1), let us consider what a proper notion of safety would be for a modular setting:

Definition 7.1.1 (*Polymorphic safety*)
Let M be a completion at a *monomorphic* type τ. We say that M is *safe at* τ, if and only if M is *operationally safe*, according to Definition 1.6.6, *i.e.*, for every completion M', $M \cong M'$ implies $M \sqsubseteq M'$, so that for every program context C into which both M and M' can be type-correctly inserted, we have

$$C[M] \Downarrow^v \varepsilon \;\Rightarrow\; C[M'] \Downarrow^v \varepsilon$$

An *arbitrary* completion M is said to be *adaptible* to a mono-type τ, if there is a coercion c such that $[c]\tilde{M}$ has type τ, where $\tilde{M}$ arises from M by a series of coercion abstractions and instantiations.

We say that a completion M at an arbitrary type is *polymorphically safe* if M is adaptible to a safe completion at *every* monotype. □

Polymorphic safety is a very strong condition, since we quantify over *all* possible contexts of use, requiring safety at all types. This notion is therefore suitable for a setting in which modularity must hold. The question is whether it is realistic to imagine an interesting completion inference system, which will invariably infer polymorphically safe completions. Here we may at once note that it is trivially the case that polymorphically safe completions always exist. We can simply insert a fresh coercion parameter at every coercion point in the term and abstract over all parameters. Such a completion is of course not interesting in general, since its type is going to be most unrevealing and the completion completely inefficient. The problem of inference is therefore to achieve polymorphic safety while still *minimizing* the use of coercions as much as possible. This is more of an art, since safety and minimality are sometimes conflicting goals, as we saw in Section 6.6.1.

The main technical innovation of polymorphic dynamic typing for dealing with this problem is the use of coercion parameterization, as was suggested in [Hen92] and illustrated by examples in Section 6.6.1. In addition to this, the inherent flexibility of dynamic typing, where coercions can in principle be placed anywhere in a term (under safety restrictions), is of potentially great importance. In a nutshell, the problem af minimization under polymorphically safe inference consists in trying to do just the following:

- *postpone as many coercions as possible to the contexts of use without compromising safety*

Now, if we can discover that, say, a function will always consume an argument at a certain type, then no harm can be done in requiring that type of the input to the function in a polymorphic completion. To illustrate, consider the simple *Scheme*-program

```
(define f (lambda (x) (x #t)))
```

A polymorphically safe completion of this program is itself, completed with no coercions at all, at type $\forall\alpha.(\texttt{bool} \to \alpha) \to \alpha$. The reason why this is safe is that this combinator will always consume a function of booleans, *i.e.*, it will generate a run-time error, unless the first argument is a function of booleans. However, if we put the function `f` away in a program library as it stands, then we need to be able to coerce the *argument* at the point of application, as in the context $C \equiv$ `([] 1)` for which `f` can be extended to a safe completion of C[`f`], namely

```
 ([f] [func?][num!]1)
```

This is so, because it lies in the discipline of modular inference, that the *only* allowed modification of a completed program is *adaptation* (cf, Definition 7.1.1) at the point of use. If we could not coerce negatively at the argument, as above, then there would be no way that the completion `f` could be extended to a completion of C[`f`]; in other words, there would be a context and a type ($\texttt{num} \to \alpha$) for which no adaptation existed.

The conclusion to draw from this is that, in order to achieve full modularity via coercion parameterization, we must develop a form of *neededness-analysis* to minimize coercion parameters. Such analysis would have to restrict unification of types to situations where it is provably the case that any use of the completed object will be consumed at the type given to it.

Note that the laziness inherent in programming languages where evaluation is to head-normal form may render parameters necessary, even though evaluation in every *ground* context consumes a certain type. An example is `apply`,

```
(define apply (lambda (f) (lambda (x) (f x))))
```

In every ground context, this program will consume a function in its first argument, and still it is unsafe for a context such as C: since C[`apply`] would have to be completed as

```
([apply] [func?][num!]1))
```

which gives a run-time type error (we could have used an error coercion instead of `func?`∘`num!`), whereas the completion

```
(define apply' (lambda (f) (lambda (x) ([func?]f x))))
```

can be extended to a completion of C[`apply'`]: which does not result in a run-time error, witness

```
(apply' [num!]1)
```

Therefore, `apply` cannot be safely completed at the "pure type"

$$\forall\alpha\beta.(\alpha \to \beta) \to \alpha \to \beta$$

but rather, we must complete at the *coercive type*

$$\forall\alpha\beta\gamma.(\alpha \rightsquigarrow (\beta \to \gamma)) \Rightarrow \alpha \to (\beta \to \gamma)$$

corresponding to the parameterized completion

```
(define parameterized_apply
        (Lambda (k : 'a --> ('b -> 'c))
                (lambda (f : 'a)
                (lambda (x : 'b)
                        ([k]f x)))))
```

In a "good" context this can the be instantiated to its pure type, as in

```
(parameterized_apply{id} (lambda (x) x))
```

and in a "bad" context we can instantiate with a negative coercion

```
parameterized_apply{func?}
```

at

$$\forall\alpha\beta\ldots\texttt{(}\ldots,\texttt{number},\ldots\texttt{)dyn}\rightarrow\alpha\rightarrow\beta$$

which yields a good completion of the "bad" application

```
(parameterized_apply{func?} [num!]1)
```

Note that these complications also arise in a call-by-name language with eager basic operations. For instance, in such a language, one cannot complete

```
(lambda (x) (if tst (car x) 1))
```

at the "pure type"

$$\forall\alpha.\texttt{number}*\alpha\rightarrow\texttt{number}$$

since this would not be safe (under CBN-semantics) for the context (verification left for the reader)

```
(+ 2 ([] 3))
```

It is also important to realize that the completion of $C[\texttt{f}]$,

```
(f [func?][num!]1)
```

is *not* in the class Σ^{∞}_{pf} of completions with coercions at fixed positions, since the negative `func?` is not at a destruction point. It is, however, in the class Σ^{∞}_{p*}, and the example indicates (by generalization) that the class Σ^{∞}_{p*} is potentially quite important for minimization in a modular setting. This also shows that the present framework is in this respect stronger than the soft type system in [Wri94]. For in that framework, one would have to complete `f` as

```
(define f (lambda (x) CHECK-ap(x, #t)))
```

with a run-time checked version of procedure application, just because run-time checks adhere to points of application of the primitive operations of the language. Compare our discussion in Section 6.8.

7.2 Translating Σ^{μ} into ML

A translation of Σ^{μ}-completions into the *core* of *SML* (see [MTH90] for the definition of the core) is outlined in Figures 7.1, 7.2, 7.3 and 7.4. We consider a minimal, kernel source language with λ-abstraction, application, boolean constants, pairing and conditional. The type language associated with this language of completions is the obvious one, generated from the constant $\mathbf{B}$, the function space constructor $\rightarrow$ and the pairing constructor $*$.

Figure 7.1 describes a set of standard declarations, relative to which the translation takes place. Like any other declarations mentioned in the translation, these declarations are to be taken as global. First, the polymorphic dynamic type associated with the type language of our source language is declared. Note that this is not a universal type, since the universal type recursion is not included (and if it were, the declared type would no longer be polymorphic.) The type-constructors (injections) of the *SML*-sum type will translate the non-trivial, positive (tagging) coercions. Next, an exception `TypeError` is declared, to be used in the translation of error-coercions. Third, a series of declarations define the non-trivial, negative (check-and-untag) coercions, by pattern matching on the corresponding injections. [1]

Figure 7.2 defines a translation on source types to *SML*-types. In the case of the sum-type, we assume that the index tc ranges through the constructor alphabet in the order $\texttt{bool}, \rightarrow, *$. The translated form

$$(\lceil \tau_{tc_1} \rceil, \ldots, \lceil \tau_{tc_k} \rceil)_{tc}\texttt{dyn}$$

denotes the instance of `dyn`, where the translations of the argument types $\overline{\tau}$ are plugged in at the appropriate positions. In case a (slack) variable occurs in top-level position within the sum, the translation can be understood to generate fresh type variables. For instance, we have

$$\lceil \alpha + \beta \rightarrow \gamma + \delta \rceil \equiv (\alpha, \beta, \delta_1, \delta_2)\texttt{dyn}$$

where fresh variables δ_1, δ_2 are generated for δ. Also, we have

$$\lceil \texttt{bool} + (\alpha \rightarrow \beta) + (\gamma \rightarrow \delta) * (\texttt{bool} * \texttt{bool}) \rceil \equiv (\alpha, \beta, \gamma \rightarrow \delta, \texttt{bool} * \texttt{bool})\texttt{dyn}$$

In the case of recursion, the translated form

$$\langle \textit{type expression} \rangle \textit{ with declaration } \langle \textit{declaration} \rangle$$

indicates that the translation of the recursive type is ⟨*type expression*⟩ relative to the declaration ⟨*declaration*⟩. The declaration may, for simplicity, be assumed to be global for the whole translation. Note that we index the type constructor $\texttt{recty}_\tau$ with τ, and also the injector $\texttt{inrec}_\tau$; this is to indicate that, for different types, we have to choose different type- and constructor names in the *SML*-program. Finally, if $V = \{\alpha_1, \ldots, \alpha_n\}$ is a set of type variables, we use the notation V `type-name` as a short-hand for $(\alpha_1, \ldots, \alpha_n)\texttt{type-name}$. As an example we have

$$\lceil \mu\alpha.\alpha \rightarrow \beta \rceil \equiv \beta\,\texttt{recty} \rightarrow \beta$$

with the declaration

$$\texttt{datatype}\ \beta\,\texttt{recty = inrec of}\ \ \beta\,\texttt{recty} \rightarrow \beta$$

which provides us with an injection `inrec` at the type

$$(\beta\,\texttt{recty} \rightarrow \beta) \rightarrow \beta\,\texttt{recty}$$

Consider also the universal type

$$\mathbf{D}^\mu \equiv \mu\alpha.\mathbf{B} + (\alpha \rightarrow \alpha) + (\alpha * \alpha)$$

[1]In principle, we could leave out the default cases in the definitions of the checking coercions and also leave out the declaration of the error exception, relying instead on the automatically raised exception `Match` of the *SML*-system. However, it is more general, and more perspicuous, to include these things explicitly. In particular, one could imagine parameterizing the error-exception with error-messages, and one could also consider introducing various error-handling routines.

for which we have

$$\lceil \mathbf{D}^{\mu} \rceil \equiv \texttt{(recty, recty, recty, recty) dyn}$$

with the declaration

```
datatype recty = inrec of (recty, recty, recty, recty) dyn
```

providing us with an injection `inrec` at the type

$$\texttt{inrec} : \texttt{(recty, recty, recty, recty) dyn} \rightarrow \texttt{recty}$$

This defines `recty` as the monomorphic universal type; for instance, composing `inrec` with `in_func` : $(\alpha \rightarrow \beta) \rightarrow (\alpha, \beta, \gamma, \delta)$`dyn` will give us back the monomorphic function tagging coercion,

$$\texttt{inrec} \circ \texttt{in\_func} : (\texttt{recty} \rightarrow \texttt{recty}) \rightarrow \texttt{recty}$$

and similarly for the other coercions.

Figure 7.3 defines the translation on coercions. Note the use of the type subscript at the translation of the type recursion coercions; we have

$\mathbf{f}_{\tau} : \tau\{\alpha := \mu\alpha.\tau\} \rightsquigarrow \mu\alpha.\tau$

$\mathbf{u}_{\tau} : \mu\alpha.\tau \rightsquigarrow \tau\{\alpha := \mu\alpha.\tau\}$

Also, note translation of the error-coercion $\updownarrow$,

$$\texttt{fn x} \Rightarrow \texttt{raise TypeError}$$

at the type $\forall\alpha\beta.\alpha \rightarrow \beta$. The translation of sum-coercions $\sum_i c_i$ uses the short-hand notation

$$\{\texttt{in}_i\, y \;\Rightarrow\; \texttt{in}_i(f\ y)\}_i$$

where $\texttt{in}_i$ should be taken to denote the sequence of standard injections, listed in order, yielding the full case-analysis

```
  in_bool y => in_bool (f y)
| in_func y => in_func (f y)
| in_pair y => in_pair (f y)
```

Finally, Figure 7.4 defines the translation on completions, which relies on the coercion translation.

As the reader may have observed already, our translation is only defined on a slightly restricted set of completions. Firstly, we have everywhere assumed sums (of types and coercions) to be *maximal*, *i.e.*, containing a summand for every type constructor (possibly represented by a *slack variable*.) This is what makes it possible to use the one standard polymorphic dynamic type $(\alpha, \beta, \gamma, \delta)$`dyn` to translate the sums. Secondly, we are assuming that every occurrence of the error-coercions has the form $\updownarrow$; in other words, the error-coercions always occur in composition. A completion satisfying these conditions will be said to be in *standard form.*

It is easy to verify, by structural induction on c, that we have

Lemma 7.2.1 *If* $\vdash_{\Sigma^{\mu}} c : \tau \rightsquigarrow \theta$ *then* $\vdash_{ML} \lceil c \rceil : \lceil \tau \rceil \rightarrow \lceil \theta \rceil$

Using this property, it is also straight-forward to verify that we have

Proposition 7.2.2 *If* M *is a completion in standard form, then* $[\![M]\!]$ *is defined, and* $\Gamma \vdash_{\Sigma^{\mu}} M : \tau$ *implies* $\Gamma \vdash_{ML} [\![M]\!] : \lceil \tau \rceil$

Type recursion

There is one issue of efficiency which can be raised already at this early stage. This is the translation of type recursion, where the translation uniformly uses the `recty`-declarations with special recursion injections `inrec` and projections `outrec`. This may not be an optimal way to proceed, because it is sometimes possible to get rid of the recursion coercions by utilizing the guarded recursion provided by ML's `datatype` facility. For instance, the type $\mathbf{D}^{\mu}$ considered above could alternatively be translated directly to the universal datatype `univ` given by

```
datatype univ = inuniv_bool of bool
              | inuniv_func of univ -> univ
              | inuniv_pair of univ * univ
```

Given this definition, we can, say, inject into the universal type by a single primitive coercion, whereas before we had to use the composition of two coercions. The opportunity to optimize the data-representation in this way is present in principle for all types of the form $\mu\alpha_1 \ldots \alpha_n . \sum_i \tau_i$, where recursion sits immediately over the sum.

```
datatype ('a1,'a2,'a3,'a4) dyn = in_bool of bool
                               | in_func of 'a1 -> 'a2
                               | in_pair of 'a3 * 'a4

exception TypeError;

fun out_bool x => case x of
                     in_bool b => b
                 |   _         => raise TypeError

fun out_func x => case x of
                     in_func f => f
                 |   _         => raise TypeError

fun out_pair x => case x of
                     in_pair p => p
                 |   _         => raise TypeError

fun      id x => x
```

Figure 7.1: Standard declarations

$$
\begin{array}{rcl}
\lceil \alpha \rceil & \equiv & \alpha \\[1ex]
\lceil \mathbf{B} \rceil & \equiv & \texttt{bool} \\[1ex]
\lceil \tau \to \tau' \rceil & \equiv & \lceil \tau \rceil \to \lceil \tau' \rceil \\[1ex]
\lceil \tau * \tau' \rceil & \equiv & \lceil \tau \rceil * \lceil \tau' \rceil \\[1ex]
\lceil \sum_{tc} tc(\overline{\tau}) \rceil & \equiv & (\lceil \tau_{tc_1} \rceil, \ldots, \lceil \tau_{tc_k} \rceil)_{tc}\texttt{dyn} \\[1ex]
\lceil \mu\alpha.\tau \rceil & \equiv & \lceil \tau \rceil \{\alpha := V\,\texttt{recty}_\tau\} \\[1ex]
 & & \textit{with declaration} \\[1ex]
 & & \texttt{datatype}\ \ V\,\texttt{recty}_\tau\ \ \texttt{=}\ \ \texttt{inrec}_\tau\ \ \texttt{of}\ \ \lceil \tau \rceil \{\alpha := V\,\texttt{recty}_\tau\} \\[1ex]
 & & \textit{where}\ \ V = FV(\tau) \setminus \{\alpha\}
\end{array}
$$

Figure 7.2: Translation on types

$$
\begin{array}{rcl}
\lceil id \rceil & \equiv & \mathtt{id} \\
\lceil \mathrm{B!} \rceil & \equiv & \mathtt{in\_bool} \\
\lceil \mathrm{B?} \rceil & \equiv & \mathtt{out\_bool} \\
\lceil \mathrm{F!} \rceil & \equiv & \mathtt{in\_func} \\
\lceil \mathrm{F?} \rceil & \equiv & \mathtt{out\_func} \\
\lceil \mathrm{P!} \rceil & \equiv & \mathtt{in\_pair} \\
\lceil \mathrm{P?} \rceil & \equiv & \mathtt{out\_pair} \\
\lceil \mathbf{f}_\tau \rceil & \equiv & \mathtt{inrec}_\tau \\
\lceil \mathbf{u}_\tau \rceil & \equiv & \mathtt{fn}\,(\mathtt{inrec}_\tau\, y) \Rightarrow y \\
\lceil \updownarrow \rceil & \equiv & \mathtt{fn\ x} \Rightarrow \mathtt{raise\ TypeError} \\
\lceil c \circ d \rceil & \equiv & \mathtt{fn\ x} \Rightarrow \lceil c \rceil(\lceil d \rceil \mathtt{x}) \\
\lceil c \rightarrow d \rceil & \equiv & \mathtt{fn\ f} \Rightarrow \mathtt{fn\ x} \Rightarrow \lceil d \rceil(\mathtt{f}\ (\lceil c \rceil \mathtt{x})) \\
\lceil c * d \rceil & \equiv & \mathtt{fn\ x} \Rightarrow (\lceil c \rceil(\mathtt{\#1\ x}), \lceil d \rceil(\mathtt{\#2\ x})) \\
\lceil \sum_i c_i \rceil & \equiv & \mathtt{fn\ x} \Rightarrow \mathtt{case\ x\ of}\ \{\mathtt{in}_i\, y \Rightarrow \mathtt{in}_i\, (\lceil c_i \rceil\, y)\}_i
\end{array}
$$

Figure 7.3: Translation on coercions

$$
\begin{array}{rcl}
\llbracket \chi \rrbracket & \equiv & \chi \text{ where } \chi \text{ is a variable or a constant} \\
\llbracket \lambda x.M \rrbracket & \equiv & \mathtt{fn\ x} \Rightarrow \llbracket M \rrbracket \\
\llbracket M\ N \rrbracket & \equiv & \llbracket M \rrbracket\, \llbracket N \rrbracket \\
\llbracket (\mathbf{if}\ M\ N\ P) \rrbracket & \equiv & \mathtt{if}\ \llbracket M \rrbracket\ \mathtt{then}\ \llbracket N \rrbracket\ \mathtt{else}\ \llbracket P \rrbracket \\
\llbracket \langle M, N \rangle \rrbracket & \equiv & (\llbracket M \rrbracket, \llbracket N \rrbracket) \\
\llbracket \pi_i\, M \rrbracket & \equiv & \mathtt{\#}i\, \llbracket M \rrbracket \\
\llbracket [c] M \rrbracket & \equiv & \lceil c \rceil\, \llbracket M \rrbracket
\end{array}
$$

Figure 7.4: Translation on completions

7.3 Translating *Core-Scheme* to *SML*

In this section we describe a translation of a core subset of IEEE *Scheme* ([CR91]) into *SML* ([MTH90], [MTH91].) Note that the specification given here is *not* by itself sufficient to determine how any legal *Scheme*-expression belonging to the subset considered can be faithfully translated to *ML*. For this to be possible, one must assume a suitable language of coercions and completions, together with a way of translating the language of coercions. We discuss the general features of these in below as well as in Section 7.4.) It is yet another (and far from trivial) problem, how one can *infer* an *ML*-translation from a *Scheme* expression. For this to be possible, one must assume a *completion inference mechanism* which will take a *Scheme* expression to a suitable completion. So, in a sense, what we present here is the *last* step in a process which takes a *Scheme* expression to an equivalent *ML* expression, since the process would work by *first* analysing the *Scheme* expression, then completing it into a richer language according to the analysis, and then finally translating the inferred completion.

The translations given here are completely uniform and therefore simple-minded and inefficient at times. Major efficiency issues are mentioned in our commentaries as we go along.

7.3.1 Syntax

The Syntax of the *Core-Scheme* language considered here is defined in Figure 7.5. *Scheme* is a stratified language with a well defined core of elementary expression forms from which the remaining *Scheme* constructions can be derived [2]

The language considered here is, with a few exceptions, the set of IEEE *Scheme* ⟨*expression*⟩s without the ⟨*derived expression*⟩s[3] In the definition of a ⟨*datum*⟩ we have left out the ⟨*abbreviation*⟩s which logically belong together with *Scheme*'s ⟨*quasiquotations*⟩ which are not considered here. We take the category ⟨*body*⟩ (used, *e.g.*, for defining procedure bodies) to be just ⟨*expression*⟩, which is a slight (and not very essential) simplification of the definition. Moreover, we have included forms of ⟨*definition*⟩, although we have left out the definition sequences which are initiated with the keyword `begin`. Finally, we have included the `let`-construct, in a slightly simplified form, where the ⟨*binding spec*⟩ is just a simple binding, and where the ⟨*body*⟩ of the `let` is again just an ⟨*expression*⟩. It is essential that this construct is *not* taken as derived, because we wish to perform polymorphic completion inference over the set of expressions. The standard derivation of

```
(let (x e) e')
```

as

```
((lambda (x) e') e)
```

is of course no longer valid in a polymorphically typed language which uses Milner's **let**-rule.

7.3.2 *Scheme* datum

Figure 7.6 defines a translation function, $\mathcal{D}$, which sends a *Scheme* ⟨*datum*⟩ to an *ML*-expression. Our translation is parameterized on translation functions $\mathcal{N}$, $\mathcal{C}$, $\mathcal{S}$, $\mathcal{I}$ for, respectively, ⟨*number*⟩s,

[2]See section 7.3 of [CR91] for a translation of the derived expression forms.

[3]See [CR91], in particular section 7.1 for the formal definition of the syntax of *Scheme*. We refer to the syntactic categories of that definition by ⟨*category name*⟩.

⟨*character*⟩s, ⟨*string*⟩s and ⟨*identifier*⟩s. We leave these functions unspecified in the present work. We shall assume here for simplicity that all these categories, except ⟨*number*⟩s, are translated to *ML*-strings, and that ⟨*number*⟩s are restricted to integers, represented by *ML* integers.

A ⟨*list*⟩ is translated into a pair, using the unit element `()` for the empty list (`'()` in IEEE *Scheme*.) The translation of lists is simplistic, because we should really want to use *ML*-lists in may cases. See Sectionsubsec:type-system for further discussion of this.

A ⟨*vector*⟩ constant is translated to the creation of an *ML* array.

7.3.3 *Scheme* expressions

The translation of *Scheme* ⟨*expression*⟩s is defined in Figure 7.7. Here we assume, for simplicity, that a ⟨*variable*⟩ is translated to itself (this is not completely realistic.)

⟨*procedure call*⟩

A ⟨*procedure call*⟩ is translated to the application of the translation of the operator to the representation of a ⟨*list*⟩ of arguments. This corresponds to the ⟨*list*⟩ pattern matching forms of *Scheme*'s ⟨*lambda expression*⟩s, which are also translated according to this discipline, as shown below. The translation of procedure calls id further discussed in Section 7.4.

⟨*lambda expression*⟩

The complication of *Scheme*'s `lambda` expressions consist in the list pattern matching forms of formal parameter specifications. Every *Scheme* procedure can be thought of as being supplied with an argument ⟨*list*⟩ at procedure application. To take a somewhat extreme example, consider the *Scheme* expression

```
((lambda x x))
```

Its value is `()`, because x gets bound to the ⟨*list*⟩ of arguments which, in this case, is the empty list, `()`. This procedure application gets translated to the *ML*-application

```
(fn x => x) ()
```

which evaluates to `()`, as is correct. On the other hand,

```
((lambda (x) x) #t)
```

gets translated to

```
(fn (x, ()) => x) (true, ())
```

which evaluates to `true`, as expected. Similarly, the trailing variable y in the third parameter pattern form $(x_1\,x_2 \ldots x_n \,.\, y)$ will get bound to the representation of a trailing argument ⟨*list*⟩ at application, via *ML* pattern matching. Note that the translation of an abstraction may be "more defined" than the source term. This is the case for the first abstraction form. For instance, the abstraction

```
(lambda x (x #t))
```

gets translated to

```
fn x => (x (true, ()))
```

which is defined for some inputs, such as, *e.g.*, `fn x => x`; however, the *Scheme* function cannot be applied to any terminating argument without generating a run-time error. Similar considerations hold for the trailing argument of the third abstraction form.

If a completely faithful translation is to be obtained, then a completion must enforce the restriction that variable parameters and trailing parameters are assumed to be list-valued. See Section 7.4 for further discussion.

⟨*conditional*⟩

The ⟨*conditional*⟩ takes two forms. The first, "unsafe" variant with no computation specified for the case where the test is false, gets translated according to the taste of the present author, since the definition of IEEE *Scheme* leaves it unspecified [4] what should happen in case the test is true. As can be seen, we have opted for punishing the programmer in this case. Note that the test expressions are assumed to be injected into the type `dyn`; this is because the *Scheme* conditional *universal* in the test, in the sense that it is defined for any test expression. Every value different from the constant `#f` counts as a "true value" which selects the first branch of the conditional [5] The implementation of universality of the conditional test via injection into the sum `dyn` is not quite satisfactory, and we may be able to do better than that. See Section 7.4 for further discussion of this.

⟨*let expression*⟩

The main problem for translation of `let`-expressions has to do with the restrictions of *ML*'s recursive *valbind* form, `let val rec`.... This problem is discussed below, under ⟨*definition*⟩s where the same problem occurs.

An extension to the translation of `let`-expressions may be considered. As we shall see later (Section 7.4, and see also the remarks on ⟨*assignment*⟩s below), there are good reasons for considering a simplification of imperative `let`-polymorphism in the present framework, as compared with either the standard [MTH90] or the *SML of New Jearsey* system (see, *e.g.*, [MLNJ93], [HMV93].) A natural restriction has been studied recently by Wright [Wri94a]. This restriction assigns a polymorphic type scheme to x in

```
(let (x e) e')
```

only if the `let`-bound expression e is a syntactic value. Under this restriction, the standard derivation of `let`-expressions is sound for expressions where e is *not* a value; so we might consider adding

$$\mathcal{E}[\![\texttt{(let (}x\;e\texttt{)}\;e'\texttt{)}]\!] \equiv \texttt{(fn}\;x\;\texttt{=>}\;\mathcal{E}[\![e]\!]\texttt{)}\;\mathcal{E}[\![e']\!]$$

for the cases where e is not a value.

[4]Note that "unspecified" means that the definition does not make any specific requirements as to the behaviour of an implementation of *Scheme*. It does not mean that the definition does not have anything to say about the situation.

[5]Note that the standard no longer considers `()` a "false value", contrary to what is the case in many implementations.

⟨*assignment*⟩

The ⟨*assignment*⟩ naturally gets translated via *ML references*. However, the assignment presents by far the most complicated new problem in the step from the toy-language of Σ^∞ to *Core-Scheme*, as regards the problem of completion. The use of references must be explicitly enforced by the completion. For instance, one can write in *Scheme*

```
((lambda (x) (set! x #t)) (lambda (y) y))
```

which gets translated into untypable non-sense if our translation function is applied to this program directly. A faithful completion must enforce something like

```
(fn (x, ()) => x := (in_bool true)) ((ref (in_func (fn (y, ()) => y))),
                                      ())
```

A possibility here is to think of this as the translation of

```
((lambda (x) (set! x  [B!]#t)) [box][F!](lambda (y) y))
```

where a new coercion **box** is used, and which has the operational effect of creating a reference, so we can imagine a translation which has $\lceil \textbf{box} \rceil \equiv$ `ref`. Note that the injections by `in_func` (F!) and `in_bool` (B!) are necessary, since x will get *ML* type τ `ref` where τ must be a common supertype of the type of `true` and the translated identity function.

The problem of constructing a discipline of completion for assignments turns out to be a major problem. See Section 7.4.

⟨*definition*⟩

Finally, ⟨*definitions*⟩ are translated into *ML* declarations with *valbinds*[6] The first, non-parameterized ⟨*definition*⟩ form

`(define` $x\ e$`)`

presents the second major difficulty, and it does not recieve a completely faithful translation here, since *ML* cannot express[7] exactly this construct. The reason is that *Scheme* ⟨*definitions*⟩ can be recursive, only with the restriction that it must be possible to evaluate the body of the definition without assigning or referring to the bound variable. In *ML*, on the other hand, the recursive *valbindings* require the body of the binding to be a syntactic abstraction of the form `fn` *match*[8] Note that the idea of using a form of *forcing* will not work, as in the attempt

$$\mathcal{P}[\![\texttt{(define } x\ e\texttt{)}]\!] \equiv \texttt{let fun } x\texttt{() = } \mathcal{E}[\![e]\!]\{x \mapsto x\texttt{()}\} \texttt{ in } x\texttt{() end}$$

since this form will re-evaluate x at each point of reference, which, apart from being inefficient, is not sound in a language with side-effects. In general, though, the restriction on *Scheme* ⟨*definition*⟩s mentioned above will typically mean that some form of delaying is necessary in the body of the definition, to protect the defined variable from being referenced during evaluation of the definition; and this, in turn, will typically be enforced by an abstraction. But still, we cannot translate a combinator such as

```
(define f ((lambda (x) (lambda (y) f)) #t))
```

[6]See [MTH90] for the formal syntax of *ML* and the syntactic categories it mentions.

[7]In the technical sense of strong, local expressibility, as in [Fel91]

[8]See [MTH90], p. 9, with the side condition on value bindings of the form *pat* = *exp* within `rec`.

alone because the body is not an abstraction. However, this happens to be a contrived way of defining the function [9]

```
(define YK (lambda (x) YK))
```

We can translate this latter definition, presupposing (as always) some suitable completion. For instance, the definition just shown could be completed and translated into

```
val rec YK = fn x => inrec(in_func YK)
```

at the type `recty` $\rightarrow$ `recty`.

[9]The function `YK` simulates the λ-calculus "black-hole" combinator **YK** which has the interesting property that $\mathbf{YK}\ \vec{N} =_\beta \mathbf{YK}$ for every argument vector $\vec{N}$.

```
⟨expression⟩          ::=  ⟨variable⟩
                        |  ⟨literal⟩
                        |  ⟨procedure call⟩
                        |  ⟨lambda expression⟩
                        |  ⟨conditional⟩
                        |  ⟨assignment⟩

⟨literal⟩             ::=  ' ⟨datum⟩ | ⟨boolean⟩ | ⟨number⟩ | ⟨character⟩ | ⟨string⟩

⟨procedure call⟩      ::=  (⟨expression⟩ ⟨expression⟩*)

⟨lambda expression⟩   ::=  (lambda ⟨variable⟩ ⟨expression⟩)
                        |  (lambda (⟨variable⟩*) ⟨expression⟩)
                        |  (lambda (⟨variable⟩+.⟨variable⟩) ⟨expression⟩)

⟨conditional⟩         ::=  (if ⟨expression⟩ ⟨expression⟩)
                        |  (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)

⟨let expression⟩      ::=  (let (⟨variable⟩ ⟨expression⟩) ⟨expression⟩)

⟨assignment⟩          ::=  (set! ⟨variable⟩ ⟨expression⟩)

⟨datum⟩               ::=  ⟨boolean⟩ | ⟨number⟩ | ⟨character⟩ | ⟨string⟩ | ⟨identifier⟩
                        |  ⟨list⟩ | ⟨vector⟩

⟨list⟩                ::=  (⟨datum⟩*) | (⟨datum⟩+.⟨datum⟩)

⟨vector⟩              ::=  #(⟨datum⟩*)

⟨definition⟩           :=  (define ⟨variable⟩ ⟨expression⟩)
                        |  (define (⟨variable⟩ ⟨variable⟩*) ⟨expression⟩)
                        |  (define (⟨variable⟩ ⟨variable⟩+.⟨variable⟩) ⟨expression⟩)
```

Figure 7.5: Syntax of Core Scheme

$\langle datum \rangle$

$\mathcal{D}[\![\texttt{\#t}]\!]$	$\equiv$	`true`
$\mathcal{D}[\![\texttt{\#f}]\!]$	$\equiv$	`false`
$\mathcal{D}[\![n]\!]$	$\equiv$	$\mathcal{N}[\![n]\!]$
$\mathcal{D}[\![c]\!]$	$\equiv$	$\mathcal{C}[\![c]\!]$
$\mathcal{D}[\![s]\!]$	$\equiv$	$\mathcal{S}[\![s]\!]$
$\mathcal{D}[\![i]\!]$	$\equiv$	$\mathcal{I}[\![i]\!]$

$\langle list \rangle$

$\mathcal{D}[\![\texttt{(}d_1\, d_2 \ldots d_n\texttt{)}]\!]$	$\equiv$	$\texttt{(}\mathcal{D}[\![d_1]\!]\texttt{, (}\mathcal{D}[\![d_2]\!]\texttt{, (}\ldots\texttt{(}\mathcal{D}[\![d_n]\!]\texttt{, ())}\ldots\texttt{)))}$
$\mathcal{D}[\![\texttt{(}d_1\, d_2 \ldots d_n\, \texttt{.}\, d\texttt{)}]\!]$	$\equiv$	$\texttt{(}\mathcal{D}[\![d_1]\!]\texttt{, (}\mathcal{D}[\![d_2]\!]\texttt{, (}\ldots\texttt{(}\mathcal{D}[\![d_n]\!]\texttt{, }\mathcal{D}[\![d]\!]\texttt{)}\ldots\texttt{)))}$

$\langle vector \rangle$

$\mathcal{D}[\![\texttt{\#(}d_1 \ldots d_n\texttt{)}]\!]$	$\equiv$	$\texttt{arrayoflist [}\mathcal{D}[\![d_1]\!], \ldots, \mathcal{D}[\![d_n]\!]\texttt{]}$

Figure 7.6: Translation of Scheme datum

⟨variable⟩		
$\mathcal{E}[\![x]\!]$	$\equiv$	x
⟨literal⟩		
$\mathcal{E}[\![\texttt{'}d]\!]$	$\equiv$	$\mathcal{D}[\![d]\!]$
$\mathcal{E}[\![\texttt{\#t}]\!]$	$\equiv$	`true`
$\mathcal{E}[\![\texttt{\#f}]\!]$	$\equiv$	`false`
$\mathcal{E}[\![n]\!]$	$\equiv$	$\mathcal{N}[\![n]\!]$
$\mathcal{E}[\![c]\!]$	$\equiv$	$\mathcal{C}[\![c]\!]$
$\mathcal{E}[\![s]\!]$	$\equiv$	$\mathcal{S}[\![s]\!]$
⟨procedure call⟩		
$\mathcal{E}[\![\texttt{(}e\, e_1\, e_2\, \ldots\, e_n\texttt{)}]\!]$	$\equiv$	$\mathcal{E}[\![e]\!]\ \texttt{(}\mathcal{E}[\![e_1]\!]\texttt{, (}\mathcal{E}[\![e_2]\!]\texttt{, (}\ldots\texttt{(}\mathcal{E}[\![e_n]\!]\texttt{, ())}\ldots\texttt{)))}$
⟨lambda expression⟩		
$\mathcal{E}[\![\texttt{(lambda}\, x\, e\texttt{)}]\!]$	$\equiv$	$\texttt{fn}\ x\ \texttt{=>}\,\mathcal{E}[\![e]\!]$
$\mathcal{E}[\![\texttt{(lambda (}x_1\, x_2 \ldots x_n\texttt{)}\ e\texttt{)}]\!]$	$\equiv$	$\texttt{fn (}x_1\texttt{, (}x_2\texttt{, (}\ldots\texttt{(}x_n\texttt{, ())}\ldots\texttt{))) =>}\,\mathcal{E}[\![e]\!]$
$\mathcal{E}[\![\texttt{(lambda (}x_1\, x_2 \ldots x_n\,\texttt{.}y\texttt{)}\ e\texttt{)}]\!]$	$\equiv$	$\texttt{fn (}x_1\texttt{, (}x_2\texttt{, (}\ldots\texttt{(}x_n\texttt{, }y\texttt{)}\ldots\texttt{))) =>}\,\mathcal{E}[\![e]\!]$
⟨conditional⟩		
$\mathcal{E}[\![\texttt{(if}\ e\, e_1\texttt{)}]\!]$	$\equiv$	$\texttt{case}\ \mathcal{E}[\![e]\!]\ \texttt{of}$
		$\texttt{in\_bool false => raise Match}\ \vert\ \texttt{\_ =>}\ \mathcal{E}[\![e_1]\!]$
$\mathcal{E}[\![\texttt{(if}\ e\, e_1\, e_2\texttt{)}]\!]$	$\equiv$	$\texttt{case}\ \mathcal{E}[\![e]\!]\ \texttt{of}$
		$\texttt{in\_bool false =>}\ \mathcal{E}[\![e_2]\!]\ \vert\ \texttt{\_ =>}\ \mathcal{E}[\![e_1]\!]$
⟨let expression⟩		
$\mathcal{E}[\![\texttt{(let (}x\, e\texttt{)}\ e'\texttt{)}]\!]$	$\equiv$	$\texttt{let val rec}\ x\ \texttt{=}\ \mathcal{E}[\![e]\!]\ \texttt{in}\ \mathcal{E}[\![e']\!]\ \texttt{end}$
		provided $(*)$
$\mathcal{E}[\![\texttt{(let (}x\, e\texttt{)}\ e'\texttt{)}]\!]$	$\equiv$	$\texttt{let val}\ x\ \texttt{=}\ \mathcal{E}[\![e]\!]\ \texttt{in}\ \mathcal{E}[\![e']\!]\ \texttt{end}$
		provided $(**)$
⟨assignment⟩		
$\mathcal{E}[\![\texttt{(set!}\ x\, e\texttt{)}]\!]$	$\equiv$	$x\ \texttt{:=}\ \mathcal{E}[\![e]\!]$
⟨definition⟩		
$\mathcal{P}[\![\texttt{(define}\ x\, e\texttt{)}]\!]$	$\equiv$	$\texttt{val rec}\ x\ \texttt{=}\ \mathcal{E}[\![e]\!]$, *provided* $(*)$
$\mathcal{P}[\![\texttt{(define}\ x\, e\texttt{)}]\!]$	$\equiv$	$\texttt{val}\ x\ \texttt{=}\ \mathcal{E}[\![e]\!]$, *provided* $(**)$
$\mathcal{P}[\![\texttt{(define (}x\, x_1\, x_2 \ldots x_n\texttt{)}\ e\texttt{)}]\!]$	$\equiv$	$\texttt{fun}\ x\ \texttt{(}x_1\texttt{, (}x_2\texttt{, (}\ldots\texttt{(}x_n\texttt{, ())}\ldots\texttt{))) =}\ \mathcal{E}[\![e]\!]$

Side conditions:
$(*)$ The definition is recursive and e is a `lambda` abstraction.
$(**)$ The definition is not recursive.

Figure 7.7: Translation of Scheme expressions and definitions

7.4 Dynamically Typed Scheme

In this section we define a type system for *Core-Scheme* completions. We assume the ideas and notation of Chapter 6 as well as the coercion calculus of Chapter 4. Figure 7.8 defines the type language, coercion language and the assignment of signatures to coercions. The coercion language is generated in the standard fashion, *except* for the new coercions `box` and `unbox`. These coercions are peculiar in that they do not inject/project to and from a sum-type, like the other primitive coercions. These new primitives are intended to support a completion inference system which detects the use of assigment in the source expression (see remarks about assignments in the previous section. See also below.)

Figure 7.9 defines the type asignment rules. We refer the reader to Chapter 4 and Chapter 6 for explanations of the last four rules of the type system. We assume here a *Core-Scheme completion language* which extends *Core-Scheme* expressions with coercion application, coercion abstraction and coercion instantiation, as introduced in Chapter 6.

We assume a function *Typeof* which assignes the expected base types to boolens, numbers, strings and `()` (the type of `()` being `nil`) We consider quotations with `'` (`quote`) typed by $\mathit{Typeof}(\texttt{'}) = \forall\alpha.\alpha \to \alpha$. The typing rules use a few abbreviations, as follows. We write

$$\texttt{(}\tau_1 \ldots \tau_n \,\texttt{.}\, \tau\texttt{)}$$

for the right-associated pair-type

$$\texttt{(}\tau_1 * \texttt{(}\tau_2 * \ldots \texttt{(}\tau_n * \tau\texttt{)} \ldots \texttt{))}$$

Here $n > 0$ is assumed. Also, we write

$$\texttt{(}\tau_1 \ldots \tau_n\texttt{)}$$

for the right-associated pair-type

$$\texttt{(}\tau_1 * \texttt{(}\tau_2 * \ldots \texttt{(}\tau_n * \texttt{nil)} \ldots \texttt{))}$$

In case $n = 0$, this is the type `nil`. We write

$$\tau\texttt{list}$$

as an abbreviation for the recursive type

$$\mu\alpha.\texttt{nil} + (\tau * \alpha)$$

Finally, we use the defined constructor `dyn`, for the *maximal* sum, in the by now standard fashion: $\texttt{(}\alpha_1, \ldots, \alpha_5\texttt{)dyn}$ abbreviates

$$\texttt{nil} + \texttt{bool} + \texttt{number} + \texttt{string} + \alpha_1 * \alpha_2 + \alpha_3 \to \alpha_4 + \alpha_5\ \texttt{vector}$$

Note that the type τ `box` is not represented in this sum; the reason is given below, where we discuss the `box`-types. We shall sometimes use the more abstract notation

$$(\tau^{(k)})\texttt{dyn}$$

with $\tau^{(k)} = \tau_1 \ldots \tau_k$ for the sum $\texttt{(}\tau_1, \ldots, \tau_k\texttt{)dyn}$, where k is tacitly assumed to be the cardinality of the type constructor alphabet. We use this notation generically, relative to a given constructor alphabet, as determined by context. In the expression $(\alpha^{(k)})\texttt{dyn}$ we assume the variables α_i to be distinct, unless anything else is said. *The reader be warned* that the order of variables in a type of the form $(\alpha^{(k)})\texttt{dyn}$ is significant. The order of summands shown above is fixed throughout the development, and the reader must either rememeber this order, or he must return to this definition in order to understand some of the expressions we are going to write down.

7.4.1 Imperative polymorphism

The `let`-rule deserves speicial attention. In this rule, the operation

$$\overline{\Gamma}(e, \omega)$$

quantifies the variables $FV(\omega) \setminus FV(\Gamma)$ *provided* e is a syntactic *value*, which is a variable, an abstraction, or a constant ($\langle datum \rangle$.) In case e is not a value, no quantification takes place. This rule, therefore, implements Wright's discipline of *imperative polymorphism*, [Wri94a], [Wri93]. The rule imposes a restricted polymorphism in comparison with that of, *e.g.*, *Standard ML* [MTH90] or *Standard Ml of New Jearsey* [MLNJ93], [HMV93]. The motivation for this restriction lies in the fact that it provides a sound simplification, which by-passes the need for a stratification of type-variables and quantification rules, as is found in the *SML*-concept of *applicative* versus *imperative* type variables (*SML of New Jearsey* has a more involved stratification.) Such rules were introduced in order to provide a type system which is *both*

- *sound* in the presence of imperative features (assignment, references) and control operations (exceptions, `callcc`, etc.), *and*
- a *conservative extension* of the Hindley-Milner system (which doesn't include the features of references and control)

The problems of type soundness in the presence of polymorphic references and control are complicated and well studied, see [Tof90], [MTH91], [DHM91], [DHM93], [HL92], [HL93].

Adopting the restricted rule is equivalent to dropping the *applicative* type variables from Tofte's rules [Tof90], which are the ones *Standard ML* adopted. THus, the restricted system only has "ordinary" types with "ordinary" type variables α, and the problematic *ML*-operations can be given the "naive" *SML*-types:

`ref`	$: \forall\alpha.\alpha \rightarrow (\alpha\,\texttt{ref})$
`!`	$: \forall\alpha.(\alpha\,\texttt{ref}) \rightarrow \alpha$
`:=`	$: \forall\alpha.(\alpha\,\texttt{ref}) \rightarrow \alpha \rightarrow \mathit{unit}$
`callcc`	$: \forall\alpha.(\alpha\,\texttt{cont} \rightarrow \alpha) \rightarrow \alpha$

Also, exceptions need no special treatment. See [Wri94a], [Wri93] for details.

The rule studied by Wright is not "new", since it was, *e.g.*, known to the authors of [MTH90] and considered a candidate for the *ML*-standard. The main reason for rejecting it appears to have been that the restricted rule results in a system which is *not* an extension (but a restriction) over the Hindley-Milner system [10] However, Wright [Wri94a], [Wri93] undertakes an empirical study of realistic *ML*-programs in order to determine how severe the restriction is in practice. His results are almost overwhelming, in favour of the view that the restriction is not only not severe but actually not very significant in practice. His data include programs such as the *Standard ML of New Jearsey* system and the *HOL 90 Theorem Prover*. These are systems of, respectively, over 60000 and 80000 lines of *ML* code. For these and other realistic cases Wright typically finds that there are some 4 to 10 places where the programs need to be changed, and the changes can typically be made by quite simple means, such as a few η-expansions. In view of the simplification provided by not having to deal with a stratified system of quantification, we feel that these empirical results strongly suggest that the restricted rule ought to be adopted in the present framework, and most certainly at this early stage of realization of the project suggested in the present report [11]

[10] Thanks to Mads Tofte for information about this.

[11] Wright's rule is adopted in his practical system for soft typing [Wri94], [WrCa94]. Also, it is worth noting that another reason for the restricted rule, besides the reasons of simplicity mentioned here, is given by Wright in

7.4.2 List types

The treatment of *Scheme* lists is a major challenge, and lists pose perhaps the single most important optimization problem. There are two kinds of problem; one is the `list` pattern matching forms of procedure parameters, and another is the general ubiquity of list data in *Scheme* programs.

In the rules for `lambda` abstractions with ⟨*list*⟩-pattern matching parameter forms we have taken the "conservative" approach of requiring the ⟨*list*⟩-valued parameters to have `list`-types. By type recursion we have the equivalence

$$\alpha\,\texttt{list} \approx \texttt{nil} + (\alpha * \alpha\,\texttt{list})$$

and therefore we can coerce in such a way that $\texttt{nil} \subseteq \alpha\,\texttt{list}$,

$$\texttt{[nil!]()} : \texttt{nil} + (\alpha * \alpha\,\texttt{list}) \approx \alpha\,\texttt{list}$$

and such that, for any $\texttt{M} : \tau$ and $\texttt{L} : \tau\,\texttt{list}$, we have $\texttt{(M . L)} \subseteq \tau\,\texttt{list}$,

$$\texttt{[pair!](M . L)} : \texttt{nil} + (\tau * \tau\,\texttt{list}) \approx \tau\,\texttt{list}$$

The introduction of type `list` leads to a need for handling *non-maximal sums*, since the type $\tau : \texttt{list}$ is not of the form $(\tau_1, \ldots, \tau_n)\texttt{dyn}$. Here it may be necessary to introduce "categorical coercions" of the kind

$$\bigvee_i c_i$$

to project a non-present component out of the `list`-sum; this can occur in error-situations, where we may need complex error-coercions. An example is the expression

```
(lambda x (x #t))
```

mentioned earlier, in Section 7.3.3. Using the "mediating morphism"

$$\texttt{[}{}^{\uparrow bool\to\alpha}_{\downarrow nil}, {}^{\uparrow bool\to\alpha}_{\downarrow \tau*\tau\ list}\texttt{]} : (\texttt{nil} + \tau * \tau\,\texttt{list}) \rightsquigarrow (\texttt{bool} \to \alpha)$$

we can complete the expression as

$$\texttt{(lambda x:}\tau\,\texttt{list ([}{}^{\uparrow bool\to\alpha}_{\downarrow nil}, {}^{\uparrow bool\to\alpha}_{\downarrow \tau*\tau\ list}\texttt{]x #t))} : \tau\,\texttt{list} \to \alpha$$

Note that this completion, in turn, can be translated to *SML* (using the translations given in Chapter 4) as

```
fn x => (case x of
           in_nil y  => raise TypeError
         | in_pair y => raise TypeError)  true
```

which will indeed produce a run-time error whenever applied to a terminating argument.

[Wri94a]; this is the fact that, in a stratified system, it may be impossible to assign compatible types to different (functional *vs* imperative) realizations of the same mathematical abstraction. This can make it impossible to write down a signature which can be matched by both a functional and imperative implementation. Using the restricted rule, this problem is no longer present.

7.4.3 Box types

Any *Scheme* ⟨*variable*⟩ is in principle assignable. An assignable object is classified as a `box`-typed object in the present framework, much as this is done in Wright's soft type system [Wri94]. The operations (coercions) `box` and `unbox` are to be interpreted as, respectively, creating a box and dereferencing a box, just like *Chez Scheme*'s operations of the same names (see [Dyb87].) Thus, `(box M)` can be thought of as returning a pointer to a cell containing `M`, and `(unbox M)` can be thought of as returning the contents of `M`, where `M` must be a box (a pointer.)

There are at least the following possible strategies for handling assignments:

1. Treat all variables as assignable, *i.e.*, automatically `box`-type.
2. Treat variables in a program as `box`-typed "only where necessary"
3. Treat assignment as a weaker operation, where assigned objects must be explicitly boxed by the programmer

The first option is appealing, because it is completely *uniform.* However, this would lead to an explosive proliferation of boxes in completions. This could potentially lead to an almost complete elimination of polymorphism, and we would effectively end up with a monomorphic completion system.

Note that variables which are assignable (in the extreme case, every variable, as in the first option) must be treated monomorphically. As is observed in [Wri94], in a program like

```
(set! y (lambda (x) x))
 ...

(let (x y) ...)
```

we cannot safely quantify the type of `y`. As a consequence of this, we must work under the principle that

- *a polymorphic completion inference system cannot abstract from the implicit boxing- and unboxing operations* of the *Scheme* source program.

This manifests itself in the introduction of `box` types and coercions in the typing rules for *DTS.*

Instead of uniformly considering any variable updatable, the second option declares a variable assignable only where "necessary", forced by the fact that the source program treats the variable as such. In a completion inference framework, this approach requires the construction of a *boxing analysis* which will automatically insert the boxing coercions, `box` and `unbox`. The analysis ought to conservatively extend the functional sublanguage, introducing no boxed objects at all for a source program with no assignment statements in it. For instance, the typing rule for `set!` might be used, by a completion mechanism, in such a way that the program phrase (considered earlier, in Section 7.3.3)

```
(lambda (x) (set! x #t))
```

would be completed as

```
(lambda (x : bool box) (set! x #t))
```

at the type `bool box` $\rightarrow$ `nil`, which in turn would yield the completion

```
((lambda (x) (set! x [bool!]#t))  [box][func!](lambda (y) y))
```

of the larger program considered in Section 7.3.3.

Note that, although we could easily define standard tagging and check-untaggging coercions (`box!` and `box?`) for the `box`-types, we have no need for them in the first two approaches, because *any* variable is (coercible to be) assignable, and the check that assignment is only applied to variables is not a run-time check, but a syntax check.

Finally, the third option is tantamount to giving up the inference problem, and it leads to a system which is no longer faithful to *Scheme*. However, it is an option which it may be worth while considering, if we are not religious about dynamic typing having to deal with exactly *Scheme*. Presumably, this approach will need tagging and check-untagging coercions (`box!` and `box?`), since we can no longer coerce an object to a boxed one. The third approach is what we would get by throwing out *Scheme*'s `set!`-operation and introducing instead the `set-box!`-operation of *Chez Scheme*, together with the operations `box` and `unbox`.

The second option is clearly preferable, if it can be supported in practice by a boxing inference mechanism. Note that the boxing coercions are closely related to the recursion coercions of Σ^{μ}; they have the same "special status" in that they work independently of the dynamic system (they do not inject/project to and from the sum-types.) The boxing analysis suggested here is related to the representation analysis developed by Henglein and Jørgensen, in [HJ94], with the difference that the souce of boxing demands in [HJ94] is polymorphism, whereas here the source is assignment. Note also that, although the *safety* problems discussed in Section 6.6.1 do not apply for boxing (since boxing demands can always be satisfied by coercion), caution must *still* be exercised in how we place coercions, for different semantical reasons. Although this issue cannot, strictly speaking, be raised at the level of *Core-Scheme*, it will arise once we move to extensions of the core language (Section 7.7.) If we assume for a moment that we are in an extended framework, where we add a procedure such as *Scheme*'s equivalence predicate `eqv?`, then we can consider the programs

```
(define (p1 x) (eqv? x x))

(define (p2 x y) (eqv? x y))
```

Then the value of the program

```
(p1 '(1 . 2))
```

will be `#t` (true), whereas the value of

```
(p2 '(1 . 2) '(1 . 2))
```

is `#f` (false). The reason is that `eqv?` uses pointer equality (or, more abstractly, "box equality") on pairs, and in the call to procedure `p2`, the two actual arguments to `eqv?` are assigned different locations, whereas in the procedure `p1`, `eqv?` is called with the same location at both arguments. Now, since `eqv?` uses pointer equality, a completion under the second option (which relies on boxing inference) must somehow enforce that `eqv?` is called with boxed objects. There is in principle two ways this could be done. One would be to perform the completion of, say, `p1` at the type

$$\forall\alpha\beta.(\alpha\ \beta) \rightarrow \texttt{bool}$$

However, this could have the consequence that we end up with the completion of `p1`:

```
(define (p1 x) (eqv? [box]x [box]x))
```

and this completion is *unsound*, since it changes the semantics of the underlying program, calling `eqv?` with two different locations in every case. The sound completion of `p1` must be performed at type

$$\forall\alpha\beta.\texttt{(}\alpha\,\texttt{box}\ \beta\,\texttt{box)} \to \texttt{bool}$$

inserting no coercions into `p1`, and the call to `p1` would be completed as

```
(p1 [box]'(1 . 2))
```

However, it appears that this complication could be uniformly solved by restricting the insertion of `box`-coercions according to the principle

- `box`-*coercions can only be applied at data-creation points.*

Here, at data-creation point must include tagged objects, where the tag is applied at a "usual" creation point of the source program, since we must be able to box coerced objects (as shown in an example program above.) The boxing discipline could therefore be smoothly integrated with a Σ_{pf}^{∞} completion system; for stronger systems, one must take special care that `box`'es are placed in a sound manner.

7.4.4 Conditionals

The conditional (considered wrt. its test argument) is one of the many constructions in *Scheme* which are defined *universally*, in that they work on *all* objects whatsoever. It shares this characteristic with, *e.g.*, the *type testing predicates* and the *equivalence predicates*, which we consider in Section 7.7 below.

This feature of universality manifests itself in the typing rules of Figure 7.9 by the absence of any constraints on the type of the test; this type τ is not related to any other type in the rules for conditionals, and hence it can be chosen freely, and a function such as

```
(define tst (lambda (x) (if x 1 0)))
```

can be given type $\forall\alpha.\alpha \to \texttt{number}$.

Universally defined operations pose a number of interesting problems in a typed framework. The type of `tst` above is polymorphic in the argument, but this is a *non-parametric* form of polymorphism, we are introducing here. Parametricity (see [Rey83]) refers to the property of *representation independence*, a principle which is often phrased in more operational terms as the possibility of *uniform implementation*; a parametrically polymorphic function can be implemented by a single piece of code, which works uniformly over all type instances. In practice, this leads to problems of optimization of data-representation, since it may be expensive to actually use a uniform representation scheme. Typically, such a scheme will use *boxed representations* of polymorphic data, and the optimization problem consists in finding opportunities for more efficient, *type-specific data-representations.* Much recent work has gone into attacking such problems, *e.g.*, [Ler92], [HJ94], [HM95]. Now, the type of `tst` above is not parametric, since an implementation of the conditional must essentially inspect the type of the test at run-time. This could, conceptually, be done by a run-time type-analysis, using, *e.g.*, the `typecase` construct known from *ML dynamics* ([ACPP89], [LM91]), as in

```
fun tst x =
        typecase x of
          dynamic(y: bool) => if y then 1 else 0
        | _                => 0
```

where a run-time representation of the type of the argument is inspected. There are other variations on this, as suggested recently in [HM95], [DRW95].

7.5 Abstraction principle

Universally defined operations, of which the conditional is one example, puts focus on an important principle of dynamic typing, which we shall call the *abstraction principle*. We illustrate it with the conditional construct.

The `typecase`-implementation of a universally defined operation shown in the previous section makes the idea obvious that we could use run-time type operations to *implement* universally defined operations, as in

$$\lambda xyz.\begin{pmatrix} \texttt{case}\; x \;\texttt{of} \\ \mathit{in\_bool}\; b \Rightarrow b \mid \_ \Rightarrow \mathit{false} \end{pmatrix} \rightarrow y\;;\; z$$

where $\bullet \rightarrow \bullet\,;\,\bullet$ is a monomorphic conditional, which is only defined on boolean test arguments. This view of the dynamic conditional appears to be closely related to our view (cf. Chapter 1) of other dynamic operations as incomplete specifications of the corresponding run-time operations; for instance, we can view dynamic addition $+$ as a shorthand (to be expanded by the compiler) for

$$\texttt{[number!]}(\oplus\;\texttt{[number?]}\;\bullet\;\;\texttt{[number?]}\bullet)$$

where $\oplus$ is non-dynamic (only defined on numbers.) We can even write the conditional above with coercions, in the style of the addition, if we use the "mediating morphism" coercion (cf. Chapter 4)

$$c_{cond} \equiv [\lambda x.\mathit{false}, \mathit{id}_{\texttt{bool}}, \lambda x.\mathit{false}, \ldots, \lambda x.\mathit{false}]$$

at the signature $(\alpha_1 \ldots \alpha_n)\texttt{dyn} \rightsquigarrow \texttt{bool}$; here $\lambda x.\mathit{false}$ represents a special coercion at $\alpha \rightsquigarrow \texttt{bool}$. Then

$$([c_{cond}]\bullet) \rightarrow \bullet\,;\,\bullet$$

would express the real meaning of the dynamic conditional. This is the strategy taken by our *ML*-translation in Figure 7.7; the "categorical coercion" used above is expressed in *ML*, as the injection analysis in

```
if (case x of
       in_bool b => b
     | _         => false) then ... else ...
```

Now, we could easily enforce this view *explicitly* in our typing rules for *Dynamically Typed Scheme.* All we have to do is to rule

$$\frac{\Gamma \vdash e : (\tau_1 \ldots \tau_k)\texttt{dyn} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{(if}\; e\, e_1\, e_2\texttt{)} : \tau}$$

and similarly for the other conditional rule. However, this would constitute a deviation from a very pleasing *abstraction principle* which underlies just about everything else we have done in this report, namely that

- *The calculi of completions should model just the aspect of run-time type safety*

It is important to stress that, in our discipline of insertion of tagging and check-untag coercions, we have in principle considered the problem of run-time type operations *in abstraction from all other aspects of the run-time system.* A case in point is the question (discussed earlier, in Section 6.8) of run-time tags in relation to garbage collection. While it is a very interesting question to what extent tags are needed for various garbage collection schemes, we confine ourselves in principle to considering the question of whether or not a given operation is necessary for run-time type safety. It is important to study this problem in *abstraction from these other considerations*, because doing so will allow us to say something about which *specific* constraints are imposed by considerations of run-time type safety, and it would certainly be a serious and unnecessary limitation of our work, were we to base it on highly specific assumptions about, say, other features of the run-time system. The reason why the suggested, alternative typing rules for the conditionals represent a deviation from this important principle, then, is that they force test expressions to be tagged even though *no* run-time type constraints are imposed by the test of the conditional. The presence of coercions in a completion such as

```
((lambda (x) (if tst
                 (if x M N)
                 (car [pair?]x)))  [pair!](1 . 2))
```

are *only* necessitated by the those typing rules and have nothing to do with run-time type safety. If we look at *Dynamically Typed Scheme* as a framework for type-recovery and static debugging, the completion just shown would be outright misleading. From this point of view, irrelevant questions of implementation are confused with the issue of run-time type safety.

We observed, in Section 7.4.3, that a polymorphic completion inference system cannot abstract from data representation features in the presence of assignment. One may ask whether the ensueing introduction of `box`-coercions (Section 7.4.3) is in conflict with the abstraction principle. The fact is that it isn't, since the boxing and unboxing coercions are "soft coercions" which work independently of the dynamic system of injection and projection to and from the tagged sum-types (cf. the remarks about this in Section 7.4.3.) Thus, it is possible to regard the boxing coercions as orthogonal to the dynamic type coercions, with no irrelevant interference between them.

Type language

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \texttt{nil} \mid \texttt{bool} \mid \texttt{number} \mid \texttt{string} \\ & \mid & \tau * \tau \mid \tau \to \tau \mid \tau\,\texttt{box} \mid \tau\,\texttt{vector} \mid \mu\alpha.\tau \mid \tau + \tau \\ \\ \omega & ::= & \tau \mid s \Rightarrow \omega \\ \\ \sigma & ::= & \omega \mid \forall\alpha.\sigma \end{array}$$

Coercions

$$\vdash \texttt{tc!} : \forall\alpha^{(k)}\beta.\, tc(\alpha^{(k)}) \leadsto tc(\alpha^{(k)}) + \beta \quad tc \notin \{\texttt{box}, \texttt{unbox}\}$$

$$\vdash \texttt{tc?} : \forall\alpha^{(k)}\beta.\, tc(\alpha^{(k)}) + \beta \leadsto tc(\alpha^{(k)}) \quad tc \notin \{\texttt{box}, \texttt{unbox}\}$$

$$\vdash \texttt{box} : \forall\alpha.\, \alpha \leadsto \alpha\,\texttt{box} \qquad \vdash \texttt{unbox} : \forall\alpha.\, \alpha\,\texttt{box} \leadsto \alpha$$

$$\frac{\vdash c_i : \tau_i \leadsto \tau_i' \qquad , tc \not\equiv \to}{\vdash tc(c^{(k)}) : tc(\tau^{(k)}) \leadsto tc(\tau'^{(k)})}$$

$$\frac{\vdash c_1 : \tau_1 \leadsto \tau_1' \qquad \vdash c_2 : \tau_2 \leadsto \tau_2'}{\vdash\ c_1 \to c_2 : (\tau_1' \to \tau_2) \leadsto (\tau_1 \to \tau_2')}$$

Figure 7.8: Type language and coercions

$$\frac{\Gamma(x) \succ \tau}{\Gamma \vdash x : \tau} \qquad \frac{\mathit{Typeof}(\chi) \succ \tau}{\Gamma \vdash \chi : \tau}$$

$$\frac{\Gamma \vdash d : \tau \qquad \Gamma \vdash d' : \tau'}{\Gamma \vdash \texttt{(}d\,\texttt{.}\,d'\texttt{)} : \tau * \tau'}$$

$$\frac{\Gamma \vdash d_i : \tau_i}{\Gamma \vdash \texttt{\#(}d_1 \ldots d_n\texttt{)} : (\tau_1 * \ldots * \tau_n)\texttt{vector}}$$

$$\frac{\Gamma \vdash e : (\tau_1 \ldots \tau_n) \to \tau' \qquad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{(}e\, e_1 \ldots e_n\texttt{)} : \tau'}$$

$$\frac{\Gamma, x : \tau\,\texttt{list} \vdash e : \tau'}{\Gamma \vdash \texttt{(lambda}\ x\, e\texttt{)} : \tau\,\texttt{list} \to \tau'}$$

$$\frac{\Gamma, x_i : \tau_i \vdash e : \tau'}{\Gamma \vdash \texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e\texttt{)} : (\tau_1 \ldots \tau_n) \to \tau'}$$

$$\frac{\Gamma, x_i : \tau_i, y : \tau\,\texttt{list} \vdash e : \tau'}{\Gamma \vdash \texttt{(lambda (}x_1 \ldots x_n\,\texttt{.}\,y\texttt{)}\ e\texttt{)} : (\tau_1 \ldots \tau_n\,\texttt{.}\,\tau\,\texttt{list}) \to \tau'}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e_1 : \tau'}{\Gamma \vdash \texttt{(if}\ e\, e_1\texttt{)} : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e_1 : \tau' \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \texttt{(if}\ e\, e_1\, e_2\texttt{)} : \tau'}$$

$$\frac{\Gamma \vdash x : \tau\,\texttt{box} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{(set!}\ x\, e\texttt{)} : \texttt{nil}}$$

$$\frac{\Gamma \vdash e : \omega \qquad \Gamma, x : \overline{\Gamma}(e, \omega) \vdash e' : \tau'}{\Gamma \vdash \texttt{(let (}x\ e\texttt{)}\ e'\texttt{)} : \tau'}$$

$$\frac{\Gamma \vdash M : \tau \quad c : \tau \leadsto \tau'}{\Gamma \vdash [c]M : \tau'}$$

$$\frac{\Gamma \vdash M : \tau \quad \tau \approx \tau'}{\Gamma \vdash M : \tau'}$$

$$\frac{\Gamma, \kappa : s \vdash M : \omega}{\Gamma \vdash \Lambda\kappa : s.M : s \Rightarrow \omega}$$

$$\frac{\Gamma \vdash M : s \Rightarrow \omega \qquad \vdash c : s}{\Gamma \vdash M\{c\} : \omega}$$

Figure 7.9: Dynamically Typed Scheme (DTS)

7.6 ML-Scheme

In this section we define yet a completion language for *Core-Scheme*. This language, called *ML-Scheme* (abbreviated *MLS*) is specifically targeted at translating *Core-Scheme* to *SML*. The language is defined by exchanging the rules for the conditional construct of Figure 7.9 with those shown in fig: Figure 7.10. Referring to our discussion in the previous section, this amounts to dropping the abstraction principle in moving from *DTS* to *MLS*. The reason why this is an option is not far away: the problem of translation contains the problem of implementation; therefore we can no longer abstract from details of implementation. Our conditional rules reflect a more *general* choice here, to be expanded in Section 7.7, of implementing the universal operations of *Scheme* via dynamic type operations.

The framework of Section 7.2 is easily transferred to the more comprehensive language considered here. The only really new feature is the introduction of the boxing and unboxing coercions, which are considered in Section 7.6.3 below. We assume the polymorphic *ML*-datatype `('a1, 'a2, 'a3, 'a4, 'a5)dyn` given by

```
datatype ('a1,'a2,'a3,'a4,'a5)dyn = in_nil    of unit
                                  | in_bool   of bool
                                  | in_number of int
                                  | in_string of string
                                  | in_symbol of string
                                  | in_char   of string
                                  | in_pair   of 'a1 * 'a2
                                  | in_func   of 'a3 -> 'a4
                                  | in_vector of 'a5 array
```

The corresponding negative primitives are obvious; also, we leave to the reader to extend the translation of Section 7.2 to translate the standard (non-boxing) coercions of Figure 7.8 into *ML*.

7.6.1 Run-time types vs. coercions

There are interesting alternatives to relying on type tagging operations for implementing *Scheme*'s universally generic operations. A major candidate would be to rely on inferred *run-time type* information. Several frameworks for optimized manipulation of types at run-time have been suggested recently, as a key ingredient for compiling polymorphically typed languages ([HM95]) and coping with *ad-hoc-polymorphism* (overloading, see [DRW95]) as well as for tag-free garbage collection (see [Tol94].) These frameworks proceed by defining a richer, type-manipulating language, which we generically call *TML*; this language extends the source language (we consider here *ML*) with non-parametric polymorphic type operations, such as type dispatching facilities which allow run-time type directed recursive descent traversals of data in a way which is related to *ML dynamics* ([ACPP89], [LM91].) Given this richer language, one typically specifies a translation

$$\mathcal{T} : ML \rightarrow TML$$

of *ML* into the richer language. In some cases, the suggestion is to extend *ML* itself with new, non-parametric constructs (as, for instance, the `generic` construct in [DRW95].) We might imagine using such frameworks for our present purposes, since the languages *TML* typically appear to be strong enough to implement such features as universally defined functions. Conceptually, this could be realized by translating *TML* back to *ML*, as in the composition of translations

$$\textit{Scheme} \rightarrow \textit{ML-Scheme} \rightarrow \textit{TML} \rightarrow \textit{ML}$$

However, in so far as we wish to stay within *ML* as we know it today, it is not clear to us how complicated the step from *TML* (or, perhaps, some suitable subset) back to *ML* would be. Naturally, the constructs contained in *TML* typically cannot be directly translated back to *ML*; this is the point of having the richer language. Therefore, one must use some form of encoding, just as the insertion of coercions in a completion is a type-restoring encoding of the source program. It is not clear to us how complicated such encoding would have to be. Also, it is not clear, at this time, that the very idea of relying on run-time type information is in the end more efficient than relying on run-time type tags. [12] It would also be a possibility to consider translating *Scheme* into one of the proposed *extensions* of *ML*. Unfortunately, we must leave these suggestions unexplored in this report. In the remainder of this work we shall outline how it is possible to stay within the framework of run-time type tagging and -checking already developed.

7.6.2 Lists

List processing is, not surprisingly, an absolutely central element of *Scheme*, and an efficient translation must use *type specific, efficient data-representations* here. Thus, it is generally desirable to use *ML* lists wherever possible. For example, consider a function such as `nlist`, which builds a list consisting of `n,n-1,...,1`:

```
(define (nlist n)
        (if (zero? n)
            '()
            (cons n (nlist (- n 1)))))
```

Under a completely uniform translation scheme, this function could, in the extreme, be translated to something as complicated as

```
fun nlist (n, ()) =
          if n = 0 then inrec(in_nil ())
          else
            inrec(in_pair (inrec(in_number n), nlist(n-1, ())) )
```

at the type `int` $*$ `unit` $\rightarrow$ `recty`. However, if we can discover that the structure produced by the function is a list, then we can translate the function to

```
fun nlist (n, ()) =
          if n = 0 then []
          else
            n :: (nlist(n-1, ()))
```

at `int` $*$ `unit` $\rightarrow$ `int list`, which is much more palatable.

The example just considered also leads to the idea of optimizing the translation of parameter lists. It seems reasonable to suppose that we could discover that `nlist` is a unary function, and we might therefore want to go still further and translate it as `nlist1`:

[12]See the cautious remarks concerning this at the end of [HM95].

```
fun nlist1 n =
           if n = 0 then []
           else
             n :: (nlist1 (n-1))
```

which is indeed how a human would be likely to translate the single *Scheme* function `nlist`. However, optimization of data-representations raises problems akin to the *safety problems* discussed earlier (Section 6.6.1). Consider as an example the program context C given by the declarations

```
(define g (lambda (x . y) y))

(apply (if tst [] g) 0)
```

where `apply` is defined as earlier. The question now is, how should we translate the call to `apply` in the expression C[`nlist`] (which evaluates to `()`), if `nlist` is translated seperately as `nlist1` above? The problem is that, if we translate `g` according to our translation scheme, as

```
val g = fn (x, (y, ())) => y
```

then there is not one single calling convention which will work, since `g` expects to be called by lists. The problem is that we have two calling conventions in play at the same time, one which we may denote *call-by-tuple* (to be used in the call to `nlist1`, in the form of a 1-tuple) and the other which we may denote *call-by-*$\langle$*list*$\rangle$ (to be used in the call to `g`.)

There is, however, a lot to be said in favour of the *call-by-*$\langle$*list*$\rangle$ convention adopted in our translation schemes. The only uniform alternative appears to be to use *ML*-lists uniformly, as in the following translations

```
fun nlist2 [n] = if n = 0 then ...
val g1 = fn x::y => y
```

corresponding to the translation of the application

```
...

(apply (if tst nlist2 g1) [0])
```

This scheme has the serious drawback of forcing all argument lists to be *homogeneous*, which will proliferate coercions.

The problem illustrated by the context C is aggrevated in a setting where we want a *fully modular* translation, since in this setting the translation must work for *all possible contexts*, which means that we have no information at all about the possible contexts of use. Balancing these conflicting considerations, we believe that it is better to stick to the uniform scheme of *call-by-*$\langle$*list*$\rangle$, and that it is not very likely to be realistic to try to optimize this calling convention in a fully modular setting.

7.6.3 Boxes

The boxing and unboxing coercions are the only essentially new feature of the coercion language considered here. As already suggested, these coercions should become operations on *ML*-references, with

$$\begin{array}{lcl} \lceil \texttt{box} \rceil & \equiv & \texttt{ref} \\ \lceil \texttt{unbox} \rceil & \equiv & \texttt{!} \end{array}$$

and the corresponding translation on types,

$$\lceil \tau\, \texttt{box} \rceil \equiv \lceil \tau \rceil\, \texttt{ref}$$

$$\frac{\Gamma \vdash e : (\tau_1 \ldots \tau_k)\texttt{dyn} \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \texttt{(if}\ e\, e_1\texttt{)} : \tau}$$

$$\frac{\Gamma \vdash e : (\tau_1 \ldots \tau_k)\texttt{dyn} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{(if}\ e\, e_1\, e_2\texttt{)} : \tau}$$

Figure 7.10: Special typing rules for the conditional of ML-Scheme

7.7 Extension to full *Scheme*

7.7.1 Derived expression forms

The definition of R4RS [CR91] gives a systematic way of translating derived expressions into the core. This translation can be adopted for the translation into *ML*. We suggest that the translation should not be adopted for dynamic type analysis, in *DTS*, since we wish to annotate the source program with coercions without changing the source itself.

7.7.2 Type testing predicates

In *Scheme*, any object can be tested with the following *type testing predicates*:

```
boolean?         pair?          null?          list?
symbol?          number?        char?          string?
vector?          procedure?     input-port?    output-port?
eof-object?
```

of which at most one will be true. [13] These predicates are completely generic, universally defined procedures. According to the discussion of the conditionals in and Sections 7.5, 7.4.4 and Section 7.6, we suggest treating these predicates differently, in *DTS* as compared with *MLS*. The *DTS*-types will preserve the *abstraction principle*, being uniformly

$$\langle \textit{type predicate} \rangle : \forall \alpha . \alpha \rightarrow \texttt{bool}$$

whereas the *MLS*-types will be "implementation oriented", being uniformly

$$\langle \textit{type predicate} \rangle : \forall \alpha_1 \dots \alpha_k .\ (\alpha_1 \dots \alpha_k)\texttt{dyn} \rightarrow \texttt{bool}$$

These predicates can be implemented in *ML*, as

```
fun tc_pred x = case x of
                    in_tc y => true
                  | _       => false
```

where `tc` ranges over the type specific tag-name. Input/output ports can be represented by *ML*-`streams`, and the end-of-file object predicates are easily incoporated into the type `( ... )dyn`, and their predicates translate in analogous same way.

7.7.3 Equivalence predicates

Scheme has three generic equivalence predicates, `eq?`, `eqv?` and `equal?`, listed in order of increasing coarseness; viewed as set-theoretic relations, they satisfy:

$$\texttt{eq?} \subseteq \texttt{eqv?} \subseteq \texttt{equal?}$$

A finer relation will use "box equality" (pointer equality) more often than a coarser (greater) relation.

[13] *Scheme* has a number of other generic predicates, the numerical type tests, `complex?`, `real?` etc. which are not considered here.

Since it is consistent with the definition [14] of the predicate `eq?` to use `box`-equality for all types except the booleans and empty lists, we can easily translate this predicate to *SML*, as follows:

```
fun eq(x,(y,())) =

    case (x, y) of
      (ref (in_nil _)        , ref (in_nil _))          => true
    | (ref (in_bool b)       , ref (in_bool b'))        => b = b'
    | (r as ref (in_number _), r' as ref (in_number _)) => r = r'
    | (r as ref (in_string _), r' as ref (in_string _)) => r = r'
    | (r as ref (in_symbol _), r' as ref (in_symbol _)) => r = r'
    | (r as ref (in_pair _)  , r' as ref (in_pair _))   => r = r'
    | (r as ref (in_pair _)  , r' as ref(in_pair _))    => r = r'
    | (r as ref (in_func _)  , r' as ref (in_func _))   => r = r'
    | (r as ref (in_vector _), r' as ref (in_vector _)) => r = r'
    | _                                                 => false
```

This translation corresponds to working with the *MLS*-typing

$$\texttt{eq?} : \forall\alpha^{(k)}\beta^{(k)}.\texttt{((}\alpha^{(k)}\texttt{)dyn}\quad\texttt{(}\beta^{(k)}\texttt{)dyn)} \rightarrow \texttt{bool}$$

In accordance with the abstraction principle (Section 7.5) we stipulate the *DTS*-typing

$$\texttt{eq?} : \forall\alpha\beta.\texttt{(}\alpha\quad\beta\texttt{)} \rightarrow \texttt{bool}$$

The predicate `eqv?` can be implemented similarly, with the same typings (in its *MLS*- and *DTS*-versions, respectively), since it is consistent with the definition to let `eqv?` be as `eq?` with the difference that `eqv?` uses numerical equality rather than box-equality on numbers.

The real problem comes when we consider the procedure `equal?`. The problem here is that this procedure more or less identifies (yields `#t` on) all objects which print the same (under *Scheme*'s `print` procedure), see [CR91], section 6.2. This means that `equal?` must be implemented by a complete recursive descent in the contents of pairs, vectors and strings, whereas `eqv?`-equivalence is used elsewhere. This has the consequence that a faithful translation of this procedure must expect its arguments to be boxed, recursively down through the data-structures, wherever pointer equality is invoked. In analogy with our discussion in Section 7.4.3, we might, in the extreme case, implement this by *uniformly* using a single "boxed universal type", let us call it $\mathbf{D}^{\texttt{box}}$, a monomorphic, recursive dynamic type the components of which are boxed according to *Scheme*'s general *storage model* (see [CR91], section 3.5) in which pairs, vectors and strings implicitly denote (sequences of) storage locations. However, as we have argued earlier (Section 7.4.3), a uniform implementation of *Scheme*'s complete storage model will lead to a total elimination of polymorphism, and therefore this is highly undesirable. A better alternative would be to work with the type constant $\mathbf{D}^{\texttt{box}}$ under completion inference; this type should have the effect of the type **dynamic** in Henglein's monomorphic inference algorithms, [Hen92], [Hen92a], [Hen94]. This should be done in such a way that the type $\mathbf{D}^{\texttt{box}}$ is introduced "only where necessary".

The type $\mathbf{D}^{\texttt{box}}$ could be implemented as the *SML* datatype

[14]The definition of [CR91] leaves open (*unspecified*) the behaviour of the equivalence predicates for certain types of objects. We refer the reader to [CR91], section 6.2 for the details.

```
      datatype D    = inD_nil    of unit
                    | inD_bool   of bool
                    | inD_number of number
                    | inD_string of boxedstring
                    | inD-symbol of string
                    | inD_char   of string
                    | inD_pair   of Dbox * Dbox
                    | inD_func   of Dbox -> Dbox
                    | inD_vector of Dbox array

     withtype Dbox  = D ref
```

where `boxedstring` is a boxed implementation of the `string`-type (see Section 7.7.4 below.) Note that this type can realized in terms of the `dyn`-type, by the definition

```
      datatype Dbox = inD of ((Dbox, Dbox, Dbox, Dbox, Dbox) dyn) ref
```

Using explicit recursion (in the style of the `recty`-injections shown earlier) we could even factor this definition further, which would give a complete analysis of the type into its logical components, viz. dynamic sum, recursion and boxing (referencing.) It is interesting, in principle, to note this, because it shows that we can, if we so choose, remain within a completely uniform framework. The *compression* of the factored type just suggested into a type like `Dbox` (first version shown) requires an amount of cleverness in an analysis which is presumably not required for inferring the factored type, just because the latter type can be found by a completely uniform process; however, as we noted in the beginning of the present chapter, in relation to type recursion, cleverness pays off in terms of efficiency. Assuming, for illustration, the second version of `Dbox` above, we can implement the predicate `equal?` by

```
      fun equal (x, (y, ())) =
          case (x,y) of
            (inD (ref (in_nil _)),
             inD (ref (in_nil _)))            => true

          | (inD (ref (in_number n)),
             inD (ref (in_number n')))        => n = n'

          | (inD (ref (in_bool b)),
             inD (ref (in_bool b')))          => b = b'

          | (inD (ref (in_string s)),
             inD (ref (in_string s')))        => string_eq (s, (s',()))

          | (inD (ref (in_pair (Db1, Db2))),
             inD (ref (in_pair (Db1', Db2')))) => (equal (Db1,
                                                          (Db1',())))
                                                   andalso
                                                  (equal (Db2,
                                                          (Db2', ())))
          | (inD (r  as ref(in_func _)),
             inD (r' as ref (in_func _)))      => r = r'

          | (inD (ref (in_vector v)),
             inD (ref (in_vector v')))         => vector_eq (equal,
                                                              (v,
                                                              (v', ())))
          | _ => false
```

at *ML* type

$$\texttt{equal} : \texttt{Dbox} * (\texttt{Dbox} * \texttt{unit}) - > \texttt{bool}$$

Here we assume functions `string_eq` and `vector_eq` for string- and vector-equality (see Section 7.7.4.) The vector-equality function is parameterized over an equivalence predicate; in the implementation of `equal?` we call this function with the function `equal?` itself, to effect a recursive traversal of the structure.

It is perhaps doubtful whether such an implementation of equality is ever going to be satisfactory, since it runs the risk of effectively turning off polymorphism in completion. There seems to be two attitudes, we can take towards this:

1. Live with it, and acknowledge that full *Scheme* cannot be faithfully implemented, unless we descend to a monomorphic framework, based on a single, universal type.
2. Consider an extended target language, such as *ML* with some form of polymorphic *dynamics* ([ACPP89], [LM91], see also [DRW95].)

The first attitude may be said to be somewhat pessimistic, because the problems will not be encountered in programs which do not use the problematic feature. We shall meet related problems again, in particular in Section 7.7.6. Since most real programs typically begin by reading something and end by writing something, the pessimistic attitude may not be quite unwarranted.

The *DTS*-typing of `equal?` will be, according to the abstraction principle:

$$\texttt{equal?} : \forall\alpha\beta.(\alpha\ \ \beta) \rightarrow \texttt{bool}$$

as for the other equivalence predicates.

7.7.4 Mutable objects

In the storage model of *Scheme*, strings are sequences of characters which denote storage locations, as many locations as there are characters in the string. Strings are updatable, by the procedure `string-set!`. Strings share these characteristics with vectors and pairs, which come with updating procedures, `vector-set!` for vectors, and `set-car!` and `set-cdr!` for pairs. Dereferencing can be done by the procedures `string-ref` for strings, `vector-ref` for vectors, and `car`, `cdr` for pairs. However, *constants* of the types mentioned here, as well as objects created by *Scheme*'s procedure `symbol->string` are *immutable* (see [CR91], section 3.5.) Any other way of creating objects of the types mentioned will result in a *mutable* object, which can be updated and dereferenced; in particular, every one of the datatypes comes equipped with standard creators, the procedures `make-string`, `make-vector` and `cons`, respectively.

Strings can be implemented by the type

```
    datatype boxedstring = immutable of string
                         | mutable of string ref list
```

The standard equality predicate on strings `string=` can then be written in *SML*,

```
fun string_eq (s1, (s2, ())) =
    case (s1, s2) of
      (immutable s1'     , immutable s2')      => s1' = s2'
    | (mutable []        , mutable [])         => true
    | (mutable (s1'::s1s), mutable (s2'::s2s)) =>
          (!s1') = (!s2') andalso
          string_eq (mutable s1s, (mutable s2s, ()))
```

In the case of vectors, we must (lest we have to introduce further simulation) rely on standard extensions of *SML*, such as the structure `Array` of *SML of New Jearsey*, since arrays are not included in the definition [MTH90] (see [MLNJ93], and also [Pau91], 300, for the standard structure agreed on by implementors.)

For pairs, we must do analysis in *DTS* in order to "protect" polymorphism from being eliminated. The situation is again parallel to that of assignment (see the discussion in Section 7.4.3.) We propose using the boxing coercions here, too, in order to delimit the amount of boxing. In this vein, a pair `(M . N)` which is assigned in the first component, but not in the second, should be represented as `([box]M . N)`, and a procedure such as

```
(lambda (p) (set-car! p 5))
```

should be completed at type

$$\forall\alpha.(\texttt{int box} * \alpha) \to \texttt{nil}$$

corresponding to the typing of `set-car!`:

$$\texttt{set-car!} : \forall\alpha\beta.((\alpha\,\texttt{box} * \beta)\ \ \alpha) \to \texttt{nil}$$

In the present section we have met for the first time some procedures which are *not* universally generic. Procedure `set-car!` is an example, since `set-car!` produces a run-time type error unless the first argument is a a pair. Therefore, the problems of *polymorphic safety* (see Section 6.6.1 and Section 7.1) need in principle to be adressed here. However, since `set-car!` will always consume a pair, it is polymorphically safe to type `set-car!` as shown. The typing goes for both *DTS* and *MLS*, and the *ML*-translation is simply

```
fun set_car ((x, _), (y, ())) = x := y
```

at the type

$$\forall\alpha\beta.(\alpha\ \ \texttt{ref} * \beta) * (\alpha * \texttt{unit}) \to \texttt{unit}$$

Similar considerations hold for `cons`, `car` and `cdr` which recieve the typing

`cons`	$: \forall\alpha\beta.(\alpha\ \ \beta) \to \alpha * \beta$
`car`	$: \forall\alpha\beta.(\alpha * \beta) \to \alpha$
`cdr`	$: \forall\alpha\beta.(\alpha * \beta) \to \beta$

Note that a completion must explicitly dereference the result of extracsting a boxed component of a pair. For instance, the procedure

```
(lambda (x) (let (y (set-car! x 1))
                 (+ (car x) 2)))
```

must be completed at type

$$\forall\alpha.(\texttt{int box} * \alpha) \rightarrow \texttt{int}$$

with the dereferencing coercion `unbox`:

```
(lambda (x) (let (y (set-car! x 1))
                 (+ [unbox](car x) 2)))
```

Note that, at top-level, an expression such as

```
(let (p (lambda (x)
              (let (y (set-car! x 0))
                   (car x))))
     (p '(1 . 1)))
```

(with value `0`) should be completed as

```
[unbox](let (p (lambda (x)
                     (let (y (set-car! x 0))
                          (car x))))
            (p '([box]1 . 1)))
```

since the top-level must implicitly be regarded as a consumer of unboxed values (a similar approach will presumably be the default in polymorphic boxing analysis as developed in [HJ94] [15])

Whereas many non-universal procedures are straight-forward from the point of view of polymorphic safety, this is not true for all. We give an example of this in the Section 7.7.5.

7.7.5 Standard procedures and coercive types

Scheme has a host of standard procedures, and we must limit ourselves, in the present report, to considering a few of them, which will hopefully be illustrative for the problems that will be met in handling full *Scheme* and how they can be solved. In principle, we must handle all of *Schemes* so-called *essential procedures*. Procedures which are not essential can often be implemented by procedures which are.

In many cases, the framework developed so far will tell us how to represent the remaining procedures of *Scheme* in *DTS* and in *MLS* and how to translate then into *ML*. For vectors we have found it useful to rely on a standard *ML* library such as *SML of New Jearsey*; while this could in principle be simulated with *SML* references, the control operator `callcc` requires, for any realistic translation, that we rely on an extension of *SML* as defined in [MTH90] (see Section 7.7.7.)

However, we have not yet illustrated the problems of *polymorphic safety* and the use of *coercive types*. These features are potentially always in play whenever we translate a user-defined program, as they are when we come to translate *Scheme*'s library of standard procedures. In this section we discuss an example which illustrates a number of important problems. The example we shall take is the procedure `append` ([CR91], section 6.3.) In its full generality, `append` can be called in either of the following forms

`(append` $\langle \textit{list} \rangle$ $\langle \textit{list} \rangle^*$`)`, or

[15] Thanks to Jesper Jørgensen for discussions on this and related points.

`(append` $\langle \mathit{list} \rangle$ $\langle \mathit{list} \rangle^*$ $\langle \mathit{obj} \rangle$`)`

where $\langle \mathit{obj} \rangle$ is any object whatsoever. `Append` returns a list consisting of the elements of all the argument lists; if $\langle \mathit{obj} \rangle$ is not a *proper list*, then the result will be an *improper list.* [16] For instance, the result of evaluating

```
(append '(1 2) (lambda (x) x))
```

is (using a standard external representation of procedures)

```
(1 2 . #<procedure>)
```

For the sake of simplicity, let us consider a restriction of `append` to the two-argument form `(append2` $\langle \mathit{list} \rangle$ $\langle \mathit{obj} \rangle$`)`. Now, an *ML*-like type such as

$$\forall\alpha.\texttt{(}\alpha\,\texttt{list}\ \alpha\,\texttt{list)} \to \alpha\,\texttt{list}$$

is unsafe for *DTS*-procedure `append2`, since `append2` does not consume a list in its second argument (the second argument is not consumed at all.) However, we can express, in our type language, the type of improper lists of elements of type α with optional terminator [17] of type β; this type, call it $\texttt{(}\alpha,\beta\texttt{)ilist}$, emerges from the type $\alpha\,\texttt{list}$ by abstracting over the terminator type of the type of $\alpha\,\texttt{list}$'s, which is `nil`:

$$\texttt{(}\alpha,\beta\texttt{)ilist} \equiv \mu\gamma.\beta + (\alpha * \gamma)$$

Then we have for instance

$$\alpha\,\texttt{list} \approx \texttt{(}\alpha,\texttt{nil)ilist} \approx \texttt{(}\alpha,\alpha\,\texttt{list)ilist}$$

Using the type `ilist`, we can give this *DTS*-typing to `append2`:

$$\texttt{append2} : \forall\alpha\beta.\texttt{(}\alpha\,\texttt{list}\ \beta\texttt{)} \to \texttt{(}\alpha,\beta\texttt{)ilist}$$

In an ideal *DTS*, which includes some form of control-flow analysis, this type would be assignable to the following implementation of `append2`:

```
(define (append2 l1 l2)
        (if (null? l1)
            l2
            (cons (car l1) (append2 (cdr l1) l2))))
```

Here we assume that `null?` is handled as the predicates of Section 7.7.2, in *DTS*. Note that, by the recursion equivalences shown above, we can instantiate `append2` at the "*ML*-type"

$$\forall\alpha.\texttt{(}\alpha\,\texttt{list}\ \alpha\,\texttt{list)} \to \alpha\,\texttt{list}$$

However, from the point of view of practical *completion inference*, the typing given to `append2` may be too sophisticated, because its use of list types does not square with a simple framework of injecting and projecting to and from the single, *maximal* polymorphic sum-type $(\alpha^{(k)})\texttt{dyn}$, as exemplified by the inference framework of Chapter 6. Under this simplification one may have to *weaken* the type of `append2` for a polymorphically safe completion. Such is

[16] See [CR91], section 6.3, for the notion of proper and improper lists.

[17] We call an occurrence of `T` the terminator of an improper list of the form `(M1 ...Mk . T)`

```
(define (append2 l1 l2)
        (if (null? l1)
            l2
            [pair!](cons (car [pair?]l1)
                         (append2 (cdr [pair?]l1) l2))))
```

at the type

$$\forall \alpha_1 \alpha_3 \ldots \alpha_k . \,(\mathtt{dyn}_{\alpha_2}\ \mathtt{dyn}_{\alpha_2}) \to \mathtt{dyn}_{\alpha_2}$$

where $\mathtt{dyn}_{\alpha_2}$ is the abbreviation

$$\mathtt{dyn}_{\alpha_2} \equiv \mu\alpha_2.(\alpha^{(k)})\mathtt{dyn}$$

Verification is by type recursion and is left for the reader (recall that α_2 in the dyn-type shown is the type of the second component of the pair-component of the dynamic type.) The completion of append2 just shown is polymorphically safe, since any object can be used at the second argument in a call, provided the object is sufficiently *tagged*, which can always be achieved without generating avoidable run-time errors. Moreover, it is clear that this *DTS* can be efficiently inferred by the simple methods of Chapter 6.

If we attempt a polymorphically safe translation of append2 into *ML* in the simple-mindedly uniform framework considered here, then we shall end up with a monomorphic, dynamic completion:

```
fun safe_append2 (l1, (l2, ())) =
    if (isnull (outrec l1)) then l2
    else
      inrec(in_pair (#1 (out_pair (outrec l1)),
                     safe_append2 (#2 (out_pair (outrec l1)),
                                   (l2, ()))))
```

at type

$$\mathtt{recty} * (\mathtt{recty} * \mathtt{unit}) \to \mathtt{recty}$$

Here we assume the tagging-based implementation of isnull? as

```
fun isnull (x, ()) = case x of
                       in_nil _ => true
                     | _        => false
```

at $(\alpha^{(k)})\mathtt{dyn} * \mathtt{unit} \to \mathtt{bool}$, in accordance with our discussion of universally generic predicates in Section 7.7.2.

If we imagine this translation of append2 stuck away in a program library, then we face the serious problem that optimization of list data representation (see the discussion in Section 7.6.2) becomes impossible. Ideally, we would wish a translation to be somehow instantiatable to a type specific implementation at type

$$\alpha\,\mathtt{list} * (\alpha\,\mathtt{list} * \mathtt{unit}) \to \alpha\,\mathtt{list}$$

Interestingly, there is a way to do this, via coercion parameterization. [18] The idea here is to perform an abstraction of the recursion coercion patterns of safe_append2, which amounts to considering the coercive-typed, parameterized *MLS*-completion

[18]This observation is due to Henglein.

```
(define (append2
        (Lambda (k1: 'a --> 'b * 'a)
        (Lambda (k2: 'b * 'c --> 'c)
                (lambda (l1 l2)
                        (if (null? l1)
                             l2
                           ([k1](cons (car ([k2] l1))
                                      (append k1 k2
                                             (cdr ([k2] l1))
                                              l2))))))))
```

at the type

$$\forall \alpha\beta\gamma.\, C \Rightarrow (\alpha\ \gamma) \rightarrow \gamma$$

with $C = \{\alpha \rightsquigarrow \beta * \alpha, \beta * \gamma \rightsquigarrow \gamma\}$. The *DTS*-completion shown above is an instance of this one. If we allow for abstraction over the predicate `null?` by a parameter `tst`, then we can write the equivalent *ML*-function:

```
fun abstract_append2 tst
                 (cons   : 'b * 'c -> 'c)
                 (uncons : 'a -> 'b * 'a)
                 (l1, (l2, ())) =
    if (tst l1) then
       l2
    else
       cons (#1 (uncons l1),
             abstract_append2 tst
                              cons
                              uncons
                              (#2 (uncons l1), (l2, ())))
```

at type

$$(\alpha \rightarrow \texttt{bool}) \rightarrow (\beta * \gamma \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta * \alpha) \rightarrow \alpha * (\gamma * \texttt{unit}) \rightarrow \gamma$$

We can instantiate this by application of the “representation-shift”- coercion functions `outlist`, `inlist`, `emptylist`, given by

```
fun emptylist x = (x = [])

fun inlist (x, y) = x :: y

fun outlist (x::y) = (x, y)
```

The instantiation

```
abstract_append2 emptylist cons uncons
```

lives at the “ideal” type (modulo equality type variables, due to the test in `emptylist`)

$$\alpha^{=}\ \texttt{list} * (\alpha^{=}\ \texttt{list} * \texttt{unit}) \rightarrow \alpha^{=}\ \texttt{list}$$

This example shows that coercion parameterization can be of absolutely vital importance for finding type-specific, optimized data-representations in a modular framework, which attacks the problem of polymorphic safety. The example also strongly suggests that it will be necessary, in a realistic framework, *either* to abandon the uniform implementation of universally generic procedures via tagging, *or* to find some manageable way to infer "additional" abstractions over the problematic functions. Perhaps one could uniformly abstract over all such functions, instantiating in each case with type-specific implementations. Parameter passing could be optimized by link-time partial evaluation. Another approach is to use a form of non-parametric polymorphism where types are passed and inspected at run-time. We mentioned these ideas earlier, in Section 7.6.1.

7.7.6 I/O

We consider how the central I/O-procedures for reading and writing ([CR91], section 6.10) can be handled in the translation. *Scheme*'s input/output ports should be hadled via *ML*'s streams. We add to the dynamic type the cases:

```
datatype ( ... )dyn = ...
                    | in_inputport  of instream
                    | in_outputport of outstream
```

Input/output poses design problems for the translation which are closely related to those encountered for other universally generic procedures (see Section 7.6.1, Section 7.7.2 and Section 7.7.3.) Although generic, the output procedures (`write`, `display`) essentially need to perform a type-directed recursive descent in the argument. And indeed, the problem of representing a general purpose *print-routine* is often considered a benchmark for frameworks which aim to extend statically typed languages with features of *dynamics* ([ACPP89], [LM91], [DRW95].) The discussion of Section 7.6.1 applies to this problem, and we refer the reader to it. In keeping with this discussion, we consider translation via dynamic type coercions.

The procedures `read` and `write` [19] effectlively convert between *external representations* [20] and run-time objects, denoted $\langle obj \rangle$. Procedure `read` is in effect a parser for the syntactic class $\langle datum \rangle$, taking an external representation of a $\langle datum \rangle$ in the input port to the corresponding run-time object, which we shall denote $\langle datum \rangle$. Procedure `write` writes the external representation of a run-time object to the port:

`read` : $\langle input\ port \rangle \rightarrow \langle datum \rangle$

`write` : $\langle obj \rangle \rightarrow \langle output\ port \rangle$

The procedures can be (informally) considered inversely related by

$$\texttt{write} \circ \texttt{read} = id$$

Now, *Lisp*-like languages such as *Scheme* are famous for (among other things) supporting the principle that "programs are data" which is so convenient in meta-programming. This principle can be succinctly expressed in terms of the formal syntax of *Scheme*, where it boils down to the fact: every string which parses as an $\langle expression \rangle$ also parses as a $\langle datum \rangle$ (the converse does not hold.) Hence, a `quote` on a program text encodes the program as a data-structure in the language. The I/O-procedures makes it necessary to implement a distinction between objects of category $\langle datum \rangle$ and other run-time objects, as is illustated in the following *Scheme* session:

[19] Procedure `display` is close enough to `write` to be left out here.

[20] See [CR91], section 3.3.

```
> (write (read))
(lambda (x) x)
> (lambda (x) x)
>
> (write (lambda (x) x))
> #<procedure>
```

The first argument to `write` is a ⟨*datum*⟩, whereas the second one is a closure.

We can implement this distinction by introduction of a representation of the run-time objects which are parsed as `datum`:

```
datatype ('a, 'b)datum = in_datum of (
                                      ('a, 'b)dyn,
                                      ('a, 'b)dyn,
                                      'a,
                                      'b,
                                      ('a, 'b)dyn
                                     )dyn
```

Objects of the type (α,β)`datum` are recursively tagged as such, with the exception of closures, which are run-time objects that are not of the category ⟨*datum*⟩; hence the parameterization over the function-space variables α, β. Closure values do not have a standard external representation, but the representation `#<procedure>` is normally used for all such objects.

The specialized, recursive form of type $(\ldots)$`dyn` is necessary (given the coercion-based framework for non-parametric operations, we have placed ourselves in here), because a function like `write` must perform a recursive descent in its object, universally over any object whatsoever (see [ACPP89], [LM91], [DRW95] for expositions of this problem in other, related frameworks.) The strategy outlined would lead to the typings:

`read` : `input-port` $\to$ `(...)datum`

`write` : `(...)datum` $\to$ `nil`

This scheme leads to the need for conversions between type `(...)datum` and type `(...)dyn`. For instance, the session

```
> (car (read))
(1 . 2)
> 1
```

would require the ⟨*datum*⟩ read to be transformed into the dynamic type (and eventually into a pair) before it is sent to procedure `car`. It would be possible to implement a recursive "coercion" (an *ML* conversion function) from `(...)datum` to the dynamic type. The converse transformation would be required before sending an object to (the translation of) `write`, because the print-routine must perform a recursive descent of an arbitrary run-time object. This would therefore be restricted to work on a highly specialized version of the dynamic type, unless the type restrictions in *ML* be violated, which requires the object to be wrapped into a uniformly tagged structure. The type `(...)datum` is uniform in this way, whereas an arbitrary object of some instance of the *polymorphic* dynamic type is not. Essentially, we would need an instance of the dynamic type isomorphic to the type `(...)datum` for a conversion from `(...)dyn` back to `(...)datum` to be type-correctly implementable in *ML*. This seems to lead to the need for special restrictions on the dynamic type, of the kind we met in considering the predicate `equal?` (Section 7.7.3.) Unfortunately, we feel that it is doubtful whether this can be realized without effectively ending up in a monomorphic inference system, for a large and important class of programs.

The dynamics outlined here for I/O should be kept clear of the dynamic system for run-time type-safety in *DTS*, in keeping with the abstraction principle (Section 7.5.) In principle, a similar approach could be taken to the translation problem; for instance, one might consider a non-parametric polymorphic *print*-routine (implementing `write`) which, rather than being guided by run-time type coercions, is directed by run-time type representations, in the style of [DRW95]. Probably, a satisfactory implementation of *Scheme*'s I/O would really require an extended target language containing some form of polymorphic *dynamics* ([ACPP89], [LM91].)

Although the R4RS document [CR91] does not require it, most *Scheme*-implementations contain the procedure `eval`, which takes an object $\langle obj \rangle$ to the result of evaluating $\langle obj \rangle$. The extension considered here via tagging coercions (perhaps, effectlively, a monomorphic system) would also enable us to give a translation of procedure `eval`, if we so wish.

7.7.7 `call/cc`

In a framework of restricted imperative polymorphism (cf. Section 7.4.1) it is safe ([Wri94a], [Wri93], see also [DHM93], [HL92], [HL93]) to type `call/cc` ([CR91], section 6.9) via *Pierce's axiom* :

`call/cc` $: \forall\alpha\beta.((\alpha \to \beta) \to \alpha) \to \alpha$

This type may be assumed for *DTS*. In framework, which does not obey the restriction of polymorphism to values, the type of continuation operations cannot be given unrestricted polymorphic types[21]

The continuation operator `call/cc` provides the single most clear example of an operation whose translation really requires an extension to *SML* as described in [MTH90]. In the absence of control operations in the target language, we would have to CPS-convert the program as part of the translation. This is of course totally unrealistic for the kind of translation we have in mind here. The operation is part of *SML of New Jearsey*, and such a dialect would have to be used as target, if we want to handle *Scheme*'s `call/cc`.

It is an interesting question whether the standard type of `call/cc` shown above is *polymorphically safe*. We have found no counter-examples showing it is not, but unfortunately, we must leave this question unresolved here. The discussion of `call/cc` in [Wri94] in the setting of soft typing is helpful.

7.7.8 Dynamic scope

Scheme has dynamic binding at top-level([CR91], section 3.1 and 6), and *ML* doesn't. A simulation of this feature could be made by abstracting over all identifiers at top-level. However, this would lead to a drastic decrease in typing presicion, and we suggest that this feature should be banned from *DTS*. We also feel that it should not be considered a major objective to support this feature in a translation, either. If it were to be part of a translation, then there ought to be a "switch" which would optionally enforce static binding. [22]

[21] As was pointed out in [HL92], where unsoundness of the system proposed in [DHM91] was discovered

[22] No doubt, the personal tastes of the author enter here; if the reader will allow us this single, very blunt personal confession, unrestricted dynamic binding is, in our view, one of the most crazy language features ever

7.8 Conclusion

Our work on dynamic type inference for a *Scheme*-like language has taken two directions, based on the observation that it may not *necessarily* be the case that a dynamic type inference system serving as the basis for static debugging and a system serving as the basis for a high-level compilation into a statically typed language like *ML* share exactly the same goals. We have codified this observation in the so-called "abstraction principle", according to which a dynamic inference system serving in the former capacity should model *just* the aspects of run-time type safety; in contrast, a system serving as basis for a high-level translation (or more generally, as basis for *compilation* of the language) may use run-time type coercions for *other* purposes than those of type safety. An example of this is the implementation of universally defined operations, such as type testing predicates. We did, however, speculate that this is not a choice which is forced upon us, and we mentioned that current interest in the use of *run-time types* and non-parametric polymorphism might well be a better choice. In the first direction mentioned above, we define the completion language *Dynamically Typed Scheme* (*DTS*) and in the second direction we define *ML-Scheme* (*MLS*). We now consider more specific points.

- We observe that a fully modular completion inference is needed for a faithful unification of dynamic and static typing as well as for the project of translating a dynamically typed language into a statically typed language. We argue that the use of coercion parameterization with coercive types is an important ingredient for achieving this goal. We identify the need for a form of *neededness-analysis* to support polymorphically safe inference of coercive types.
- We investigate *ML-Scheme* as a basis for a *Scheme*-to-*ML* translation. The difference between the two languages, *MLS* and *DTS*, is tiny at the core level (showing up only in the conditional) but this is not so outside the core. The main idea of *ML-Scheme* is to investigate the use of coercions to implement universally defined generic functions, such as the conditional, type testing predicates as well as strong equivalence predicates and I/O procedures. Our investigation lets us conclude that a totally uniform translation schema based on run-time coercions is possible for a substantial subset of *Scheme* but that this method will not be satisfactory in practice. A practical translation must address the problems of universal procedures, lists and type recursion with more sophisticated methods, and the use of run-time types may be an answer to some of these problems. Unfortunately, our treatment of this option remains at a purely speculative level, however. Also, we suggest the use of so-called `box`-types to delimit the use of assignment, since a completely uniform translation using *ML*-references would effectively eliminate polymorphism. This leads to the requirement for a `box`-analysis. We believe such an analysis to be feasible, but we have not defined one. The idea is inspired by Wright's treatment of the assignment problem in [Wri94]. We indicate that it may be possible to optimize data-representation (the use of *ML* `lists` being a prominent example of this) by a form of coercion parameterization, but we do not take this idea far enough for an inference method to be derivable from our investigation.
- Our development of *Dynamically Typed Scheme* has witnessed considerably fewer problems than our investigation of *ML-Scheme*. This is due to our adoptation of the abstraction principle, which entails that, for *DTS*, we can (indeed) abstract from many of the serious problems that had to be faced in *MLS*, since we can work with much better types for the

suggested. The restricted, dynamic binding of *Scheme* can perhaps be useful sometimes.

universal procedures, leaving the details of how we *implement* these operations aside. Note that this (meaningfully) is so, *even if* we were to choose a "tagging-based" implementation of *DTS*; this choice still would not affect the type system. The main desiderata discovered for a system supporting inference for *DTS* is a `box`-analysis to delimit the scope of imperative operations, in addition to the neededness analysis already mentioned.

Chapter 8

Conclusion

8.1 Summary and appraisal

This section lists what we feel are the *main* things achieved and things *not* achieved in this work.

The overall purpose set out for the present work was

1. To study the proof theoretic aspects of dynamic typing in order to see how far the methods introduced in [Hen94] can be pushed, preferably with solutions to the major open problems stated therein as part of the outcome.
2. To generalize the monomorphic framework of Henglein's dynamically typed λ calculus to include polymorphism.
3. To develop completion inference methods for a generalized framework of polymorphic dynamic typing and to study their behaviour in practice.
4. To investigate *modular* dynamic typing for a *Scheme*-like language and the possibility of using dynamic typing as a basis for high-level compilation of a *Scheme*-like language to *ML*.

In relation to each of the four tasks described above, we feel that the most important outcome of our work is the following:

8.1.1 Proof theory

The outcome of our proof theoretic investigation can be summarized in the following main points.

1. We have proven (Chapter 2) one "half" of Henglein's conjectures in [Hen94] correct, by proving that ϕ-reduction is confluent over λI-completions. This shows that the process of completion exhibits a surprising degree of order and that it is possible to reason formally about that process as a meaningful notion of static computation over run-time type operations.
2. We have shown (Chapter 5) that, for a generalized calculus of dynamic typing including discriminative polymorphic sums and type recursion (see Section 8.1.2 below), we have existence and uniqueness of normal forms under an oriented form of ϕ-reduction in so-called primitive completion classes. The most significant result is that this holds for the class $\mathcal{C}_{p*}$ where no restrictions are imposed on how primitives are placed in a completion. We feel that this could be a significant result with implications for practical purposes. We

have attempted to support this view with examples in several parts of the report. However, our termination proof is not very constructive, and the obvious constructivization of it is probably not computationally interesting. The result should be seen as indicating that it is, in theory, possible to extend the present framework in a possibly very promising way.

3. We have shown (Chapter 3) that, not so surpisingly, the cornerstones (minimality, safety, coherence) of an “ideal” theory can be developed rather easily for a natural subsystem suitable for modelling a form of quasi-static typing, where type assumptions are held fixed. The developement of Chapter 3 is really implicit in Henglein’s work [Hen94], since it is rather straight-forward to carry it out once the formal framework of [Hen94] is given. The development may perhaps be of some interest on its own, but our main motivation for including the investigation of Chapter 3 in this work was to demonstrate technically where the problems of the general calculi stem from and to understand, in detail, where and why the simpler techniques of Chapter 3 fail to generalize.

4. We have not, however, been able to decide the second “half” of the problems left open in [Hen94], which is the termination problem. We have also not been able to push the proof theoretic methods as far as we could have wished in other directions, such as regarding the formal concept of safety (cf. Chapter 3.) We have tried (Chapter 5) to explain more precisely why the termination problem is difficult, and we have demonstrated that (not so surprisingly) the source of difficuilty lies in unfolding of type recursion. A positive side of this is that we were able (Chapter 5) to prove termination of reduction relative to an arbitrary but fixed structure of type recursion. We restate Henglein’s conjecture of termination, with renewed conviction, on the basis of many experiments and some more general considerations in the final two sections of Chapter 5.
   Unless the problems mentioned here are due to shortcomings of the author (a very real possibility), these findings point to the conclusion that it may not be feasible to develop a completely “ideal” syntactic theory of dynamic typing based on proof theoretic methods. This is the more so, since we can in many cases get by with operational concepts and methods for practical purposes. Examples of this are found in our notions of operational safety.

8.1.2 Generalization of dynamic typing

The outcome of our attempt at generalizing Henglein’s dynamic typing can be summarized thus:

1. Starting from an equational presentation of catogorical co-products we have defined a generalized calculus Σ^{∞} of dynamic type operations (Chapter 4), which comprises polymorphic discriminative, tagged sums, recursive types and explicit error-coercions. We found that the introduction of the “absurd” equation $\mathbf{0} = \mathbf{1}$ leads to a theory in which a generalized form of ϕ-reduction can be defined. The calculus Σ^{∞} can be seen as an attempt to unify dynamic and soft typing, since it articulates the monomorphic dynamic type by means of the core technical ingredients of soft type systems, polymorphic sum types and recursion. Apart from lying in direct continuation of Henglein’s dynamic typing, our approach was particularly inspired by the Rice system of soft typing.

2. We have developed basic proof theory of the generalized calculus, taking advantage of stronger factorization properties due to the presence of error coercions, following a proposal in [Hen94]. Our results concerning existence and uniqueness of normal forms in primitive completion classes were shown (Chapter 5) to hold for the generalized system.

8.1.3 Completion inference

We have constructed and implemented a simple extension of Henglein's [Hen91] near-linear, constraint based completion inference algorithm, tuned to work for our generalized framework. Main points in relation to this part (Chapter 6) of our work are:

1. We have given a constraint based specification of the inference algorithm together with proofs of its correctness and a description of our prototype implementation of it. The development here is completely parallel to that of [Hen92] for the monomorphic system, and the algorithmic generalization of that work can be considered almost obvious, once the framework of Σ^∞ is given. In fact, we observe that the more powerful framework of Σ^∞ leads to a *simplification* of the constraint solving algorithm in several respects.
2. We observe that the inference system mentioned above as well as extant systems of soft type inference are inherently *non-modular* for reasons of operational safety. We show how safety and minimality of completions can be conflicting goals, and following ideas of [Hen92], we present how the introduction of *coercion parameterization* and *coercive types* leads to a form of qualified type system, which may be used to reach a compromise in this conflict.
3. On the basis of examples we give an impression of the current limitations of our simple, non-modular inference system regarded as a tool for static debugging of dynamically typed languages. We observe that our framework theoretically admits a number of pragmatically interesting extensions. Among these the passage to completion in the class $\mathcal{C}_{p*}$ is prominent. We have *not*, however, brought our implementation to the point where it can be tested on large scale programs.
4. We compare two systems of soft typing with the generalized dynamic type system presented in the chapter. We observe that our approach is most closely related to the soft type system developed at Rice, by Cartwright, Fagan and Wright. The system of Aiken, Wimmers and Lakshman can be regarded as an "approximation from above" towards the practical ideal, in terms of computational cost and perhaps also in terms of expressiveness; in contrast, the existence of highly efficient algorithms for our framework makes it possible to view this as an "approximation from below". Which approach is better in the very end, we do not know. We observe that our framework is in some respects more expressive and flexible than the Rice system, due to the fact that we model both of the dual run-time type operations, checking and tagging, and due to the fact that our framework potentially allows coercions to be inserted at arbitrary points; compare the results about the class $\mathcal{C}_{p*}$ in Section 8.1.1 above. We observe later (Chapter 7) that the possibility of passing to $\mathcal{C}_{p*}$-completions is important for achieving an interesting form of *modular* inference.

8.1.4 Dynamic typing for *Scheme* and translation to *ML*

We defined and investigated two completion languages, *Dynamically Typed Scheme* (*DTS*) and *ML-Scheme* (*MLS*).

1. Following suggestions in [Hen92] we argued the need for coercion parameterization with coercive types in order to achieve a fully modular completion inference. We identified a form of *neededness-analysis* to support safe polymorphic inference of coercive types.

2. Our development of *DTS* identified the need for a form of `box`-analysis in order to delimit the scope of imperative operations, to prevent polymorphism from being effectively eliminated. This view percieves `box`-operations as a special kind of "soft" coercions, comparable in status to explicit type recursion coercions. We developed *DTS* under the adoption of an "abstraction principle" which allowed us to seperate clearly concerns of language implementation from the type analysis. By contrast, we experienced that this separation kept us clear from a number of very difficult problems that were met in our investigation of *MLS*, where the abstraction principle was methodologically suspended.
3. Our development of *MLS* suggested that a totally uniform translation schema with a coercion-based implementation of universally generic procedures would be unsatisfactory in practice, once we try to go beyond the core of *Scheme*. We gave a list of features of full *Scheme* that would be especially difficult to handle in satisfactory way. We suggested the use of non-parametric polymorphic run-time types as an alternative, but these ideas remain at the speculative level in the present work.

8.2 Related work

8.2.1 Monomorphic dynamic typing

Henglein defined the monomorphic, dynamically typed λ-calculus [Hen92] [Hen92a] [Hen94]. He studied its completion inference problem ([Hen92], [Hen92a], see also [Hen91]), and he initiated the proof theoretic study of dynamic typing. The present work lies in direct continuation of Henglein's work, both regarding the basic framework, the fundamental ideas and the techniques employed. Henglein's works contain suggestions of several of the extensions attempted here. He identified the need for coercion parameterization in polymorphic extensions of his dynamic typing framework ([Hen92]). He suggested the use of an error type with explicit error-coercions ([Hen94].) He defined the notion of coercion polarity and realized that it could be made the basis of a well behaved orientation of ϕ-reduction ([Hen94]), thereby providing the most important single insight on which our confluence proofs are based. He posed the main proof theoretic questions of dynamic typing and proposed conjectures for their solutions, notably the problem of termination of ϕ-reduction, which remains unsolved, and he defined the formal notions of minimality and safety ([Hen94].) He gave efficient, constraint based completion inference algorithms ([Hen92], [Hen91]) and studied their complexity; our work on simple polymorphic completion inference descends directly from this work.

8.2.2 Partial and quasi-static typing

Thatte [Tha88] introduced partial types (see also [Tha94]) in order to address the problem of heterogeneous structures in statically typed programming languages. His partial typing is characterized by a subtype system generated from the non-structural *top type* axiom $\tau \leq \Omega$ which is extended via standard axioms of *structural subtyping* ([FM90].) Heterogeneous structures are typed by weakening (subtyping) the type of an object to Ω. This determines Ω as a *terminal type*, since an object cannot be projected out of Ω once it is injected. This may have the consequence that, *e.g.*, `hd (x::tl)` is not equivalent to `x`, since no destructive computation can be (type correctly) specified for `x` if `x` is injected into, say, a heterogeneous list. This is in fundamental distinction to the approach of dynamic typing which is characterized by the definability of a *universal type* ($\mathbf{D}$ in Henglein's monomorphic system, or $\mu\overline{\alpha}.\sum_{tc} tc(\overline{\alpha})$ in our system Σ^∞) into which any object can be injected via *positive* coercions and projected back via *negative* coercions. In dynamic typing, this gives rise to a *co*-variant type order, whereas the subtype system of partial typing is extended under the usual, contra-variant function space axiom. Moreover, partial typing is not *universal* since not every program has a type. The basic idea of partial typing is thus to extend conventional statically typed languages, in contrast to dynamic typing which provides static type information for dynamically typed programs.

Thatte posed the problem of type inference for partial typing in the form of a subtype constraint solution problem. The typability problem has been shown to be decidable, first by a doubly-exponential algorithm in [OW92] and then in polynomial time in [KPS92]. Partially typed programs are known to be strongly normalizing (by reduction to simply typed λ-calculus, [WOP94].)

Gomard proposed, in [Gom90], to use a type **untyped** to modify Milner's algorithm $\mathcal{W}$ to accept any program by injecting untypable program phrases into the type **untyped**. Like Thatte's Ω for partial typing, the type **untyped** type is a terminator. However, Gomards system is universal, accepting every syntactically correct program.

Thatte [Tha88] introduced *quasi-static* typing with the declared intention of mixing static and dynamic typing. He introduced the use of *type coercions* to represent the run-time type operations of tagging and checking and untagging. He employs a notion of *positive* and *negative* coercions, which, however, is different from the one introduced by Henglein and used in the present work; in [Tha88], positive (negative) coercions correspond roughly to dynamic type coercions which are built exlusively from positive (negative) primitives. In comparison with our calculus **Q** of quasi-static typing, Thatte's system does not allow displacement of coercions under flow-equations. In contrast to general dynamic typing (**D** or Σ^∞), Thatte's quasi-static typing assumes that programs carry fixed, explicit type declarations. Thatte [Tha88] also introduced the notions of *plausibility checking* and *convergence*. The first notion refers to a process of coercion reduction with the aim of discovering situations where run-time type errors are guaranteed to arise. The coercion reduction of dynamic typing is closely related to plausibility checking, and the formal notion of safety in [Hen94] was inspired by Thatte's semantical notion of convergence.

8.2.3 Soft typing

Apart from Henglein's monomorphic dynamic typing (Section 8.2.1) the systems of soft typing were the most important source of inspiration for our work. Having already given an extended comparison of dynamic and soft typing in Section 6.8 of this report, we confine ourselves to a very short treatment here and refer the reader to Section 6.8 for a more detailed account.

The notion of soft typing was developed at Rice, by Cartwright, Fagan and Wright [Fag90],[CF91], [CF92], [Wri94], [WrCa94] as a generic name for a type system with the properties of *universality* (every program has a type), *minimality* (a typed program has as few run-time checks as possible) and *expressiveness* (the type system is highly expressive wrt. heterogeneous structures and recursion over such.) The actual systems of soft typing developed at Rice analyses *Scheme*-programs, handling essentially full *Scheme* and are technically characterized by a form of unification-based subtyping together with polymorphic, discriminative unions and recursive types. Insertion of run-time type checks restores tybability at unification break-down for programs which are not statically typable. Dynamic type checks adhere to applications of primitive operations of the language, which come in pairs of checked and unchecked variants.

A highly expressive system of soft typing is described by Aiken, Wimmers and Lakshman in [AWL94], extending Hindley-Milner polymorphic typing with (non-discriminative) union types, intersection types and so-called conditional types (the latter for control flow dependency analysis.) The system has been defined and implemented for a functional higher order programming language (*FL*.) The inference system relies on recently developed techniques for solution of systems of set constraints [AW92], [AW93]. The techniques are, potentially, computationally very costly ([BGW93], [AKVW93]), and experimental evidence suggests that, at least at this time, the implemented system is slower than the Rice system; on the other hand, the type language of [AWL94] is more expressive.

Of the systems of soft typing, the approach of the present work was most directly influenced by the Rice system, which inspired us to integrate Henglein's dynamic type coercions with a type language comprising discriminative sums with regular recursive types. Unlike dynamic typing, the systems of soft typing assume all objects to be tagged at run-time, and optimization of run-time type operations is only done for run-time type checks. The Rice system remains essentially within the completion class $\mathcal{C}_{pf}$ (classified by the position of run-time type checks) whereas extensions to more powerful classes (prominently $\mathcal{C}_{p*}$) arise naturally in our framework. None of the works on soft typing deal with the problem of modular inference and polymorphic safety.

8.2.4 Dynamics

Work on *dynamics* [ACPP89], [LM91] arises from desire to integrate specialized forms of dynamic typing into (*ML*-like) statically typed languages, in order to provide statically typable means of expressing a particular range of functions which are not (practically) expressible in conventional statically typed languages. Prominent examples are structured I/O and other "universal" operations such as the `eval` function. Central to systems of dynamics is the extension of the conventional language with very powerful run-time type based constructs which allow an object to be annotated with a more or less complete run-time type description, together with operations for inspecting and dispatching on the run-time type information. Related ideas arise in other contexts, notably in a recent contribution [DRW95], directed at type inference of ad-hoc polymorphism via an explicit construct for structural pattern matching on run-time types.

In contrast, dynamic typing does not *extend* the source language with new constructs under programmer control, but aims at inferring run-time type completions of the (unextended) subject language. As noted Henglein [Hen94], the frameworks for dynamics are more expressive than that of dynamic typing (and, we add, including the present extension) but hardly feasible for automatic *inference* of run-type type completions, and they are more expensive to implement than the simpler dynamic type coercions.

8.2.5 Proof theory of higher order bounded polymorphism

Coercion based presentations of subtyping proofs have been widely used in the study of higher order bounded polymorphism, as expressed in descendants of Cardelli and Wegner's system **Fun** [CW85] and Ghelli's system $F_{\leq}$ [Ghe90].

The problem of semantic coherence arises in situations where interpretations are specified by induction on typings; one is interested in showing that all typing proofs resulting in the same typing judgement recieve the same interpretation, such that semantics is seen to be independent of typing. This problem was attacked in a coercion framework in [BCGS89] [BGS90] [BCGS91] [CG90].

The study of forms of cut-elimination in explicit, term-encoded subtyping proofs has been used extensively in Ghelli's study of $F_{\leq}$, using a coercion calculus to reason about proof terms.

Normalization of subtyping proofs is used to study completeness and minimal typing properties of type checking procedures in [Ghe90]. Pierce has used proof theoretic methods in [Pie91] in this direction also (note that the subtyping relation of $F_{\leq}$ is shown undecidable there.) Properties of weak normalization and confluence are used in [CG94] to establish decidability of an equational theory of $F_{\leq}$.

8.2.6 Binding time analysis

Binding time analysis is used in partial evaluation [GJS93] to determine which parts of a program admit evaluation at compile time. Type based binding time analysis is both logically, historically and practically closely related to dynamic typing. Logically, a type based binding time analysis employs a form completion process which inserts so-called *lift* operations to weaken the binding time description of an object to be *dynamic* (as opposed to *static*), meaning executable only at run-time. The lifting operation is reminiscent of a dynamic tagging coercion.

Historically, Gomards work on partial typing (see Section 8.2.2) grew out of his work on partial evaluation, and his partial typing was one of the sources of inspiration behind Henglein's work on dynamic typing (see [Hen94] for this appraisal.)

Practically, Henglein's fast constraint based algorithm for binding time analysis in [Hen91] has been used to implement the algorithm of [Hen92]. Also, it is used as the basis of our implementation of simple completion inference with sums and recursion, in the present work. Moreover, the basic flow-analysis of Bondorf and Jørgensen [BJ93a], [BJ93b] descends from [Hen91], and the analysis performed by our simple inference is very close to the one used by them. In fact, their analysis performs a form of dynamic type analysis in order to classify as *dynamic* (in the sense mentioned above) parts of a program which cannot be guaranteed to be type safe; this is to prevent the partial evaluator from executing (at compile time) source code which is type unsafe (and which could cause the evaluator to crash.)

8.2.7 Representation analysis for polymorphic languages

Representation analysis for polymorphic languages aims at optimizing the uniform data representation scheme which is used to implement polymorphism. In this scheme, the polymorphic parts of an argument to a polymorphic function are represented, at run-time, in *boxed* form (usually meaning that data are represented by a pointer to the actual data), thus providing the opportunity to implement the function by a single piece of code. However, if boxed data are later inspected by the program, the run-time system must effect a *representation shift* (*unboxing* or indirection) before this can happen.

The *boxing analysis* of Henglein and Jørgensen [HJ94] attempts to minimize the number of run-time representation shifts in a framework which is very closely related to that of dynamic typing. In boxing analysis, the run-time representation shift operations are viewed as coercions, called `box` and `unbox`, and the analysis can be formulated in terms of a completion calculus with notions of ϕ- and ψ-reductions under flow-equations with a polarization according to coercion polarities, in close correspondence to the dynamic calculus.

There are important structural differences between boxing analysis and dynamic type analysis. First, boxing analysis takes place in a statically typed (*ML*-like) programming language setting so the analysis can take advantage of type information about the source program. In particular, representation types arise by a form of annotation of the types of the underlying program; coercion polarity gives rise to an order on representation types, organising these as a finite lattice. Finiteness entails strong normalization of polarized completion reduction. In contrast, the corresponding dynamic type hierarchy contains infinite descending chains due to unfolding of type recursion, and therefore the general termination problem becomes much harder for dynamic typing. The style of argument which can be used to prove termination in the case of boxing analysis can (only) be used to prove acyclicity of reduction in dynamic typing. The work [HJ94] by Henglein and Jørgensen provided an important inspiration here.

Second, the problems of safety, so crucial to dynamic typing, do not arise in boxing analysis, because no equivalent of run-time type errors are found in boxing analysis. In this respect, our recursion coercions, **f** and **u** for recursion folding and unfolding, introduced for the system Σ^{μ} in the present work, are similar to the representation shift coercions of boxing analysis. The same holds for the coercions `box` and `unbox` proposed in Chapter 7 for delimiting the scope of imperative operations.

8.2.8 Safety analysis

Safety analysis, as initiated by Shivers [Shi91], and continued by Palsberg and Schwartzbach [PS95] aims at collecting type information for dynamically typed languages or, generally, for extensions of the untyped λ-calculus. The analysis classifies programs as to whether they can be guaranteed to be type safe or not. These analyses are based on flow analysis (a form of closure analysis) and they are inherently global. Recently, Palsberg and O'Keefe [PaOK95] have showed an interesting result to the effect that the programs accepted by safety analysis are precisely those programs which are typable in a very natural type system of recursive subtyping, composed of $\top$ (top), $\bot$ (bottom) and recursive types together with a subtyping relation. The result entails decidability (in cubic time) of recursive subtyping, and the techniques used to prove equivalence with safety analysis involve interesting translations between flow information and types. As is noted by the authors, the techniques might have implications for dynamic typing.

8.3 Future work

The present work suggests extensions in three main directions:

1. Large scale implementation of safe polymorphic type inference for a dynamically typed language like *Scheme*. Here, the major problems to be adressed appear to be to specify, to develop correctness proos for and to implement a good "neededness-analysis" to discover where coercion parameters can be eliminated. Second, a `box` analysis must be developed which will infer boxing and unboxing of possibly assigned objects. Third, a form of control flow analysis would be desirable to enable the system to learn from the use of type testing predicates. Fourth, in a final stage, it would be interesting to implement a form of link-time partial evaluation of coercion parameter instantiation.

2. Implementation of a high-level translation from a *Scheme*-like language to *ML*. In addition to the problems mentioned above, this enterprise will have to deal with complications of language implementation. A major problem area here appears to be the implementation of universally defined operations, like type testing predicates and structured I/O; our investigation suggests that a purely coercion based implementation schema is not likely to be satisfactory in practice. The idea of using run-time type information should be explored in detail. Another, quite crucial, problem area is the use of type-specific data-representations; a prominent example is the use of *ML* lists.

3. Development of more powerful completion inference algorithms. A major area of interest would be to realize the theoretical possibility of computing minimization of safe completions in the class $\mathcal{C}_{p*}$. Also, it would be interesting to investigate the possible use of cubic time flow-analysis, suitable for a global analysis framework of static debugging of a single, entire program. Recent results by Palsberg and O'Keefe (cf. Section 8.2.8) makes such a project especially attractive.

Appendix A

System D

In this appendix we recall the dynamically typed λ-calculus as first presented in [Hen92] and [Hen94]. We shall be brief, since the main intuitions underlying the formalization have been given already, in Chapter 1. Also, the reader is generally referred to Henglein's original presentations; in particular, [Hen94] gives a thorough introduction.

We shall introduce two formal systems, called **D** and **Q**, respectively. The system **D** is exactly Henglein's calculus [1] As is explained more carefully in Section 3.1 below, **Q** can be regarded as a restriction of system **D**, which *respects fixed type declarations* of programs. This makes **Q** an object worth of consideration to us, because it can be seen to be a natural generalization of Thatte's *Quasi-static typing* discipline within Henglein's framework. This point is further discussed in Chapter 3.

In general, a dynamically typed λ-calculus consists of

- A *pure term language*, which defines the source programs for dynamic typing. They contain no run-time type operations and are referred to as *pure terms*. Pure terms are what the programmer writes and sees before him in a dynamically typed language.
- A *coercion language*, which defines our formal model of the run-time type operations.
- A *completion language*, which defines our formal model of run-time typed programs. Completions arise from pure terms by insertion of coercions. Note that a pure term is a special completion containing no coercions at all.
- A *type language* and a *type system*, which defines the set of well typed completions. Rules are also given for assignment of type *signatures* to coercions, defining the set of well type coercions.
- A notion of *canonical coercions*
- A notion of *canonical completions*
- Equational theories and reduction relations on coercions, giving rise to the notions of *safe coercions* and *minimal coercions*
- Equational theories and reduction relations on completions, giving rise to the notions of *safe completions* and *minimal completions*

A dynamically typed λ-calculus is naturally divided in two: a *calculus of coercions* and a *calculus of completions.*

[1] The system is called $\lambda\Delta$ in [Hen94]

A.1 System D

Type Expressions

$$\tau ::= \mathbf{D} \mid \mathbf{B} \mid \tau \rightarrow \tau'$$

Coercions

$$\begin{array}{lll} c & ::= & \mathtt{F!} \mid \mathtt{F?} \mid \mathtt{B!} \mid \mathtt{B!} \mid id_\tau \\ & \mid & c \rightarrow c \mid c \circ c \end{array}$$

Completions

Completions are the *preterms* of system **D**. The type system of **D** to be defined later then defines the set of *well typed completions.* We presuppose the notions of a preterm of $\lambda^{\rightarrow}$ and well typed term of $\lambda^{\rightarrow}$.

$$M ::= x \mid \mathit{true} \mid \mathit{false} \mid \lambda x : \tau . M \mid M\ M \mid (\mathbf{if}\ M\ M\ M) \mid [c]M$$

Completions result from a process of inserting coercions into a *pure* λ*-term* (called the *underlying term* of the completion) in such a way that the completion is well typed in system **D** (although the underlying pure term may not be well typed in system $\lambda^{\rightarrow}$.) In other words, the *underlying term* or coercion erasure of a **D** completion is the pure λ-term obtained from the completion by erasing all coercions and all types on bound variables.

We think intuitively of type **Dynamic** (**D**) as a universal domain solving

$$\mathbf{D} \cong \mathbf{B} \oplus [\mathbf{D} \rightarrow \mathbf{D}]$$

with injection $\Phi : [\mathbf{D} \rightarrow \mathbf{D}] \rightarrow \mathbf{D}$ into the right component of **D** and projection $\Psi : \mathbf{D} \rightarrow [\mathbf{D} \rightarrow \mathbf{D}]$ from **D** onto the right component, such that $\Psi \circ \Phi = id$.

Conventions

We are going to introduce congruence and reduction relations on coercions and completions, defining the conversion theory and reduction theory of **D**. We introduce the following conventions :

- For a set of equational axioms A we let $A^{\rightarrow}$ denote the reduction relation induced by orienting the equations of A from left to right.
- If R is a relation on coercions we tacitly lift R to completions by stipulating $[c]M\ R\ [c']M$ if $c\ R\ c'$.
- If R is a reduction on completions we let $\Rightarrow_R$ denote $\rightarrow_R$ $(modE)$ where E is the equational theory on completions given below.
- For any reduction relation R we let $=_R$ (R-conversion) denote the congruence relation induced by R, by reflexive, symmetric, transitive, compatible closure on the appropriate structures (*e.g.*, coercions if R is a relation on coercions and completions if R is a relation on completions.)

A.2 Calculus of coercions

The coercion calculus is presented in Figure A.1 and Figure A.2.
The least set of coercions closed under the signature rules of Figure A.1 is the set of *well formed coercions*. If $c : \tau \rightsquigarrow \tau'$ then $\tau \rightsquigarrow \tau'$ is called the *signature* of c, and c is said to be a coercion *at* the signature $\tau \rightsquigarrow \tau'$. The coercions id, $\mathtt{F}!$, $\mathtt{F}?$, $\mathtt{B}!$, $\mathtt{B}?$ are called *primitive coercions*, and the others are called *induced coercions*.

Lemma A.2.1 *For every signature $\tau \rightsquigarrow \tau'$ there is a syntactically unique canonical coercion c such that $\vdash_\kappa c : \tau \rightsquigarrow \tau'$.*

Proof Induction on the size of the signature. □

Conventions

- We sometimes write c^σ_τ as meta-notation for a coercion c such that $c : \tau \rightsquigarrow \sigma$.
- We sometimes write κ^σ_τ for the unique canonical coercion with signature $\tau \rightsquigarrow \sigma$.

A.3 Calculus of completions

We define, for arbitrary types τ, the completion $[\![M]\!]\,\Gamma\,\tau$, by

$$[\![M]\!]\,\Gamma\,\tau \equiv [\kappa^\tau_{\mathbf{D}}][\![M]\!]\,\Gamma$$

As is easily seen one has

Lemma A.3.1 *For every pure term M and Γ with $FV(M) \subseteq \Gamma$:*

$$\Gamma \vdash [\![M]\!]\,\Gamma\,\tau : \tau$$

Note that, in particular,

$$\Gamma \vdash [\![M]\!]\,\Gamma : \mathbf{D}$$

for all M, Γ with $FV(M) \subseteq \Gamma$, and $C \vdash [\![M]\!]\,\Gamma = [\![M]\!]\,\Gamma\mathbf{D}$.
We call $[\![M]\!]\,\Gamma\,\tau$ *the canonical completion of M at τ wrt.* Γ. For any term M we let $\Gamma_{\mathbf{D}} = \{x : \mathbf{D} \mid x \in FV(M)\}$. The completion $[\![M]\!]$ given by

$$[\![M]\!] \equiv [\![M]\!]\,\Gamma_{\mathbf{D}}$$

is called just *the canonical completion at* $\mathbf{D}$ *of* M.
The definition of canonical completions is a constructive demonstration of the important fact that any term can be completed at any type:

Lemma A.3.2 *For every λ-term M, type τ and assumption set Γ with $FV(M) \subseteq \Gamma$ there is a syntactically unique canonical completion of M at τ wrt. Γ.*

Proof Structural induction on M, using Lemma A.2.1. □

For any term M we write $\tilde{M}$ to denote the λ-term which results from erasing all type annotations and coercions from M. For any terms M, N we write

$$M \cong N$$

if and only if $\tilde{M} \equiv \tilde{N}$, and we say that M and N are *completion congruent.*

Coercion signatures

$$id_\tau : \tau \rightsquigarrow \tau \qquad \mathtt{F!} : (\mathbf{D} \to \mathbf{D}) \rightsquigarrow \mathbf{D} \qquad \mathtt{F?} : \mathbf{D} \rightsquigarrow (\mathbf{D} \to \mathbf{D})$$

$$\mathtt{B!} : \mathbf{B} \rightsquigarrow \mathbf{D} \qquad \mathtt{B?} : \mathbf{D} \rightsquigarrow \mathbf{B}$$

$$\frac{c : \tau \rightsquigarrow \tau' \quad d : \tau' \rightsquigarrow \tau''}{d \circ c : \tau \rightsquigarrow \tau''} \qquad \frac{c : \tau \rightsquigarrow \tau' \quad d : \theta \rightsquigarrow \theta'}{c \to d : (\tau' \to \theta) \rightsquigarrow (\tau \to \theta')}$$

Canonical coercions

$$id_\tau : \tau \rightsquigarrow \tau \qquad \mathtt{F!} : (\mathbf{D} \to \mathbf{D}) \rightsquigarrow \mathbf{D} \qquad \mathtt{F?} : \mathbf{D} \rightsquigarrow (\mathbf{D} \to \mathbf{D})$$

$$\mathtt{B!} : \mathbf{B} \rightsquigarrow \mathbf{D} \qquad \mathtt{B?} : \mathbf{D} \rightsquigarrow \mathbf{B}$$

$$\frac{\kappa : \tau \rightsquigarrow \tau' \quad \kappa' : \theta \rightsquigarrow \theta'}{(\kappa \to \kappa') : (\tau' \to \theta) \rightsquigarrow (\tau \to \theta')} \qquad \frac{\kappa : \mathbf{D} \rightsquigarrow (\tau \to \tau')}{(\kappa \circ \mathtt{B!}) : \mathbf{B} \rightsquigarrow (\tau \to \tau')}$$

$$\frac{\kappa : (\tau \to \tau') \rightsquigarrow \mathbf{D}}{(\mathtt{B?} \circ \kappa) : (\tau \to \tau') \rightsquigarrow \mathbf{B}} \qquad \frac{\kappa : (\mathbf{D} \to \mathbf{D}) \rightsquigarrow (\tau \to \tau')}{(\kappa \circ \mathtt{F?}) : \mathbf{D} \rightsquigarrow (\tau \to \tau')}$$

$$\frac{\kappa : (\tau \to \tau') \rightsquigarrow (\mathbf{D} \to \mathbf{D})}{(\mathtt{F!} \circ \kappa) : (\tau \to \tau') \rightsquigarrow \mathbf{D}}$$

Coercion polarity

$$\mathtt{F!} : + \qquad \mathtt{F?} : - \qquad \mathtt{F?} \circ \mathtt{B!} : * \qquad \mathtt{B?} \circ \mathtt{F!} : *$$

$$\mathtt{B!} : + \qquad \mathtt{B?} : - \qquad \frac{c_1 : * \quad c_2 : -}{c_2 \circ c_1 : *}$$

$$\frac{c : + \qquad d : +}{c \circ d : +} \qquad \frac{c : - \qquad d : -}{c \circ d : -} \qquad \frac{c_1 : * \quad c_2 : +}{c_1 \circ c_2 : *}$$

$$\frac{c : - \qquad d : +}{c \to d : +} \qquad \frac{c : + \qquad d : -}{c \to d : -} \qquad \frac{c_1 : * \quad c_2 : *}{c_1 \circ c_2 : * \quad c_1 \to c_2 : *}$$

A positive (negative) coercion which is not trivial is said to be *properly* positive (negative). The identity coercions are both positive and negative, but not *properly* so: $id_\tau : +, -$

Figure A.1: Coercion signatures and canonical coercions

Coercion equality (C)

$$
\begin{array}{lcll}
c \circ (c' \circ c'') & = & (c \circ c') \circ c'' & [C1] \\
id_\tau \circ c & = & c & [C2] \\
c \circ id_\tau & = & c & [C3] \\
(c' \to d) \circ (c \to d') & = & (c \circ c') \to (d \circ d') & [C4] \\
id_\tau \to id_{\tau'} & = & id_{\tau \to \tau'} & [C5]
\end{array}
$$

A coercion c such that $C \vdash c = id$ is called a *trivial coercion.*

Coercion conversion ($\phi\psi$)

$$
\begin{array}{lcll}
\mathtt{F?} \circ \mathtt{F!} & = & id_{\mathbf{D} \to \mathbf{D}} & [\phi 1] \\
\mathtt{B?} \circ \mathtt{B!} & = & id_{\mathbf{B}} & [\phi 2] \\
\\
\mathtt{F!} \circ \mathtt{F?} & = & id_{\mathbf{D}} & [\psi 1] \\
\mathtt{B!} \circ \mathtt{B?} & = & id_{\mathbf{D}} & [\psi 2]
\end{array}
$$

Coercion reduction ($\phi\psi$)

The ϕ and ψ reductions on coercions are obtained by orienting the corresponding congruences and taking the result $mod\, C$, thus :

$$
\begin{array}{lcl}
\to_\phi & = & \phi^{\to}\, (mod\, C) \\
\to_\psi & = & \psi^{\to}\, (mod\, C) \\
\to_{\phi\psi} & = & (\phi\psi)^{\to}\, (mod\, C)
\end{array}
$$

Figure A.2: Coercion equality, conversion and reduction

Type system

$$\Gamma, x : \tau \vdash x : \tau \qquad [AX]$$

$$\Gamma \vdash \mathit{true} : \mathbf{B} \qquad [T]$$

$$\Gamma \vdash \mathit{false} : \mathbf{B} \qquad [F]$$

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau.M : \tau \to \sigma} \qquad [{\to}I]$$

$$\frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash M\,N : \sigma} \qquad [{\to}E]$$

$$\frac{\Gamma \vdash M : \mathbf{B} \quad \Gamma \vdash N : \tau \quad \Gamma \vdash N' : \tau}{\Gamma \vdash (\mathbf{if}\; M\; N\; N') : \tau} \qquad [IF]$$

$$\frac{\Gamma \vdash M : \tau \quad c : \tau \leadsto \sigma}{\Gamma \vdash [c]M : \sigma} \qquad [CO]$$

Canonical completions

For every pure term M we define the completion $[\![M]\!]\,\Gamma$ wrt. assumption set Γ containing type assumptions on all the free variables of M:

$$\begin{aligned}
[\![x]\!]\,\Gamma &\equiv [\kappa^{\mathbf{D}}_{\Gamma(x)}]x \\
[\![\mathit{true}]\!]\,\Gamma &\equiv [\mathtt{B!}]\mathit{true} \\
[\![\mathit{false}]\!]\,\Gamma &\equiv [\mathtt{B!}]\mathit{false} \\
[\![\lambda x.M]\!]\,\Gamma &\equiv [\mathtt{F!}](\lambda x : \mathbf{D}.[\![M]\!]\,\Gamma[x : \mathbf{D}]) \\
[\![M\;N]\!]\,\Gamma &\equiv ([\mathtt{F?}][\![M]\!]\,\Gamma)\;([\![N]\!]\,\Gamma) \\
[\![(\mathbf{if}\; M\; N\; P)]\!]\,\Gamma &\equiv (\mathbf{if}\;([\mathtt{B?}][\![M]\!]\,\Gamma)\;([\![N]\!]\,\Gamma)\;([\![P]\!]\,\Gamma))
\end{aligned}$$

Figure A.3: Type system and canonical completions

Completion equality (E)

$$
\begin{array}{lcll}
[c]M & = & [c']M \quad \text{if } C \vdash c = c' & [E1] \\
[id_\tau]\, M & = & M & [E2] \\
[c \circ d]\, M & = & [c][d]M & [E3] \\
[c^\sigma_\tau \to d]\, (\lambda x : \sigma.M) & = & \lambda x : \tau.[d]M\{x := [c^\sigma_\tau]x\} & [E4] \\
([c \to d]\, M)\, N & = & [d]\, (M\, ([c]\, N)) & [E5] \\
[c]\, (\mathbf{if}\ M\ N\ N') & = & (\mathbf{if}\ M\ ([c]\, N)\, ([c]\, N')) & [E6]
\end{array}
$$

Completion conversion ($\phi\psi$)

The congruences $=_\phi$, $=_\psi$ and $=_{\phi\psi}$ on completions are obtained by adding to E the lifts of the corresponding congruences on coercions to completions.

Completion reduction ($\phi\psi$)

The ϕ and ψ reductions on completions are obtained by lifting the corresponding reductions on coercions to completions and taking the result *mod* E, thus :

$$
\begin{array}{lcl}
\Rightarrow_\phi & = & \to_\phi \ (mod\, E) \\
\Rightarrow_\psi & = & \to_\psi \ (mod\, E) \\
\Rightarrow_{\phi\psi} & = & \to_{\phi\psi} \ (mod\, E)
\end{array}
$$

Figure A.4: Completion equality, conversion and reduction

Appendix B

Proofs

B.1 Proofs for coercion factorization

Proof of Lemma 2.1.12

$$C \vdash c = c' \, ifandonlyif \pi(c) \sim \pi(c')$$

PROOF The implication from right to left is trivial, by definition of $\sim$. For the implication from left to right, we first prove $\pi(\mathcal{L}) \sim \pi(\mathcal{R})$ for every equation $\mathcal{L} = \mathcal{R}$ among the C-axioms. The only non-trivial case is for $\mathcal{L} \equiv (c_1 \to c_2) \circ (d_1 \to d_2)$ and $\mathcal{R} \equiv (d_1 \circ c_1) \to (c_2 \circ d_2)$, *i.e.*for axiom $[C4]$. In this case we have, on the one hand

$$\begin{array}{rcl} \rho(\mathcal{L}) & = & \rho(c_1 \to c_2) :: \rho(d_1 \to d_2) \\ & = & (\rho(c_1)^R \Rightarrow id) :: (id \Rightarrow \rho(c_2)) :: (\rho(d_1)^R \Rightarrow id) :: (id \Rightarrow \rho(d_2)) \end{array}$$

and on the other hand

$$\begin{array}{rcl} \rho(\mathcal{R}) & = & (\rho(d_1 \circ c_1)^R \Rightarrow id) :: (id \Rightarrow \rho(c_2 \circ d_2)) \\ & = & ((\rho(d_1) :: \rho(c_1))^R \Rightarrow id) :: (id \Rightarrow (\rho(c_2) :: \rho(d_2))) \\ & = & (\rho(c_1)^R \Rightarrow id) :: (\rho(d_1)^R \Rightarrow id) :: (id \Rightarrow \rho(c_2)) :: (id \Rightarrow \rho(d_2)) \\ & \sim & (\rho(c_1)^R \Rightarrow id) :: (id \Rightarrow \rho(c_2)) :: (\rho(d_1)^R \Rightarrow id) :: (id \Rightarrow \rho(d_2)) \end{array}$$

where we used Lemma 2.1.11 (first part) at the last step, and $\rho(\mathcal{L}) \sim \rho(\mathcal{R})$ is seen to hold. From this we clearly have $\pi(\mathcal{L}) \sim \pi(\mathcal{R})$.

We now lift this result to closure under abritrary coercion contexts, *i.e.*, we claim

$$\pi(\mathbf{C}[\mathcal{L}]) \sim \pi(\mathbf{C}[\mathcal{R}])$$

for any coercion context $\mathbf{C}$ and any equation $\mathcal{L} = \mathcal{R}$ of the equational system C. Clearly, the claim follows from $\rho(\mathbf{C}[\mathcal{L}]) \sim \rho(\mathbf{C}[\mathcal{R}])$ for the special case where $\mathcal{L} = \mathcal{R}$ is axiom $[C4]$. We proceed by structural induction on $\mathbf{C}$. The base case where $\mathbf{C} \equiv []$ is covered above. For the inductive step, only the cases where $\mathbf{C} \equiv d \to \mathbf{C}'$ and $\mathbf{C} \equiv \mathbf{C}' \to d$ are non-trivial. We consider only the hardest case where $\mathbf{C} \equiv \mathbf{C}' \to c$, the other case is analogous. So consider

$$\begin{array}{rcl} \rho(\mathbf{C}[\mathcal{L}]) & = & \rho(\mathbf{C}'[\mathcal{L}] \to d) \\ & = & (\rho(\mathbf{C}'[\mathcal{L}]^R) \Rightarrow id) :: (id \Rightarrow \rho(d)) \end{array}$$

By induction we know that

$$\rho(\mathbf{C}'[\mathcal{L}]) \sim \rho(\mathbf{C}'[\mathcal{R}])$$

Hence, by Lemma 2.1.11 (third property), we also have

$$\rho(\mathbf{C}'[\mathcal{L}])^R \sim \rho(\mathbf{C}'[\mathcal{R}])^R$$

Hence, by Lemma 2.1.11 (second property)

$$\rho(\mathbf{C}'[\mathcal{L}])^R \Rightarrow id \sim \rho(\mathbf{C}'[\mathcal{R}])^R \Rightarrow id$$

Hence, in sum we have

$$\begin{array}{rcl} \rho(\mathbf{C}[\mathcal{L}]) & = & (\rho(\mathbf{C}'[\mathcal{L}]^R) \Rightarrow id) :: (id \Rightarrow \rho(d)) \\ & \sim & (\rho(\mathbf{C}'[\mathcal{R}]^R) \Rightarrow id) :: (id \Rightarrow \rho(d)) \\ & = & \rho(\mathbf{C}'[\mathcal{R}]) \rightarrow d \\ & = & \rho(\mathbf{C}[\mathcal{R}]) \end{array}$$

□

Proof of Theorem 2.1.13

In order to prove the theorem, we will need some lemmas.

Definition B.1.1 For coercion sequences q, s and a non-trivial prime coercion π, write

$$q \mapsto_\pi s$$

if and only if

- $q =< \pi, q_1, \ldots, q_n >$,
- π commutes with $q_1, \ldots, q_i$, and
- $s =< q_1, \ldots, q_i, \pi, q_{i+1}, \ldots, q_n >$.

So $q \mapsto_\pi s$ if and only if $q \sim s$ by commuting the leftmost coercion π in q with $q_1, \ldots, q_i$ successively. A corresponding relation can of course be defined where π occurs at the right end of q. □

Lemma B.1.2 *If $< \pi >:: q \sim s$ then there exists an r such that $q \sim r$ and $< \pi >:: r \mapsto_\pi s$.*

Proof By induction on n where $< \pi >:: q \sim^n s$, and the base case for $n = 0$ is trivial. So assume for the inductive step that $< \pi >:: q \sim^{n-1} t \sim^1 s$, for some t. By induction, there exists $p =< p_1, \ldots, p_n >$ such that $q \sim p$ and $< \pi >:: p \mapsto_\pi t$. This means that

$$t =< p_1, \ldots, p_i, \pi, p_{i+1}, \ldots, p_n >$$

with π commuting with $p_1, \ldots, p_i$. Now consider what the final step $t \sim^1 s$ could consist in:

- $t \sim s$ by $< \pi, p_{i+1} > \mapsto < p_{i+1}, \pi >$. Then clearly $< \pi >:: p \mapsto_\pi s$, and the claim holds true.

- $t \sim^1 s$ by $< p_i, \pi > \mapsto < \pi, p_i >$. Then clearly $< \pi >:: p \mapsto_\pi s$, and the claim holds true.
- $t \sim^1 s$ by $< p_k, p_{k+1} > \mapsto < p_{k+1}, p_k >$ for $k, k+1 \in \{1, \ldots, i\}$. In this case we must have

  $$q \sim p \sim < p_1, \ldots, p_k, p_{k+1}, \ldots, p_i, \ldots, p_n > \sim < p_1, \ldots, p_{k+1}, p_k, \ldots, p_i, \ldots, p_n >$$

  and since π commutes with $p_1, \ldots, p_i$ we also have

  $$< \pi >::< p_1, \ldots, p_{k+1}, p_k, \ldots, p_i, \ldots, p_n > \mapsto_\pi$$

  $$< p_1, \ldots, p_{k+1}, p_k, \ldots, p_i, \pi, p_{i+1}, \ldots, p_n >= s$$

  so we can choose $r =< p_1, \ldots, p_{k+1}, p_k, \ldots, p_i, \ldots, p_n >$.
- $t \sim^1 s$ by $< p_k, p_{k+1} > \mapsto < p_{k+1}, p_k >$ with $k, k+1 \in \{i+1, \ldots, n\}$. In this case we must have

  $$q \sim p \sim < p_1, \ldots, p_i, \ldots, p_k, p_{k+1}, \ldots, p_n > \sim < p_1, \ldots, p_i, \ldots, p_{k+1}, p_k, \ldots, p_n >$$

  and since π commutes with $p_1, \ldots, p_i$ we also have

  $$< \pi >::< p_1, \ldots, p_i, \ldots, p_{k+1}, p_k, \ldots, p_n > \mapsto_\pi$$

  $$< p_1, \ldots, p_i, \pi, p_{i+1}, \ldots, p_{k+1}, p_k, \ldots, p_n >= s$$

  so we can choose $r =< p_1, \ldots, p_i, \ldots, p_{k+1}, p_k, \ldots, p_n >$.

□

Lemma B.1.3 *If $< \pi >:: q \sim < \pi >:: r$ then $q \sim r$.*

Proof By assumption and Lemma B.1.2, there must exist p such that $q \sim p$ and $< \pi >:: p \mapsto_\pi < \pi >:: r$. If $< \pi >:: p =< \pi >:: r$ we are finished. So assume $< \pi >:: p \neq < \pi >:: r$. Then, since $< \pi >:: p \mapsto_\pi < \pi >:: r$, it must be the case that $r =< r_1, \ldots, r_n >$ with $r_1 \equiv \pi$ and π commuting with r_1. But this is impossible, since no non-trivial prime commutes with itself. □

Lemma B.1.4 *Let s, p, q be sequences of non-trivial prime coercions. Then it holds: If $s :: p \sim s :: q$ then $p \sim q$, and if $p :: s \sim q :: s$ then $p \sim q$.*

Proof We prove only the first implication, since the second is analogous.

Proof is by induction on the length of s, and the case where s is empty is trivial. So assume, for the inductive step, that $s = t ::< s_n >$. Then, by assumption, $t ::< s_n >:: p \sim t ::< s_n >:: q$. By inductive hypothesis, $< s_n >:: p \sim < s_n >:: q$. Hence, by Lemma B.1.3, we have $p \sim q$. □

Definition B.1.5 If s, s' are coercion sequences of the same length n we write $C \vdash s = s'$ if

$$C \vdash s_i = s'_i \text{ for all } i = 1 \ldots n$$

□

Theorem B.1.6 *(C-Factoring)*
Let $\diamond$ be any fixed polarity, $+$ or $-$. Suppose either

1. $C \vdash < c_1, \ldots c_n > = < h^\diamond \circ c'_1, \ldots, h^\diamond \circ c'_n > = < g^\diamond \circ c''_1, \ldots, g^\diamond \circ c''_n >$, *or*
2. $C \vdash < c_1, \ldots, c_n > = < c'_1 \circ h^\diamond, \ldots, c'_n \circ h^\diamond > = < c''_1 \circ g^\diamond, \ldots, c''_n \circ g^\diamond >$

Then there exist coercions $\tilde{c}_1, \ldots, \tilde{c}_n, \tilde{h}^\diamond, \tilde{g}^\diamond$ *such that if* 1. *is the case then*

i $C \vdash < c'_1, \ldots, c'_n > = < \tilde{h}^\diamond \circ \tilde{c}_1, \ldots, \tilde{h}^\diamond \circ \tilde{c}_n >$

ii $C \vdash < c''_1, \ldots, c''_n > = < \tilde{g}^\diamond \circ \tilde{c}_1, \ldots, \tilde{g}^\diamond \circ \tilde{c}_n >$

iii $C \vdash h^\diamond \circ \tilde{h}^\diamond = g^\diamond \circ \tilde{g}^\diamond$

and if 2. *is the case then*

i $C \vdash < c'_1, \ldots, c'_n > = < \tilde{c}_1 \circ \tilde{h}^\diamond, \ldots, \tilde{c}_n \circ \tilde{h}^\diamond >$

ii $C \vdash < c''_1, \ldots, c''_n > = < \tilde{c}_1 \circ \tilde{g}^\diamond, \ldots, \tilde{c}_n \circ \tilde{g}^\diamond >$

iii $C \vdash \tilde{h}^\diamond \circ h^\diamond = \tilde{g}^\diamond \circ g^\diamond$

Proof We prove only the claim in case 1. since the second part is symmetric with analogous proof. It is easy to see that lemmas analogous to the ones used below hold by analogous proofs.

By assumption and Lemma 2.1.12 we have

$$\pi(c_i) \sim \pi(h^\diamond \circ c'_i) \sim \pi(g^\diamond \circ c''_i)$$

Now write, for every i,

$$\Pi^i = \pi(c_i) = < \pi^i_1, \ldots, \pi^i_{n_i} >$$

Also, let

$$H^\diamond = \pi(h^\diamond) = < h^\diamond_1, \ldots, h^\diamond_m >$$

so that $h^\diamond = [\circ]\, H^\diamond$, and let

$$G^\diamond = \pi(g^\diamond) = < g^\diamond_1, \ldots, h^\diamond_l >$$

so that $g^\diamond = [\circ]\, G^\diamond$.

By our assumptions, then, there must be indices $j_1, \ldots, j_m$ such that $\pi^i_{j_1}, \ldots, \pi^i_{j_m}$ can be commuted to the front of Π^i, with $\pi^i_{j_x} \equiv h^\diamond_x$ for $x = 1, \ldots, m$. That is, we must have

$$\Pi^i = < \pi^i_1, \ldots, \pi^i_{j_1}, \ldots, \pi^i_{j_2}, \ldots, \pi^i_{j_m}, \ldots, \pi^i_{n_i} >$$

where

$$\begin{array}{lll}
\pi^i_{j_1} & \text{commutes with} & \pi^i_1, .., \pi^i_{j_1-1} \\
\pi^i_{j_2} & \text{commutes with} & \pi^i_1, .., \pi^i_{j_1-1}, \pi^i_{j_1+1}, .., \pi^i_{j_2-1} \\
\vdots & \vdots & \vdots \\
\pi^i_{j_m} & \text{commutes with} & \pi^i_1, .., \pi^i_{j_1-1}, \pi^i_{j_1+1}, .., \pi^i_{j_2-1}, \pi^i_{j_2+1}, .., \pi^i_{j_{(m-1)}-1}, \pi^i_{j_{(m-1)}+1}, .., \pi^i_{j_m-1}
\end{array}$$

and with $\pi^i_{j_x} \equiv h^\diamond_x$ for $x = 1, \ldots, m$.

Also, we must have

$$\Pi^i = < \pi^i_1, \ldots, \pi^i_{k_1}, \ldots, \pi^i_{k_2}, \ldots, \pi^i_{k_l}, \ldots, \pi^i_{n_i} >$$

with $\pi^i_{k_x} \equiv g^\diamond_x$ for $x = 1, \ldots, l$, and such that

$$\pi^i_{k_x} \quad \text{commutes with} \quad \pi^i_1, .., \pi^i_{k_x-1}, \pi^i_{k_{(x-1)}+1}, .., \pi^i_{k_{(x-1)}+1}$$

as before.

Here it is a possibility that $k_x = j_y$ for some $x \in \{1, \ldots, l\}$, $y \in \{1, \ldots, m\}$. It follows from the commutativity properties that the following property holds:

$$\Pi^i \sim < \pi^i_{j_1}, \ldots, \pi^i_{j_m}, \ldots, \pi^i_{k_{(x_1)}}, \ldots, \pi^i_{k_{(x_p)}} > :: P^i$$

$$\sim < \pi^i_{k_1}, \ldots, \pi^i_{k_l}, \ldots, \pi^i_{j_{(y_1)}}, \ldots, \pi^i_{j_{(y_q)}} > :: P^i$$

for some sequence P^i, where

- $\pi(c'_i) =< \pi^i_{k_{x_1}}, \ldots, \pi^i_{k_{x_p}} >$, $\pi(c''_i) =< \pi^i_{j_{y_1}}, \ldots, \pi^i_{j_{y_q}} >$, and
- the $\pi^i_{k_{x_s}}$, $s \in \{1, \ldots, p\}$, are just those factors among $\pi^i_{k_1}, \ldots, \pi^i_{k_l}$ which have not been chosen among the $\pi^i_{j_1}, \ldots, \pi^i_{j_m}$ (*i.e.*, the factors used to build $G^\diamond$ which have not been chosen to build $H^\diamond$.) Alternatively, $k_{x_1}, \ldots, k_{x_p}$ are those indices among $k_1, \ldots, k_l$ such that for any k_{x_r}, $r \in \{1, \ldots, p\}$, we have $k_{x_r} \notin \{j_1, \ldots, j_m\}$.
- dually, the $\pi^i_{j_{y_t}}$, $t \in \{1, \ldots, q\}$, are just those factors among $\pi^i_{j_1}, \ldots, \pi^i_{j_m}$ which have not been chosen among the $\pi^i_{k_1}, \ldots, \pi^i_{k_l}$ (*i.e.*, the factors used to build $H^\diamond$ which have not been chosen to build $G^\diamond$.) Alternatively, $j_{y_1}, \ldots, j_{y_q}$ are those indices among $j_1, \ldots, j_m$ such that for any j_{y_r}, $r \in \{1, \ldots, q\}$, we have $j_{y_r} \notin \{k_1, \ldots, k_l\}$.

Now put

$\tilde{G}^\diamond_i =< \pi^i_{k_{x_1}}, \ldots, \pi^i_{k_{x_p}} >$

$\tilde{H}^\diamond_i =< \pi^i_{j_{y_1}}, \ldots, \pi^i_{j_{y_q}} >$

Then, since

$$\Pi^i \sim H^\diamond :: \tilde{G}^\diamond_i :: P^i \sim G^\diamond :: \tilde{H}^\diamond_i :: P^i$$

we get from Lemma B.1.4 (second implication) that

$$H^\diamond :: \tilde{G}^\diamond_i \sim G^\diamond :: \tilde{H}^\diamond_i$$

Now, by construction, we have $\pi^i_{j_x} \equiv h^\diamond_x$, for $x = 1, \ldots, m$, and $\pi^i_{k_y} \equiv g^\diamond_y$, for $y = 1, \ldots, l$. It follows that, for all $s, t \in \{1, \ldots, n\}$ we have $\tilde{G}^\diamond_s = \tilde{G}^\diamond_t$ and $\tilde{H}^\diamond_s = \tilde{H}^\diamond_t$. Therefore we can define

$\tilde{h}^\diamond \equiv [\circ]\, \tilde{G}^\diamond_i$ for any $i \in \{1, \ldots, n\}$

$\tilde{g}^\diamond \equiv [\circ]\, \tilde{H}^\diamond_i$ for any $i \in \{1, \ldots, n\}$

$\tilde{c}_i \equiv [\circ]\, P^i$

Then, by previous relations,

i $c'_i = [\circ]\, (\tilde{G}^\diamond_i :: P^i) = \tilde{g}^\diamond \circ \tilde{c}_i$

ii $c''_i = [\circ]\, (\tilde{H}^\diamond_i :: P^i) = \tilde{h}^\diamond \circ \tilde{c}_i$

iii $h^\diamond \circ \tilde{h}^\diamond = [\circ]\, (H^\diamond :: \tilde{G}^\diamond) = [\circ]\, (G^\diamond :: \tilde{H}^\diamond) = g^\diamond \circ \tilde{g}^\diamond$

□

B.2 Proof of canonicity lemma

We are to prove:
For all Σ*-types* τ, θ, η

$$\Sigma(\phi\psi) \vdash \kappa(\tau,\theta) \circ \kappa(\eta,\tau) \to^{*} \kappa(\eta,\theta)$$

Proof The proof is by induction on the generation of $\kappa(\tau,\theta)$, $\kappa(\eta,\tau)$. We use (prove by induction in types) that

$$\updownarrow^{\theta}_{\tau} \to^{*}_{\psi} \kappa\tau,\theta$$

We proceed by cases x, y over the rules used to generate $\kappa(\tau,\theta)$, $\kappa(\eta,\tau)$, respectively. All cases $(1,y)$ and $(x,1)$ are trivial and left out.

Case $(2,2)$.

$$\begin{array}{ll}
\kappa(\tau,\theta) \circ \kappa(\eta,\tau) & = \\
\updownarrow \circ \updownarrow & \to^{*} \\
\updownarrow^{\theta}_{\eta} & \to^{*} \\
\kappa(\eta,\theta) &
\end{array}$$

Case $(2,3)$.

$$\begin{array}{ll}
\kappa(\tau,\theta) \circ \kappa(\eta,\tau) & = \\
\kappa(F_i\overline{\tau}, F_j\overline{\theta}) \circ \kappa(F_i\overline{\eta}, F_i\overline{\tau}) & \to \\
\updownarrow^{F_j\overline{\theta}}_{F_i\overline{\tau}} \circ \kappa(F_i\overline{\eta}, F_i\overline{\tau}) & \to \\
\updownarrow^{F_j\overline{\theta}}_{F_i\overline{\eta}} & = \\
\kappa(F_i\overline{\eta}, F_j\overline{\theta}) &
\end{array}$$

Case $(2,4)$. This case is impossible, by signatures.
Case $(2,5)$.

$$\begin{array}{ll}
\kappa(\tau,\theta) \circ \kappa(\eta,\tau) & = \\
\kappa(F_i\overline{\tau}, F_j\overline{\theta}) \circ \kappa(\sum_k F_k\overline{\eta}, F_i\overline{\tau}) & \to \\
\updownarrow^{F_j\overline{\theta}}_{\Sigma} & \to \\
F_j(\updownarrow^{\theta_1}_{\eta_1}, \ldots, \updownarrow^{\theta_m}_{\eta_m}) \circ F^{?} & \to^{*} \\
F_j(\kappa(\eta_1,\theta_1), \ldots, \kappa(\eta_m,\theta_m)) \circ F^{?} & = \\
\kappa(\sum_k F_k\overline{\eta}, F_j\overline{\theta}) &
\end{array}$$

Case $(2,6)$. This case is impossible, by signatures.
Case $(3,2)$.

$$\begin{array}{ll}
\kappa(\tau,\theta) \circ \kappa(\eta,\tau) & \to \\
\updownarrow^{F_i\overline{\theta}}_{F_j\overline{\eta}} & = \\
\kappa(F_j\overline{\eta}, F_i\overline{\theta}) &
\end{array}$$

Case $(3,3)$.

$$\begin{array}{lll}
\kappa(\tau,\theta) \circ \kappa(\eta,\tau) & = & \\
F_i(\kappa(\tau_1,\theta_1), \ldots, \kappa(\tau_m,\theta_m)) \circ F_i(\kappa(\eta_1,\tau_1), \ldots, \kappa(\eta_m,\tau_m)) & = & \\
F_i(\kappa(\tau_1,\theta_1) \circ \kappa(\eta_1,\tau_1), \ldots, \kappa(\tau_m,\theta_m) \circ \kappa(\eta_m,\tau_m)) & \to^{*} & \text{induction} \\
F_i(\kappa(\eta_1,\theta_1), \ldots, \kappa(\eta_m,\theta_m) & = & \\
\kappa(F_i\overline{\eta}, F_i\overline{\theta}) & &
\end{array}$$

Case $(3,4)$. This case is impossible, by signatures.

Case $(3,5)$.

$$\begin{array}{lll}
\kappa(\tau,\theta)\circ\kappa(\eta,\tau) & = & \\
\kappa(F_i\overline{\tau},F_i\overline{\theta})\circ\kappa(\sum_k F_k\overline{\eta},F_i\overline{\tau}) & = & \\
F_i(\kappa(\tau_1,\theta_1),\ldots,\kappa(\tau_m,\theta_m)\circ F_i(\kappa(\eta_1,\tau_1),\ldots,\kappa(\eta_m,\tau_m)\circ F_i^? & = & \\
F_i(\kappa(\tau_1,\theta_1)\circ\kappa(\eta_1,\tau_1),\ldots,\kappa(\tau_m,\theta_m)\circ\kappa(\eta_m,\tau_m)) & \rightarrow^* & \text{induction} \\
F_i(\kappa(\eta_1,\theta_1),\ldots,\kappa(\eta_m,\theta_m))\circ F_i^? & = & \\
\kappa(\sum_k F_k\overline{\eta},F_i\overline{\theta}) & &
\end{array}$$

Case $(3,6)$. This case is impossible, by signatures.
Case $(4,2)$.

$$\begin{array}{ll}
\kappa(\tau,\theta)\circ\kappa(\eta,\tau) & = \\
\kappa(F_j\overline{\tau},\sum_k F_k\overline{\theta})\circ\kappa(F_i\overline{\eta},F_j\overline{\tau}) & = \\
\kappa(F_j\overline{\tau},\sum_k F_k\overline{\theta})\circ\updownarrow & \rightarrow \\
\updownarrow^{\sum}_{F_i\overline{\eta}} & \rightarrow^* \\
\kappa(F_i\overline{\eta},\sum_k F_k\overline{\theta}) &
\end{array}$$

Case $(4,3)$.

$$\begin{array}{ll}
\kappa(\tau,\theta)\circ\kappa(\eta,\tau) & = \\
\kappa(F_j\overline{\tau},\sum_k F_k\overline{\theta})\circ\kappa(F_j\overline{\eta},F_j\overline{\tau}) & = \\
F_j^!\circ F_j(\kappa(\tau_1,\theta_1),\ldots,\kappa(\tau_m,\theta_m))\circ F_j(\kappa(\eta_1,\tau_1),\ldots,\kappa(\eta_m,\tau_m)) & = \\
F_j^!\circ F_j(\kappa(\tau_1,\theta_1)\circ\kappa(\eta_1,\tau_1),\ldots,\kappa(\tau_m,\theta_m)\circ\kappa(\eta_m,\tau_m)) & \rightarrow^* \\
F_j^!\circ F_j(\kappa(\eta_1,\theta_1),\ldots,\kappa(\eta_m,\theta_m)) & = \\
\kappa(F_j\overline{\eta},\sum_k F_k\overline{\theta}) &
\end{array}$$

Case $(4,4)$. This case is impossible, by signatures.
Case $(4,5)$.

$$\begin{array}{lll}
\kappa(\tau,\theta)\circ\kappa(\eta,\tau) & = & \\
\kappa(F_j\overline{\tau},\sum_i F_i\overline{\theta})\circ\kappa(\sum_i F_i\overline{\eta},F_j\overline{\tau}) & = & \\
F_j^!\circ F_j(\kappa(\tau_1,\theta_1),\ldots,\kappa(\tau_m,\theta_m))\circ F_j(\kappa(\eta_1,\tau_1),\ldots,\kappa(\eta_m,\tau_m))\circ F_j^? & = & \\
F_j^!\circ F_j(\kappa(\tau_1,\theta_1)\circ\kappa(\eta_1,\tau_1),\ldots,\kappa(\tau_m,\theta_m)\circ\kappa(\eta_m,\tau_m))\circ F_j^? & \rightarrow^* & \text{induction} \\
F_j^!\circ F_j(\kappa(\eta_1,\theta_1),\ldots,\kappa(\eta_m,\theta_m))\circ F_j^? & = & \\
F_j^!\circ\kappa(F_j\overline{\eta},F_j\overline{\theta})\circ F_j^? & = & \text{by } [C10] \\
(\sum_i\kappa(F_i\overline{\eta},F_i\overline{\theta}))\circ F_j^!\circ F_j^? & \rightarrow & \\
\sum_i\kappa(F_i\overline{\eta},F_i\overline{\theta}) & = & \\
\kappa(\sum_i F_i\overline{\eta},\sum_i F_i\overline{\theta}) & &
\end{array}$$

Case $(4,6)$. This case is impossible, by signatures.
case $(5,2)$. This case is impossible, by signatures.
Case $(5,3)$. This case is impossible, by signatures.
Case $(5,4)$.

$$\begin{array}{lll}
\kappa(\tau,\theta)\circ\kappa(\eta,\tau) & = & \\
\kappa(\sum_i F_i\overline{\tau},F_j\overline{\theta})\circ\kappa(F_j\overline{\eta},\sum_i F_i\overline{\tau}) & = & \\
F_j(\kappa(\tau_1,\theta_1),\ldots,\kappa(\tau_m,\eta_m))\circ F_j^?\circ F_j^!\circ F_j(\kappa(\eta_1,\tau_1),\ldots,\kappa(\eta_m,\tau_m)) & \rightarrow & \\
F_j(\kappa(\tau_1,\theta_1),\ldots,\kappa(\tau_m,\eta_m))\circ F_j(\kappa(\eta_1,\tau_1),\ldots,\kappa(\eta_m,\tau_m)) & = & \\
F_j(\kappa(\tau_1,\theta_1)\circ\kappa(\eta_1,\tau_1),\ldots,\kappa(\tau_m,\theta_m)\circ\kappa(\eta_m,\tau_m)) & \rightarrow^* & \text{induction} \\
F_j(\kappa(\eta_1,\theta_1)\circ\kappa(\eta_m,\theta_m)) & = & \\
\kappa(F_j\overline{\eta},F_j\overline{\theta}) & &
\end{array}$$

Case $(5,5)$. This case is impossible, by signatures.
Case $(5,6)$.

$$\begin{array}{lll}
\kappa(\tau,\theta)\circ\kappa(\eta,\tau) & = & \\
\kappa(\sum_i F_i\overline{\tau}, F_j\overline{\theta})\circ\kappa(\sum_i F_i\overline{\eta}, \sum_i F_i\overline{\tau}) & = & \\
F_j(\kappa(\tau_1,\theta_1),\ldots,\kappa(\tau_m,\theta_m))\circ F_j^{?}\circ\sum_i F_i(\kappa(\eta_1,\tau_1),\ldots,\kappa(\eta_m,\tau_m)) & = & \text{by } [C11] \\
F_j(\kappa(\tau_1,\theta_1),\ldots,\kappa(\tau_m,\theta_m))\circ F_j(\kappa(\eta_1,\tau_1),\ldots,\kappa(\eta_m,\tau_m))\circ F_j^{?} & = & \\
F_j(\kappa(\tau_1,\theta_1)\circ\kappa(\eta_1,\tau_1),\ldots,\kappa(\tau_m,\theta_m)\circ\kappa(\eta_m,\tau_m))\circ F_j^{?} & \to^* & \text{induction} \\
F_j(\kappa(\eta_1,\theta_1),\ldots,\kappa(\eta_m,\theta_m))\circ F_j^{?} & = & \\
\kappa(\sum_i F_i\overline{\eta}, F_j\overline{\theta}) & &
\end{array}$$

Case $(6,2)$. This case is impossible, by signatures.
Case. $(6,3)$. This case is impossible, by signatures.
Case $(6,4)$.

$$\begin{array}{lll}
\kappa(\tau,\theta)\circ\kappa(\eta,\tau) & = & \\
\kappa(\sum_i F_i\overline{\tau}, \sum_i F_i\overline{\theta})\circ\kappa(F_j\overline{\eta}, \sum_i F_i\overline{\tau}) & = & \\
(\sum_i F_i(\kappa(\tau_1,\theta_1),\ldots,\kappa(\tau_m,\theta_m)))\circ F_j^{!}\circ F_j(\kappa(\eta_1,\tau_1),\ldots,\kappa(\eta_m,\tau_m)) & = & \text{by } [C10] \\
F_j^{!}\circ F_j(\kappa(\tau_1,\theta_1),\ldots,\kappa(\tau_m,\theta_m))\circ F_j(\kappa(\eta_1,\tau_1),\ldots,\kappa(\eta_m,\tau_m)) & = & \\
F_j^{!}\circ F_j(\kappa(\tau_1,\theta_1)\circ\kappa(\eta_1,\tau_1),\ldots,\kappa(\tau_m,\theta_m)\circ\kappa(\eta_m,\tau_m)) & \to^* & \text{induction} \\
F_j^{!}\circ F_j(\kappa(\eta_1,\theta_1),\ldots,\kappa(\eta_m,\theta_m)) & = & \\
\kappa(F_j\overline{\eta}, \sum_i F_i\overline{\theta}) & &
\end{array}$$

Case $(6,5)$. This case is impossible, by signatures.
Case $(6,6)$.

$$\begin{array}{lll}
\kappa(\tau,\theta)\circ\kappa(\eta,\tau) & = & \\
\kappa(\sum_i F_i\overline{\tau}, \sum_i F_i\overline{\theta})\circ\kappa(\sum_i F_i\overline{\eta}, \sum_i F_i\overline{\tau}) & = & \\
(\sum_i \kappa(F_i\overline{\tau}, F_i\overline{\theta}))\circ(\sum_i \kappa(F_i\overline{\eta}, F_i\overline{\tau})) & = & \\
\sum_i \kappa(F_i\overline{\tau}, F_i\overline{\theta})\circ\kappa(F_i\overline{\eta}, F_i\overline{\tau}) & \to^* & \text{induction} \\
\sum_i \kappa(F_i\overline{\eta}, F_i\overline{\theta}) & = & \\
\kappa(\sum_i F_i\overline{\eta}, \sum_i F_i\overline{\theta}) & &
\end{array}$$

□

B.3 Proofs for uniqueness of normal forms

Lemma B.3.1

$$\begin{tikzcd}
 & M \arrow[dl, "\Delta^{\phi} R"'] \arrow[dr, "\Delta^{\phi} S_i", "*"'] & \\
M_1 \arrow[dr, dashed, "\Delta^{\phi} S'_i", "*"'] & & M_2 \arrow[dl, dashed, "\Delta^{\phi} R'"'] \\
 & M' &
\end{tikzcd}$$

where it holds that

1. *if the reduction $\Delta^{\phi}R'$ contains a non-empty P-step, then so does the reduction $\Delta^{\phi}R$, and moreover $\Delta R \cong \Delta R'$; and if the reduction $\Delta^{\phi}R'$ contains no non-empty P-step, then the ϕ^*-reductions eliminate the coercions involved in the ΔR-redex*
2. *if the reduction $\Delta^{\phi}S'_i$ contains a non-empty P-step, then so does the reduction $\Delta^{\phi}S_i$, and moreover $\Delta S_i \cong \Delta S'_i$; and if the reduction $\Delta^{\phi}S'_i$ contains no non-empty P-step, then the ϕ^*-reductions eliminate the coercions involved in the ΔS_i-redex*

Proof By induction in the length of $M \overset{\Delta^{\phi}S_i}{\rightarrow^*} M_2$, using Lemma 5.3.11. □

Lemma B.3.2

$$\begin{tikzcd}
 & M \arrow[dl, "\Delta^{\phi}R_i"', "*"] \arrow[dr, "\Delta^{\phi}S_i", "*"'] & \\
M_1 \arrow[dr, dashed, "\Delta^{\phi}S'_i"', "*"] & & M_2 \arrow[dl, dashed, "\Delta^{\phi}R'_i", "*"'] \\
 & M' &
\end{tikzcd}$$

where it holds that

1. *if the reduction $\Delta^{\phi}R'_i$ contains a non-empty P-step, then so does the reduction $\Delta^{\phi}R_i$, and moreover $\Delta R_i \cong \Delta R'_i$; and if the reduction $\Delta^{\phi}R'_i$ contains no non-empty P-step, then the ϕ^*-reductions eliminate the coercions involved in the ΔR_i-redex*
2. *if the reduction $\Delta^{\phi}S'_i$ contains a non-empty P-step, then so does the reduction $\Delta^{\phi}S_i$, and moreover $\Delta S_i \cong \Delta S'_i$; and if the reduction $\Delta^{\phi}S'_i$ contains no non-empty P-step, then the ϕ^*-reductions eliminate the coercions involved in the ΔS_i-redex*

Proof By induction in the length of $M \overset{\Delta^{\phi}R_i}{\rightarrow^*} M_1$, using Lemma B.3.1. □

Proposition B.3.3 *Let $\mathcal{C}$ be any one of the primitive completion classes. Suppose $M \Rightarrow^*_{P_\phi} M_1$ and $M \Rightarrow^*_{P_\phi} M_2$ with $M_1, M_2 \in \mathcal{C}$. Then there exists $M' \in \mathcal{C}$ such that $M_2 \Rightarrow^*_{P_\phi} M'$ and $M_2 \Rightarrow^*_{P_\phi} M'$:*

$$\begin{tikzcd}
 & M \arrow[dl, "(P_\phi)^*"'] \arrow[dr, "(P_\phi)^*"] & \\
M_1 \in \mathcal{C} \arrow[dr, dashed, "(P_\phi)^*"'] & & M_2 \in \mathcal{C} \arrow[dl, dashed, "(P_\phi)^*"] \\
 & M' \in \mathcal{C} &
\end{tikzcd}$$

Proof We construct the lower reductions $M_1 \Rightarrow^*_{P_\phi} M'$ and $M_2 \Rightarrow^*_{P_\phi} M'$ according to Lemma B.3.2. Consider any of those lower reductions, say, $M_2 \Rightarrow^*_{P_\phi} M'$; by the Lemma, this reduction can be constructed such that every P-step satisfies:

- *either* it eliminates coercions involved in a corresponding step in the top reduction $M \Rightarrow^*_{P_\phi} M_1$; and in this case, the P-step of the top reduction will lead to a situation where the involved coercions are stuck wrt. further P-reductions, until the same elimination is performed. The latter property follows by inspection of polarities (any coercion in a ϕ-redex is stuck under P-reduction.) It follows from this property that the coercions created in the top reduction cannot be used to eliminate any other coercions than those which are eliminated under the ϕ-reduction of the lower reductions.
- *or else* it generates a coercion of the same structure as is done in the corresponding step in the reduction $M \Rightarrow^*_{P_\phi} M_1$, and at the same point in the term, by displacing coercions coming from the same points in the term.

By the way the lower reductions are constructed we can see that each one of those simulate its corresponding top reduction in a lock-step fashion; in particular, every time a top step constructs/destructs a coercion, the lower reduction will either construct/destruct a corresponding coercion, or it will eliminate it; in the case of elimination, the coercion constructed in the top reduction is stuck (until it gets eliminated at some later point.) From this characterization it can be seen that, if the end point of a top reduction is in a primitive class $\mathcal{C}$, then so must the end point of the simulating reduction be. And so, by construction, $M_2 \in \mathcal{C}$ implies $M' \in \mathcal{C}$. □

B.4 Proofs for normalization and β-reduction

Lemma B.4.1 *For every coercion d and coercion context $\mathbf{C}$, one has*

$$\lceil \mathbf{C}[d] \rceil \equiv \lceil \mathbf{C} \rceil [\lceil d \rceil]$$

Proof Easy structural induction on $\mathbf{C}$. □

The translation on coercions respects coercion equality under $\beta\eta$ equality:

Lemma B.4.2 *If $C \vdash c = c'$ then $\lceil c \rceil =_{\beta\eta} \lceil c' \rceil$*

Proof We first show that, for each axiom $\mathcal{L} = \mathcal{R}$ of system C, we have $\lceil \mathcal{L} \rceil =_{\beta\eta} \lceil \mathcal{R} \rceil$.

Case $[C1]$. We clearly have $\lceil c \rceil \equiv \lceil c' \rceil$ in this case.

Case $[C2], [C3]$. We have

$$\begin{array}{rcl} \lceil id \circ c \rceil & \equiv & \lambda x. \lceil id \rceil (\lceil c \rceil x) \\ & \rightarrow_\beta & \lambda x. \lceil c \rceil x \\ & \rightarrow_\eta & \lceil c \rceil \end{array}$$

The case for $[C3]$ is similar and left out.

Case $[C4]$. We have, on the one hand

$$\begin{array}{rcl} \lceil (c' \to d) \circ (c \to d') \rceil & \equiv & \lambda x.(\lambda f.\lambda y.\lceil d \rceil (f(\lceil c' \rceil y)))\ ((\lambda g.\lambda z.\lceil d' \rceil (g(\lceil c \rceil z)))x) \\ & \rightarrow^*_\beta & \lambda x.\lambda y.\lceil d \rceil ((\lambda z.\lceil d' \rceil (x(\lceil c \rceil z)))\ (\lceil c' \rceil y)) \\ & \rightarrow_\beta & \lambda x.\lambda y.\lceil d \rceil (\lceil d' \rceil (x(\lceil c \rceil (\lceil c' \rceil y)))) \end{array}$$

On the other hand,

$$\begin{array}{rcl} \lceil (c \circ c') \to (d \circ d') \rceil & \equiv & \lambda x.\lambda y.\lceil d \circ d' \rceil (x(\lceil c \circ c' \rceil y)) \\ & \equiv & \lambda x.\lambda y.(\lambda z.\lceil d \rceil (\lceil d' \rceil z))\ (x((\lambda w.\lceil c \rceil (\lceil c' \rceil w))y)) \\ & \rightarrow^*_\beta & \lambda x.\lambda y.\lceil d \rceil (\lceil d' \rceil (x(\lceil c \rceil (\lceil c' \rceil y)))) \end{array}$$

Case [$C5$]. We have

$$\begin{array}{rcl}\lceil id \to id\rceil & \equiv & \lambda f.\lambda x.\lceil id\rceil(f(\lceil id\rceil x)) \\ & \to_\beta^* & \lambda f.\lambda x.fx \\ & \to_\eta & \lambda f.f \\ & \equiv & \lceil id\rceil\end{array}$$

Hence, one has $\lceil\mathcal{L}\rceil =_{\beta\eta} \lceil\mathcal{R}\rceil$ in all cases of C. From this the result follows by extension to arbitrary coercion contexts , *i.e.*, $\mathbf{C}[\mathcal{L}] = \mathbf{C}[\mathcal{R}]$ implies $\lceil\mathbf{C}[\mathcal{L}]\rceil =_{\beta\eta} \lceil\mathbf{C}[\mathcal{R}]\rceil$. This, in turn, follows from Lemma B.4.1. □

The translation on coercions respects ϕ-reduction on coercions, defining the notion of ϕ-reduction on $\lambda^{\to}(\mathbf{D})$-completions by

Definition B.4.3 (ϕ reduction over $\lambda^{\to}(\mathbf{D})$)

$$\mathtt{F?}(\mathtt{F!}\,M) \to_\phi (\lambda x.x)M$$

□

Lemma B.4.4 *For any coercion context* $\mathbf{C}$*, one has* $\lceil\mathbf{C}[\mathtt{F?}\circ\mathtt{F!}]\rceil \to_{\eta\circ\phi} \lceil\mathbf{C}[id]\rceil$.

Proof For the empty context we have $\lceil\mathtt{F?}\circ\mathtt{F!}\rceil \equiv \lambda x.\mathtt{F?}(\mathtt{F!}\,x) \to_\phi \lambda x.(\lambda y.y)x \to_\eta \lambda y.y \equiv \lceil id\rceil$. This is generalized to arbitrary contexts by Lemma B.4.1. □

It immediately follows that the translation on coercions respects ϕ conversion under $\beta\eta\phi$ conversion:

Corollary B.4.5 *If* $c =_\phi c'$ *then* $\lceil c\rceil =_{\beta\eta\phi} \lceil c'\rceil$.

Proof Follows directly from Lemma B.4.2 together with Lemma B.4.4. □

We now consider the translation on completions. First we need a substitution lemma:

Lemma B.4.6 *(Substitution)*

$$[\![M\{x := N\}]\!] \equiv [\![M]\!]\{x := [\![N]\!]\}$$

Proof By structural induction on M. We consider just the case
Case $M \equiv [c]M'$. Here we have

$$\begin{array}{lll}[\![M\{x := N\}]\!] & \equiv & \\ [\![([c]M')\{x := N\}]\!] & \equiv & \\ \lceil c\rceil([\![M'\{x := N\}]\!]) & \equiv & \text{induction} \\ \lceil c\rceil([\![M']\!]\{x := [\![N]\!]\}) & \equiv & \\ (\lceil c\rceil[\![M']\!])\{x := [\![N]\!]\} & \equiv & \\ [\![[c]M']\!]\{x := [\![N]\!]\} & \equiv & \\ [\![M]\!]\{x := [\![N]\!]\} & & \end{array}$$

The other cases follow by easy application of the induction hypothesis. □

We see that completion equality is respected by the translation on completions under $\beta\eta$ equality:

Proposition B.4.7 *If $E \vdash M = M'$ then $[\![M]\!] =_{\beta\eta} [\![M']\!]$.*

Proof We first prove $[\![\mathcal{L}]\!] =_{\beta\eta} [\![\mathcal{R}]\!]$ for every axiom $\mathcal{L} = \mathcal{R}$ of system E. The cases $[E1-3]$ follow from Lemma B.4.2. For the remaining cases we have

Case $[E4]$.

$$\begin{array}{rcll}
[\![[c \to d](\lambda x.M)]\!] & \equiv & (\lambda f.\lambda y.\lceil d\rceil(f(\lceil c\rceil y)))\,(\lambda x.[\![M]\!]) & \\
& \to\beta & \lambda y.\lceil d\rceil((\lambda x.[\![M]\!])\,(\lceil c\rceil y)) & \\
& \to_\beta & \lambda y.\lceil d\rceil([\![M]\!]\{x := \lceil c\rceil y\}) & \\
& \equiv & \lambda y.\lceil d\rceil([\![M]\!]\{x := [\![[c]y]\!]\}) & \\
& \equiv & \lambda y.\lceil d\rceil([\![M\{x := [c]y\}]\!]) & \text{by Substitution Lemma} \\
& \equiv & \lambda y.[\![[d](M\{x := [c]y\})]\!] & \\
& \equiv & [\![\lambda y.[d](M\{x := [c]y\})]\!] &
\end{array}$$

Case $[E5]$.

$$\begin{array}{rcl}
[\![([c \to d]M)]\!]\,N & \equiv & ((\lambda f.\lambda x.\lceil d\rceil(f(\lceil c\rceil x)))\,[\![M]\!])\,[\![N]\!] \\
& \to^*_\beta & \lceil d\rceil([\![M]\!](\lceil c\rceil[\![N]\!])) \\
& \equiv & [\![[d](M\,([c]N))]\!]
\end{array}$$

The claim now follows by considering the E axioms used in any completion context. This is done by structural induction on completion contexts. It is straight-forward and left out. □

We now prove some lemmas needed for Proposition B.4.10 below.

Lemma B.4.8 *If $M \to_\phi M'$ just by contraction of a ϕ-redex (using no equations), then $[\![M]\!] \to_{\eta^*\circ\phi} [\![M']\!]$.*

Proof The lemma follows immediately from Lemma B.4.4; the closure η^* is due to the case where we have

$$M \equiv [\mathtt{F?}]([\mathtt{F!}]M'') \to_\phi [id]M'' \equiv M'$$

where the translated reduction is just a ϕ-step:

$$[\![M]\!] \equiv \mathtt{F?}(\mathtt{F!}\,[\![M'']\!]) \to_\phi (\lambda x.x)[\![M'']\!] \equiv [\![[id]M'']\!]$$

□

Lemma B.4.9 *Let M be a $\lambda^{\to}(\mathbf{D})$-term such that $M \to_\phi M_1$ and $M \to_{\beta\eta} M_2$. Then*

1. *There exists an M_3 such that $M_2 \to^*_\phi M_3$ and $M_1 \to_{\beta\eta} M_3$, and in particular,*
2. *If M is a λI-term then one has in addition $M_2 \to^+_\phi M_3$*
3. *ϕ^* commutes with $(\beta\eta)^*$ on $\lambda^{\to}(\mathbf{D})$-terms.*
4. *$\phi\beta\eta$ is confluent.*

Proof We consider the first two claims first. Observe that a $\beta\eta$-reduction from M can only do one of two things to a ϕ-redex in M:

- it can duplicate the ϕ-redex, or

- it can throw it away; this holds only if M is a non-λI-term, as for instance one has $(\lambda x.y)\ (\texttt{F?}(\texttt{F!}\,z)) \rightarrow_\beta y$.

Hence, in the λI-case, we can choose for $M_2 \rightarrow^+_\phi M_3$ the contraction of just those ϕ-redices in M_2 which are duplicates of the ϕ-redex contracted in $M \rightarrow_\phi M_1$; in the non-λI-case, where the ϕ-redex is thown away, we choose just the empty ϕ-reduction.

Now, ϕ-reduction cannot block a $\beta\eta$-redex: If $M \rightarrow_\phi M'$ then M' contains all the $\beta\eta$-redices which M contains (this is obvious.) Therefore, the $\beta\eta$-redex contracted in M to yield M_2 must also be present in M_1, so that, by construction of $M_2 \rightarrow^*_\phi M_3$, it must be the case that $M_1 \rightarrow_{\beta\eta} M_3$.

This establishes the two first claims of the Lemma. The third claim follows from the first by diagram chase. The fourth claim follows, by the Hindley-Rosen Lemma, from the third claim together with confluence of ϕ on coercions and confluence of $\beta\eta$. □

We can now establish a simulation property for the translation $[\![\bullet]\!]$:

Proposition B.4.10 *(Correspondence)*
Let M be a $\mathbf{D}$-completion with

$$M =_E M' \rightarrow_\phi N$$

Then

1. *There exist $\lambda^{\rightarrow}(\mathbf{D})$-terms Q, S such that $[\![M]\!] \rightarrow^*_{\beta\eta} S$, $[\![M']\!] \rightarrow^*_{\beta\eta} S$, $S \rightarrow^*_{\beta\eta\phi} Q$, and $[\![N]\!] \rightarrow^*_{\beta\eta} Q$, and in particular,*
2. *If M is a λI-completion, then one has in addition that $S \rightarrow^+_{\beta\eta\phi} Q$.*

$$\begin{tikzcd}
M \arrow[rr, equal] \arrow[d, no head] & & M' \arrow[r, "\phi"'] \arrow[d, no head] & N \arrow[d, no head] \\
{[\![M]\!]} \arrow[rr, equal, "\beta\eta"] \arrow[dr, dash, no head, "(\beta\eta)^*"] & & {[\![M']\!]} \arrow[r, "\eta\phi"'] \arrow[dl, dash, no head, "(\beta\eta)^*"] & {[\![N]\!]} \arrow[d, dash, no head, "(\beta\eta)^*"] \\
& S \arrow[rr, dash, no head, "(\beta\eta\phi)^{*/+}"'] & & Q
\end{tikzcd}$$

Proof By Proposition B.4.7, one has $[\![M]\!] =_{\beta\eta} [\![M']\!]$. Moreover, by Lemma B.4.8, there exists a term P such that

$$[\![M']\!] \rightarrow^*_\eta P \rightarrow_\phi [\![N]\!]$$

Now, by classical results, we now that $\beta\eta$ is confluent. By Proposition 4.6.3, $[\![M]\!]$ and $[\![M']\!]$ are both $\lambda^{\rightarrow}(\mathbf{D})$-terms. Since ϕ-reduction on $\lambda^{\rightarrow}(\mathbf{D})$-terms evidently satisfies subject reduction, P is also a $\lambda^{\rightarrow}(\mathbf{D})$-term. Therefore, $[\![M]\!]$ and P have a common $\beta\eta$-reduct, call it S. By commutativity (Lemma B.4.9), there exists a term Q such that $S \rightarrow^*_\phi Q$ and $[\![N]\!] \rightarrow^*_{\beta\eta} Q$. Suppose now that M is a λI-completion. Then, as is easily verified, $[\![M]\!], [\![M']\!], P, S, [\![N]\!]$ must all be λI-terms. It then follows by Lemma B.4.9 that, in the λI-case, we have $S \rightarrow^+_\phi Q$. In both cases, the existence of the desired terms Q, S is established. □

Bibliography

[ACPP89] Abadi, M. & Cardelli, L. & Pierce, B. & Plotkin, G. Dynamic Typing in a Statically-Typed Language. In *Proceedings POPL '89, Symposium on Principles of Programming Languages.* 1989

[Aik94] Aiken, A. Set Constraints: Results, Applications and Future Directions. In *Proceedings Workshop on Principles and Practice of Constraint Programming, PPCP. LNCS 874.* 1994

[AKVW93] Aiken, A. & Kozen, D. & Vardi, M. & Wimmers, E. The Complexity of Set Constraints. In *Proceedings CSL, Conference on Computer Science Logic, LNCS* 1993

[And94] Andersen, J. K. Semantics of Dynamically Typed Lambda Calculus. In *DIKU Student Project.* 1994

[App89] Appel, A. Runtime Tags Aren't Necessary. In *Lisp and Symbolic Computation, vol. 2, no. 2, pp. 153 - 162.* 1989

[ASU86] Aho, A. V. & Sethi, R. & Ullman, J. D. *Compilers. Principles, Techniques, and Tools* Addison-Wesley 1986

[AW92] Aiken, A. & Wimmers, E. L. Solving Systems of Set Constraints. In *Proceedings LICS, Symposium on Logic in Computer Science* 1992

[AW93] Aiken, A. & Wimmers, E. L. Type Inclusion Constraints and Type Inference. In *Proceedings FPCA, Conference on Functional Programming and Computer Architecture, 31-41.* 1993

[AWL94] Aiken, A. & Wimmers, E. L. & Lakshman, T. K. Soft Typing with Conditional Types. In *Proceedings POPL, 163-173.* 1994

[Bar84] Barendregt, H. P. *The Lambda Calculus. Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics, 103, Amsterdam 1984

[Bar92] Barendregt, H. P. Lambda Calculi with Types. In *[HLCS92], vol. 2, 117-311*

[BGS90] Breazu-Tannen, V. & Gunter, C. & Scedrov, A. Computing with Coercions. In *Proceedings Lisp and Functional Programming, 44-60.* 1990

[BCGS89] Breazu-Tannen, V. & Coquand, T. & Gunter, C. & Scedrov, A. Inheritance and Implicit Coercion (Preliminary Report). In *Proceedings LICS '89, Symposium on Logic in Computer Science, 112-134.* 1989

[BCGS91] Breazu-Tannen, V. & Coquand, T. & Gunter, C. & Scedrov, A. Inheritance as Implicit Coercion. In *Information and Computation, 93, 172-221.* 1991

[BD86] Bachmair, L. & Dershowitz, N. Commutation, Transformation, and Termination. In *8'th Intnl. Conf. on Automated Deduction.* LNCS 230, 1986

[BGW93] BACHMAIR, L. & GANZINGER, H. & WALDMANN, U. Set Constraints are the Monadic Class. In *Proceedings LICS, Symposium on Logic in Computer Science.* 1993

[BJ93a] BONDORF, A. & JØRGENSEN, J. Efficient Analyses for Realistic Off-line Partial Evaluation. In *Journal of Functional Programming, special issue on partial evaluation.* 1993

[BJ93b] BONDORF, A. & JØRGENSEN, J. Efficient Analyses for Realistic Off-line Partial Evaluation. In *DIKU Report 93/4* 1993

[BW90] BARR, M. & WELLS, C. *Category Theory for Computing Science.* Prentice Hall, 1990.

[Cam94] CAMERON, P. J. *Combinatorics. Topics, Techniques, Algorithms.* Cambridge University Press, 1994

[CC91] CARDONE, F. & COPPO, M. Type Inference with Recursive Types: Syntax and Semantics. In *Information and Computation 92.* 1991

[CG90] CURIEN, P-L. & GHELLI, G. Coherence of Subsumption. In *Coll. on Trees in Algebra and Programming, 132-146* 1990

[CG94] CURIEN, P-L. & GHELLI, G. Decidability and Confluence of $\beta\eta top$ Reduction in $F_{\leq}$. In *Information and Computation, 109, 57-114.* 1994

[CF91] CARTWRIGHT, R. & FAGAN, M. Soft Typing. In *Proceedings SIGPLAN '91 Conference on Programming Language Design and Implementation, 278-292.* 1991

[CF92] CARTWRIGHT, R. & FAGAN, M. Soft Typing. In *Unpublished Manuscript.* 1992

[Cour83] COURCELLE, B. Fundamental Properties of Infinite Trees. In *Theoretical Computer Science 25.* 1983

[CR91] CLINGER, W. & REES, J. (EDS) *Revised^4 Report on the Algorithmic Language Scheme.*

[CW85] CARDELLI, L. & WEGNER, P. On Understanding Types, Data Abstraction and Polymorphism. In *ACM Computing Surveys, 17.* December 1985

[DaPr90] DAVEY, B.A. & PRIESTLEY, H. A. *Introduction to Lattices and Order.* Cambridge. 1990

[Der87] DERSHOWITZ, N. Termination of Rewriting. In *Journal of Symbolic Computation 1-2.* 1987

[DeJo90] DERSHOWITZ, N. & JOUANNAUD, J-P. Rewrite Systems. In *[HTCS90] chapter 6*

[DeMa79] DERSCHOWITZ, N. & MANNA, Z. Proving Termination with Multiset Orderings. In *Communications of the ACM, 22 (8), 465-476.* 1979

[Dam85] DAMAS, L. M. M. *Type Assignment in Programming Languages.* PhD Thesis, University of Edinburgh, 1985

[deGr95] de Groote, P. A Simple Calculus of Exception Handling. In *To appear: Proceedings Typed Lambda Calculi and Applications (TLCA)* 1995

[DHM91] DUBA, B. F. & HARPER, R. & MACQUEEN, D. Typing First Class Continuations in ML. In *Proceedings POPL '91.* 1991

[DHM93] DUBA, B. F. & HARPER, R. & MACQUEEN, D. Typing First Class Continuations in ML. In *Journal of Functional Programming, 3 (4), 465-484.* Cambridge 1993. Revised version of [DHM91].

[DM82] DAMAS, L. & MILNER, R. Principal Type-Schemes for Functional Programs. In *POPL '82, 207-212* 1982

[DRW95] Dubois, C. & Rouaix, F. & Weis, P. Extensional Polymorphism. In *Proceedings POPL '95, Symposium on Principles of Programming Languages.* 1995

[Dyb87] Dybvig, R. K. *The Scheme programming Language.* Prentice Hall 1987

[Fag90] Fagan, M. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages.* Ph.D. Thesis, Rice University. 1990

[Fel87] Felleisen, M. *The Calculi of $\lambda_v - CS$ Conversion: A Syntactical Theory of Control and State in Imperative Higher Order Programming Languages.* Ph.D. thesis, Indiana University, 1987

[Fel91] Felleisen, M. On the Expressive Power of Programming Languages. In *Science of Computer Programming 17, 35-75.* 1991. Preliminary version in: Proceedings ESOP, European Symposium on Programming, LNCS 432, 134- 151, 1990.

[FFKD87] Felleisen, M. & Friedman, D. & Kohlbecker, E & Duba, B. A Syntactic Theory of Sequential Control. In *Theoretical Computer Science, 52 (3), 205-237.* 1987

[FH89] Felleisen, M & Hieb, R. The Revised Report on the Syntactic Theory of Sequential Control and State. In *Rice University Technical Report, Rice COMP TR89-100.* 1989

[FM90] Fuh, Y-C. & Mishra, P. Type Inference with Subtypes. In *Theoretical Computer Science, 73, 155-175.* 1990

[Ghe90] Ghelli, G. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, TD-6/90, University of Pisa, 1990.

[GJS93] Jones, N. D. & Gomard, C. & Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice Hall 1993

[GLT89] Girard, J.Y. & Lafont, Y. & Taylor, P. *Proofs and Types.* Cambridge Tracts in Theoretical Conmputer Science 7, Cambridge University Press, 1989

[Gom90] Gomard, C. Partial Type Inference for Untyped Functional Programs (Extended Abstract). In *Proceedings LISP and Functional Programming (LFP '90)* 1990

[Gun92] Gunter, C. A. *Semantics of Programming Languages. Structures and Techniques.* Foundations of Computing Series, MIT. 1992

[Hei94] Heintze, N. Set-based Analysis of ML Programs (Extended Abstract). In *Proceedings LFP, Conference on Lisp and Functional Programming.* 1994

[HeJa94] Heintze, N. & Jaffar, J. Set Constraints and Set-Based Analysis. In *Proceedings Workshop on Principles and Practice of Constraint Programming, PPCP. LNCS 874.* 1994

[Hen89] Henglein, F. *Polymorphic Type Inference and Semi-Unification* PhD Thesis, Courant Institute of Mathematical Sciences, New York University, 1989

[Hen91] Henglein, F. Efficient Type Inference for Higher-Order Binding Time Analysis. In *Proceedings Conference on Functional Programming and Computer Architecture (FPCA) '91.* 1991

[Hen92] Henglein, F. Dynamic Typing. In *ESOP '92, LNCS 582, 233-253.* 1992

[Hen92a] Henglein, F. Global Tagging Optimization by Type Inference. In *Proceedings LISP and Functional Programming, LFP '92, 233-253.* 1992

[Hen94] HENGLEIN, F. Dynamic Typing. Syntax and Proof Theory. In *Science of Computer Programming 22, 197-230.* 1994

[Hen94a] HENGLEIN, F. *Personal communication.*

[Hin69] HINDLEY, R. The Principal Type-Scheme of an Object in Combintory Logic In *Transactions of the Amarican Mathematical Society, 146, 29-60* December 1969

[HJ94] HENGLEIN, F. & JØRGENSEN, J. Formally Optimal Boxing. In *Proceedings POPL'94.* 1994

[HL92] HARPER, R. & LILLIBRIDGE, M. Polymorphic Type Assignment and CPS Conversion. In *Proceedings ACM SIGPLAN Workshop on Continuations, San Francisco.* June 1992

[HL93] HARPER, R. & LILLIBRIDGE, M. Polymorphic Type Assignment and CPS Conversion. In *Lisp and Symbolic Computation, 6, 361-380.* 1993. Revised version of [HL92].

[HLCS92] ABRAMSKY, S. & GABBAY,D.M. & MAIBAUM,T.S.E (EDS) *Handbook of Logic in Computer Science.* Oxford 1992

[HM95] HARPER, R. & MORRISETT, G. Compiling Polymorphism Using Intensional Type Analysis. In *Proceedings POPL '95, Symposioum on Principles of Programming Languages.* 1995

[HMV93] HOANG, M. & MITCHELL, J. & VISWANATHAN, J. Standard ML-NJ Weak Polymorphism and Imperative Constructs. In *Proceedings LICS '93, Symposium on Logic in Computer Science.* 1993

[HO80] HUET, G. & OPPEN, , D. C. Equations and Rewrite Rules: A Survey. In *Book, R. ed., Formal Language Theory: Perspectives and Open Problems, New York 1980, 349-405.* 1980

[HS90] HINDLEY, R. & SELDIN, J. P. *Introduction to Combinators and λ-Calculus.* London Mathematical Society Student Texts, 1, 1990 (reprint of first ed. 1986.)

[HTCS90] VAN LEEUWEN, J. *Handbook of Theoretical Computer Science.* Elsevier 1990

[Hue81] HUET, G. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. In *Journ. ACM 27 (4), 797-821.* 1981

[Jon92] JONES, M.P. A Theory of Qualified Types. In *Proceedings ESOP '92, LNCS 582.* 1992

[Jon94] JONES, M.P. ML Typing, Explicit Polymorphism, and Qualified Types. In *Proceedings TACS '94, LNCS 789, 56-75.* 1994

[KB67] KNUTH, D.E. & BENDIX, P. B. Simple Word Problems in Universal Algebra. In *Proceedings of the Conference on Computational Problems in Abstract Algebra, ed.: Leech, J., 263-298. Oxford, Pergamon Press.* 1967

[Kir94] KIRCHNER, C. & KIRCHNER, H. *Rewriting, Solving, Proving.* Worknotes for forthcoming book, 1994

[Klo87] KLOP, J.-W. Term Rewriting Systems: A Tutorial. In *Bull. European Assoc. Theoretical Computer Science, 32, 143-183.* 1987

[KW94] KFOURY, A. J. & WELLS, J. B. A Direct Algorithm for Type Inference in the Rank-2 Fragment of the Second-Order λ-Calculus. In *Proceedings Lisp and Functional Programming, 196-207.* 1994

[LaSc86] LAMBEK, J. & SCOTT, P.J. *Introduction to Higher Order Categorical Logic.* Cambridge studies in advanced mathematics, 7, Cambridge University Press, 1986.

[Ler92] LEROY, X. Unboxed Objects and Polymorphic Typing. In *Proceedings POPL '92, Symposium on Principles of Programming Languages.* 1992

[LM91] LEROY, X. & MAUNY, M. Dynamics in ML. In *Proceedings FPCA '91, Conference on Functional Programming and Computer Architecture, LNCS 523, 406-426.* 1991

[Mai92] MAIRSON, H. Quantifier Elimination and Parametric Polymorphism in Programming Languages. In *Journal of Functional Programming, 2 (2), 213-226.* Cambridge, 1992

[Mil78] MILNER, R. A Theory of Type Polymorphism in Programming In *Journal of Computer and System Sciences, 17, 348-375* 1978

[MLNJ93] *Standard ML of New Jearsey. Version 0.93.* AT & T Bell Laboratories. 1993

[MPS86] MACQUEEN, D. & PLOTKIN, G. & SETHI, R. An Ideal Model for Recursive Polymorphic Types. In *Information and Control, 71.* 1986

[MTH90] MILNER, R. & TOFTE, M. & HARPER, R. *The Definition of Standard ML* MIT Press, 1990

[MTH91] MILNER, R. & TOFTE, M. & HARPER, R. *Commentary on Standard ML* MIT Press, 1991

[OW92] O'KEEFE, P. & WAND, M. Type Inference for Partial Types is Decidable. In *Proceedings ESOP '92, European Symposium on Programming, LNCS 582, 408-417.* 1992

[KPS92] KOZEN, D. & PALSBERG, J. & SCHWARTZBACH, M. Efficient Inference of Partial Types. In *Technical Report DAIMI PB-394, University of Aarhus.* 1992

[PaOK95] PALSBERG, J. & O'KEEFE, P. A Type System Equivalent to Flow Analysis. In *Proceedings POPL'95, forthcoming.* 1995

[Pau91] PAULSON, L. C. *ML For the Working Programmer.* Canbrifge University Press, 1991.

[Pie91] PIERCE, B. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, School of Computer Science, Carnegie mellon University. 1991

[Plo75] PLOTKIN, G. D. Call-by-Name, Call-by-Value and the λ Calculus. In *Theoretical Computer Science 1.* 1975

[Pra65] PRAWITZ, D. *Natural Deduction.* Almquist & Wiksell, Uppsla 1965.

[PS95] PALSBERG, J. & SCHWARTZBACH, J. Safety Analysis Versus Type Inference. In *Information and Computation, to appear*

[Rem89] RÉMY, D. Typechecking Records and Variants in a Natural Extension of ML. In *Proceedings POPL '89, 77-87.* 1989

[Rey83] REYNOLDS, J.C. Types, Abstraction, and Parametric Polymorphism. In *Information Processing 83, 513-523, ed. Mason, R. E. A., North-Holland, Amsterdam.* 1983

[RS94] REHOF, J. & SØRENSEN, M.H. The λ_Δ Calculus. In *Proceedings TACS '94, LNCS 789, 516-543.* 1994

[Sed88] SEDGEWICK, R. *Algorithms.* Second Edition, Addison Wesley, 1988.

[Shi91] SHIVERS, O. Data-Flow Analysis and Type recovery in Scheme. In *Peter Lee, ed, Topics in Advanced Language Implementation, 47-87.* MIT 1991

[Tar83] TARJAN, R. *Data Structures and Network Flow Algorithms.* Vol. CMBS 44, Regional Conference Series in Applied Mathematics, SIAM 1983.

[Tha88] THATTE, S. Type Inference with partial Types. In *Automata, Languages and Programming: 15'th International Colloquium, LNCS 317, 615-629.* 1988

[Tha90] THATTE, S. Quasi-Static Typing. In *Proceedings POPL, 367-381.* 1990

[Tha94] THATTE, S. Type Inference with partial Types. In *Theoretical Computer Science, 124, 127-148.* 1994

[Tof90] TOFTE, M. Type Inference for Polymorphic References. In *Information and Computation 89, 1, 1-34.* November 1990

[Tol94] TOLMACH, A. Tag-free Garbage Collection Using Explicit Type Parameters. In *Proceedings LFP '94, Conference on Lisp and Functional Programming.* 1994

[TT93] TOFTE, M. & TALPIN, J. P. *A Theory of Stack Allocation in Polymorphically Typed Languages.* DIKU Report 93/15, 1993.

[TT94] TOFTE, M. & TALPIN, J. P. Implementation of the Typed Call-by-Value λ-calculus using a Stack of Regions. In *Proceedings POPL '94, Conference on the Principles of Programming Languages.* 1994

[Wad89] WADLER, P. Theorems for Free ! In *Proceedings FPCA '89, Conference on Functional Programming and Computer Architecture, 347-359.* 1989

[Wan86] WAND, M. Finding the Source of Type Errors. In *Proceedings POPL '86, 38-43.* 1986

[Wan87] WAND, M. A Simple Algorithm and Proof for Type Inference. In *Fundamentae Informaticae, X, 115-122.* North-Holland, 1987

[WF91] WRIGHT, A. K. & FELLEISEN, M. A Syntactic Approach to Type Soundness. In *Rice University Technical Report, 91-160.* 1991. Also in Information and Computation, 1994.

[WOP94] WAND, M. & O'KEEFE, P. & PALSBERG, J. Strong Normalization with Non-Structural Subtyping. In *Unpublsihed manuscript, to appear in Mathematical Structures in Computer Science.* 1994

[Wyn94] WYNSKEL, G. *The Formal Semantics of Programming Languages. An Introduction.* MIT Press, 1994

[Wri94] WRIGHT, A. K. *Practical Soft Typing.* Ph.D. Thesis, Rice University. 1994

[Wri94a] WRIGHT, A. K. Simple Imperative Polymorphism. In *Lisp and Symbolic Computation. Special issue on State in Programming Languages.* 1994

[Wri93] WRIGHT, A. K. Polymorphism for Imperative languages without Imperative Types. In *Rice University technical report TR93-200.* 1993

[WrCa94] WRIGHT, A. K. & CARTWRIGHT, R. A Practical Soft Type System for Scheme. In *Proceedings LISP and Functional Programming.* 1994