Polymorphic Type Inference for Object-Oriented Programming Languages

Dissertation

zur Erlangung des Grades
des Doktors der Naturwissenschaften
der Technischen Fakultät
der Universität des Saarlandes
von

Andreas V. Hense

Saarbrücken 1994

Tag des Kolloquiums: 24.5.1994

Dekan:	Prof. Dr. G. Hotz
Berichterstatter:	Prof. Dr. G. Smolka
	Prof. Dr. R. Wilhelm

Polymorphic Type Inference for Object-Oriented Programming Languages

Andreas V. Hense

Second unchanged edition of ISBN 3-930714-00-0, Bonn 2006

Abstract

We present a type inference algorithm and its verification for an object-oriented programming language called O'SMALL. O'SMALL is a class-based language with imperative features. Classes are not first-class citizens. No type declarations are required.

Type inference operates on an extended λ-calculus into which O'SMALL is translated. The system features extensible record types, μ-types, and imperative types.

This work belongs to both theoretical and practical computer science. In the theoretical part, the type inference algorithm for our λ-calculus with records is formalized in order-sorted logic. In the practical part, the algorithm for let-polymorphism and imperative features is based on well-known approaches. These approaches are presented in a new fashion but they are not proven correct.

Keyword Codes: D.1.5, F.3.1, F.3.2[1]

Keywords: Object-oriented programming, Specifying and verifying and reasoning about programs, Semantics of programming languages

[1]From "The Computing Reviews Classification System" (1991 version).

Acknowledgements

First, I would like to thank my two supervisors Gert Smolka and Reinhard Wilhelm.

Reinhard Wilhelm gave me the opportunity to realize my plans in his excellent research group. He kindly supported and advised me during my Saarbrücken years.

Gert Smolka greatly influenced the formal part of this work. He introduced me to order-sorted logic and helped me use it for the correctness proof of my algorithm.

Many thanks to Reinhold Heckmann, who has found answers to all of my questions. He carefully read draft versions of this work. I am indebted to my father Gerhard, my wife Isabelle, and Marie-Hélène Mathieu for patiently proofreading draft versions.

I would like to thank Fritz Müller who has contributed a lot by his broad overview of the field, Frank Pfenning who made valuable suggestions concerning the imperative type inference rules, and Harald Ganzinger who helped me with the termination proofs. I am grateful for "typeful" discussions with Christian Fecht, Martin Müller, Helmut Seidl, and Kurt Sieber.

My special gratitude goes to my wife Isabelle for encouraging and supporting me throughout my graduate research.

This work is dedicated to my parents, Elisabeth and Gerhard.

Contents

1. Einleitung

Große Software-Systeme so zu entwerfen und zu realisieren, daß sie überschaubar, wartbar und erweiterbar werden, ist eine häufig schmerzlich vermißte Fähigkeit in einer Welt, in der der Bedarf an Lösungen durch Computer ständig wächst. Es ist die Aufgabe der Informatik, Methoden und Werkzeuge bereitzustellen, die dazu befähigen, solche Systeme zu erstellen. Programmierung wird häufig als Kunst bezeichnet, und es ist sicher, daß Kreativität und Eingebung zu ästhetischen Programmen führen können. Da hierbei jedoch Funktionalität eine große Rolle spielt, ist Programmierung wohl doch eher eine Ingenieurskunst, weshalb man auch von Software Engineering spricht. Im Software Engineering wird die Erstellung eines Systems in Analyse, Entwurf und Implementierung eingeteilt. Einen zentralen Platz nehmen die Programmiersprachen ein, die nicht nur die Implementierungsphase vollständig bestimmen, sondern auch auf die beiden ersten Phasen einen starken Einfluß haben. Beispielsweise entstanden die objektorientierte Analyse und der objektorientierte Entwurf [5, 19] zeitlich nach den objektorientierten Sprachen. Es ist ohnehin unabdingbar, die in der Analyse- und Entwurfsphase getroffenen Entscheidungen auch in der Programmiersprache ausdrücken zu können.

Der objektorientierte Ansatz verspricht eine wesentliche Milderung der erwähnten Probleme. Nun haben objektorientierte Sprachen wie SMALLTALK [31] zwar den Ruf, überschaubare, wartbare und erweiterbare Programme zu ermöglichen, aber nicht, zu besonders fehlerfreien Programmen zu führen. Man kann darin zwar gut Prototypen erstellen, aber zu einem fertigen Produkt reicht es nicht. Diese Vorbehalte mögen einerseits auf die geringere Effizienz der Implementierungen dieser Sprachen zurückzuführen sein. Andererseits spielt aber auch eine Rolle, daß die Programme nur dynamisch getypt sind, d.h. daß Typfehler erst während der Laufzeit erkannt werden.

Bevor wir mit der Diskussion fortfahren, möchten wir bereits verwendete Begriffe klären. In dieser Arbeit wird der Begriff der objektorientierten Sprache folgendermaßen definiert [103]: die Konzepte *Objekt*, *Klasse* und *Klassenvererbung* müssen unterstützt werden. Ein Objekt besitzt eine Menge von Operationen, üblicherweise Methoden genannt, und einen internen Zustand. Der interne Zustand ist nicht direkt zugänglich, sondern kann nur indirekt über die Methoden verändert werden. Auf diese Weise wird Datenabstraktion erreicht. Objekte kommunizieren miteinander über

Nachrichtenaustausch. Das Ergebnis des Sendens einer Nachricht an ein Objekt (den *Empfänger*) wird nicht nur durch die aktuellen Parameter, sondern auch durch den internen Zustand des Empfängers bestimmt. Wir können Objekte als Verbunde von Methoden modellieren, die beim Senden von Nachrichten selektiert werden. Klassen dienen als Schablonen zur Erzeugung von Objekten. Sie spezifizieren Methoden und können auch deren Implementierung enthalten. Klassenvererbung ist ein Mechanismus, der zur Komposition von Spezifikationen und Implementierungen benutzt wird. Beim Redefinieren von Methoden in Unterklassen wird späte Bindung eingesetzt. Manchmal wird auch von dynamischer Bindung gesprochen, obwohl diese Bezeichnung verwirrend sein kann.

Ein Typfehler ist das Aufeinandertreffen von Werten, die nicht zueinander passen, wie z.B. die Addition einer ganzen Zahl mit einem Wahrheitswert. In objektorientierten Sprachen treten Typfehler meist dann auf, wenn ein Empfänger eine Nachricht "nicht versteht".

Setzen wir nun die Diskussion fort. Werden Typfehler erst während der Laufzeit erkannt, so gilt eine Programmiersprache zu Recht als unsicher, da in diesem Fall manche Fehler häufig erst dann auftreten, wenn das Datenbankanfragesystem schon beim Kunden ist oder die Sonde schon hinter dem Mars. Ein erschöpfendes Testen wie auch ein vollständiges Verifizieren sind in der Praxis meist unmöglich. Man sollte daher solche Typfehler bereits zur Übersetzungszeit erkennen. Die meisten statischen Typsysteme garantieren, daß ein Programm, welches sie akzeptieren, zur Laufzeit keine Typfehler produziert. Wie alle interessanten Probleme in der Informatik ist auch die Frage der Typfehlerfreiheit für alle interessanten Programmiersprachen unentscheidbar. Dies bedeutet, daß alle statischen Typsysteme nur hinreichende Bedingungen für Typfehlerfreiheit beschreiben und damit nur einen Teil der typfehlerfreien Programme akzeptieren. Die Sicherheit muß also durch eine geringere Flexibilität bezahlt werden. Natürlich ist es Aufgabe der Forschung im Bereich der Typsysteme, den zu zahlenden Preis möglichst gering zu halten.

Bei statischen Typsystemen gebrauchen wir oft die Begriffe 'Typüberprüfung' und 'Typinferenz' synonym, obwohl sie sich durch eine Nuance unterscheiden. Die Typüberprüfung liefert zu einem Programm mit Typdeklarationen den Typ desselben bzw. eine Meldung, daß das Programm fehlerhaft getypt ist. Die Typinferenz stellt darüberhinaus fehlende Typ-

deklarationen aus dem Kontext her. Die Typüberprüfung ist daher immer auch Teil der Typinferenz. Wir unterscheiden zwischen monomorphen und polymorphen Typsystemen. Klassische imperative Sprachen wie C [54] oder Pascal [51] haben ein monomorphes Typsystem, welches eher unflexibel ist. C und damit auch die objektorientierte Sprache C++ [90] gewinnen die nötige Flexibilität, indem sie die Typsicherheit aufgeben. In einem monomorphen Typsystem gibt es nur Basistypen, wie z.B. ganze Zahlen oder Wahrheitswerte, und daraus konstruierte Typen, wie z.B. Listen von ganzen Zahlen oder Paare von Wahrheitswerten u.s.w. Wenn man beispielsweise in jenen Sprachen eine Prozedur zum Sortieren von Listen ganzer Zahlen vorfindet, so muß man den Programmtext der Prozedur kopieren, um mit geringfügigen Änderungen eine Prozedur zum Sortieren von Listen reeller Zahlen zu erhalten. In polymorph getypten Sprachen kann man ein und dieselbe Sortierprozedur auf alle Datentypen, die eine totale Ordnung besitzen, anwenden.

Der Autor hat in verschiedenen Projekten [17, 18, 37, 41, 109] die Vorteile objektorientierter Sprachen [31] einerseits und die Vorteile polymorpher Typsysteme [35, 97] andererseits kennengelernt. Die Idee, beides zu kombinieren, lag also nahe. Diese Kombination ist jedoch viel schwieriger als zunächst vermutet. Verbunde müssen auf eine flexible Weise behandelt werden, da sie die Basis für die Modellierung von Objekten bilden. Die Klassenvererbung und die damit zusammenhängende späte Bindung müssen berücksichtigt werden. Das schwierigste Problem stellen jedoch die imperativen Sprachanteile, die nötig sind, um den veränderlichen internen Zustand der Objekte zu beschreiben. Die polymorphe Typinferenz funktionaler Sprachen funktioniert, solange es nur unveränderliche Werte gibt. Die Mischung von polymorphen Typsystemen und imperativen Sprachanteilen führt anscheinend zwangsläufig zu einer wesentlichen Komplizierung des Typsystems.

Als Zielsprache für die Verschmelzung von Objektorientierung und polymorpher Typinferenz wählten wir O'SMALL [38], da es frei von unnötigen Details ist, aber alle für uns wichtigen Aspekte enthält. O'SMALL ist zunächst ungetypt und besitzt keine Typdeklarationen. Es müssen also alle Typen inferiert werden. Da die Semantik der Sprache von der statischen Typüberprüfung unabhängig ist, kann man letztere als eine optionale Analyse betrachten. Programme, die abgelehnt werden, können trotzdem

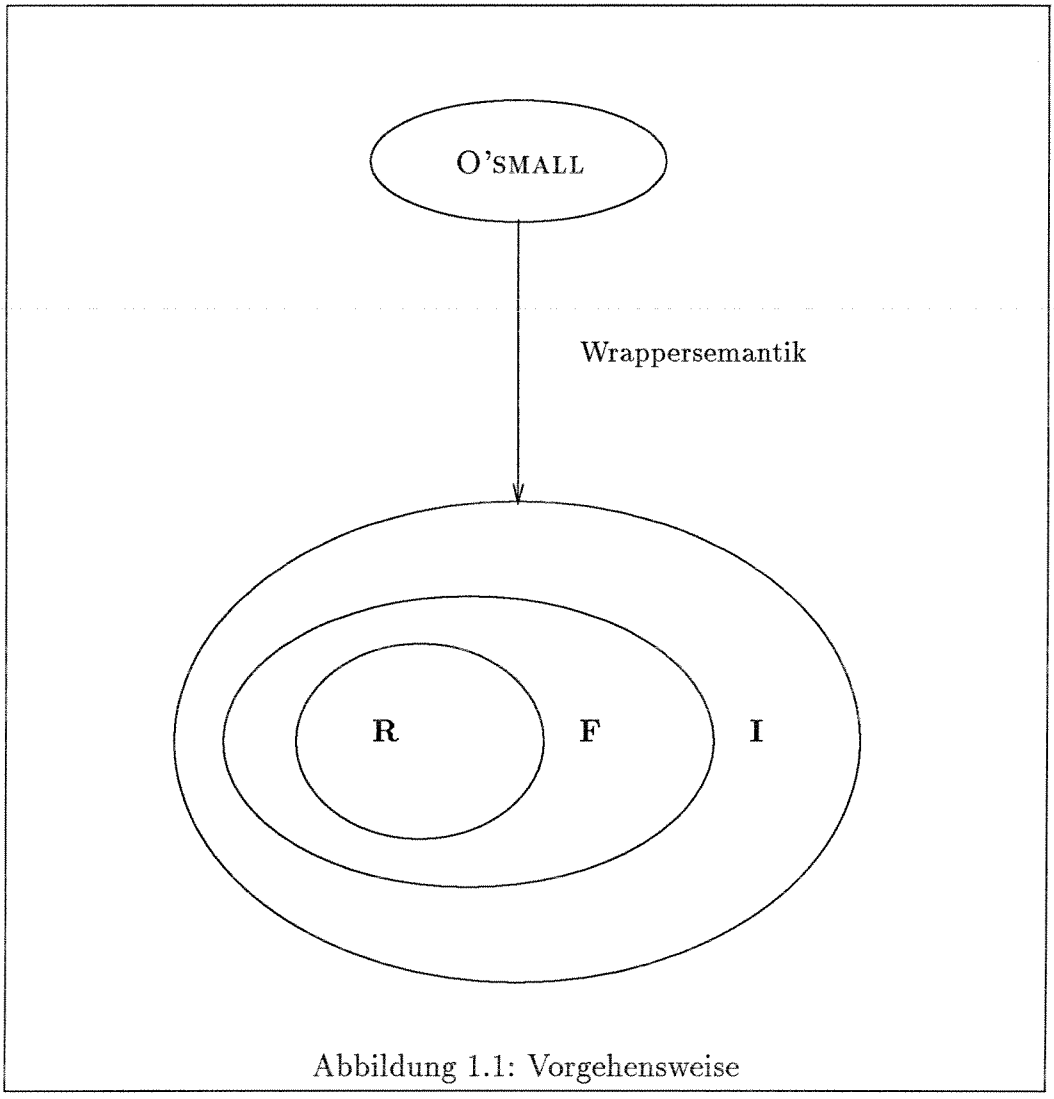


Abbildung 1.1: Vorgehensweise

benutzt werden – allerdings gewissermaßen auf eigene Gefahr. Die vom Typinferenzer erzeugte Typinformation ist sowohl für akzeptierte als auch für zurückgewiesene Programme eine wertvolle Dokumentation.

> *Das Ziel dieser Arbeit ist also ein polymorphes Typinferenzsystem für* O'SMALL, *das alle Typen inferiert und möglichst viele sinnvolle Programme akzeptiert.*

Der scheinbar direkteste Weg, dieses Ziel zu erreichen, besteht in der Angabe einer Menge von Typinferenzregeln: *eine* Regel für jede Sprachkonstruktion von O'SMALL. Obwohl der Sprachumfang von O'SMALL sehr klein ist, verbleiben noch relativ viele Konstruktionen, wenn es um Typinferenzregeln und die Induktionsbeweise ihrer Korrektheit geht. Wir haben daher einen anderen Weg gewählt, der in Abb. 1.1 gezeigt ist. O'SMALL wird in die einfachere Sprache **RFI** übersetzt, die Verbunde, Funktionen und imperative Sprachanteile enthält. Wir inferieren und überprüfen die Typen dann für die Übersetzung. Es bleibt beim Übersetzungsprozess genug Information erhalten, um die Typinformation zur Dokumentation der Quellprogramme einsetzen zu können. Ein weiterer Vorteil dieser Vorgehensweise ist die Nähe der Sprache **RFI** zum λ-Kalkül. Für letzteren gibt es einen reichhaltigen Literaturschatz über Typinferenz, auf den wir uns berufen können.

Wir führen O'SMALL und die Grundidee der Klassenvererbung anhand des Beispielprogramms aus Abb. 1.2 ein. Anschließend gehen wir auf die Typen ein, die von unserem Typinferenzer für die Variablen des Programms inferiert werden. In dem Programm werden Punkte und Kreise mit kartesischen Koordinaten implementiert. Dazu benutzt man zwei Klassendefinitionen: Die Klasse `Punkt` erbt von der leeren Klasse `Base`, und die Klasse `Kreis` erbt von `Punkt`.

Objekte der Klasse `Punkt` haben zwei Instanzvariablen, die die kartesischen Koordinaten des Punktes enthalten. Kreiert man einen Punkt (d.h. ein Objekt der Klasse `Punkt`) mit der Operation `new`, so befindet er sich zunächst im Ursprung, da seine Instanzvariablen jeweils mit Null initialisiert werden. Zwei Methoden, `x` und `y`, machen die nach außen sonst unsichtbaren Werte der Instanzvariablen sichtbar. Die Methode `verschiebe` ändert die Position des Empfängers. In der objektorientierten Sprechweise steht der Ausdruck `p.m(a)` für das Senden der Nachricht `m` mit dem

```
class Punkt inheritsFrom Base
def var xKomp := 0
    var yKomp := 0
in meth x() xKomp
   meth y() yKomp
   meth verschiebe(X,Y)
            xKomp := X+self.x;
            yKomp := Y+self.y
   meth abstVomUrspr()
            sqrt(sqr(self.x) + sqr(self.y))
   meth naeherAmUrspr(punkt)
            self.abstVomUrspr < punkt.abstVomUrspr
ni

class Kreis inheritsFrom Punkt
def var radius := 0
in meth r() radius
   meth setzeR(r)  radius := r
   meth abstVomUrspr()
            max(0, super.abstVomUrspr - self.r)
ni

def var p := new Punkt
    var k := new Kreis
in p.verschiebe(2,2); k.verschiebe(3,3); k.setzeR(2);
   p.naeherAmUrspr(k);                    {ergibt FALSE}
   p.verschiebe(0,-2); k.verschiebe(0,-2);
   p.naeherAmUrspr(k)                     {ergibt FALSE}
ni
```

Abbildung 1.2: O'SMALL-Program mit Punkten und Kreisen

aktuellen Parameter `a` an den Empfänger `p`. Zum Beispiel "verschiebt" `k.verschiebe(3,3)` den Kreis `k` um einen bestimmten Betrag. Es gibt weiterhin eine Methode für den Abstand vom Ursprung und eine Methode, die den Empfänger der Nachricht und den aktuellen Parameter in bezug auf ihren Abstand vom Ursprung vergleicht. Beim Aufruf einer parameterlosen Methode lassen wir die leeren Klammern weg.

Die Klasse `Kreis`, die die Instanzvariablen und Methoden der Klasse `Punkt` erbt, hat eine weitere Instanzvariable für den Radius und Methoden zum Lesen und Verändern desselben. Außerdem wird die Methode `abstVomUrspr` neu definiert. Bei dieser Neudefinition wird die Definition der Oberklasse mit `super.abstVomUrspr` angesprochen. Auf diese Weise kann die gerade überschriebene bzw. zu überschreibende Definition noch verwendet werden. Die Methode `naeherAmUrspr` wird ererbt und nicht neu definiert. Trotzdem wird sie, dank der späten Bindung, auf konsistente Weise übernommen: Betrachten wir den Rumpf der Methode `naeherAmUrspr`: die Nachricht `abstVomUrspr` wird an die Pseudovariable `self` gesandt. Wenn der Empfänger der ursprünglichen Nachricht – also der mit `self` bezeichnete – ein Kreis ist, wird die neu definierte `abstVomUrspr`-Methode gewählt, obwohl `naeherAmUrspr` nicht umdefiniert worden ist! Dies nennen wir späte Bindung (late binding).
Kommen wir nun zu den inferierten Typen der Objekte `p` und `k`.

$$\mathtt{p} \;:\; \left\langle \begin{array}{rcl} \mathtt{x} & : & \text{num} \\ \mathtt{y} & : & \text{num} \\ \mathtt{verschiebe} & : & \text{num} \to \text{num} \to \text{unit} \\ \mathtt{abstVomUrspr} & : & \text{num} \\ \mathtt{naeherAmUrspr} & : & \langle \alpha \mid \mathtt{abstVomUrspr} : \text{num} \rangle \to \text{bool} \end{array} \right\rangle$$

$$\mathtt{k} \;:\; \left\langle \begin{array}{rcl} \mathtt{x} & : & \text{num} \\ \mathtt{y} & : & \text{num} \\ \mathtt{r} & : & \text{num} \\ \mathtt{setzeR} & : & \text{num} \to \text{unit} \\ \mathtt{verschiebe} & : & \text{num} \to \text{num} \to \text{unit} \\ \mathtt{abstVomUrspr} & : & \text{num} \\ \mathtt{naeherAmUrspr} & : & \langle \beta \mid \mathtt{abstVomUrspr} : \text{num} \rangle \to \text{bool} \end{array} \right\rangle$$

Verbundtypen schreiben wir als die Liste der Komponenten des Verbundes mit den Typen der Komponenten und eventuell einer Erweiterung. Letztere wird links eines senkrechten Striches als sogenannte Erweiterungsvariable aufgeführt. Sie steht für die unendliche Menge von Komponentennamen, die nicht explizit erwähnt werden. Der aktuelle Parameter der Methode `naeherAmUrspr` kann außer der Komponente `abstVomUrspr` noch weitere Komponenten haben. Hat ein Verbundtyp keine Erweiterungsvariable, so bedeutet dies, daß er nicht erweiterbar ist. Der Typ 'unit' enthält nur ein Element. Dieses Element wird als Resultat von Sprachelementen geliefert, die nur einen Seiteneffekt bewirken sollen. Da die Typinferenz auf der Sprache **RFI** durchgeführt wird, erscheinen die Typen der Methoden in curryfizierter Form, obwohl die Parameter in Tupeln auftreten. Der Typ von `x` ist z.B. in Wirklichkeit unit→num, aber, da die Klammern beim Aufruf weggelassen werden können, lassen wir auch den Typ auf der linken Seite des Pfeils weg.

Einige Eigenschaften der Sprache O'SMALL sind für die Typinferenz von Bedeutung:

- *Zustand*: Objekte haben Instanzvariablen, deren Werte durch Zuweisungen verändert werden können. Diese Variablen sind nur in der Klasse, in der sie deklariert werden, sichtbar (eingekapselte Instanzvariable). Ungewöhnlicher ist der Initialisierungszwang jeder Variable eines Programms. In vielen anderen Sprachen beispielsweise haben uninitialisierte Variable den Wert 'nil'.

- *Klassen*: Klassen werden am Anfang des Programms deklariert. Sie sind keine Objekte, d.h. sie können nicht Resultate von Berechnungen sein.

- *Vererbung*: O'SMALL hat einfache Vererbung à la SMALLTALK [31] mit Pseudovariablen `self` und `super`. Eine ebenfalls hier vorgestellte Erweiterung um explizite Wrapper ermöglicht jedoch die Modellierung einer Teilklasse der mehrfachen Vererbung.

- *Parameterübergabe*: Parameter werden als Wertparameter (*call-by-value*) übergeben. Objekte sind Werte. Sie werden bei der Übergabe nicht kopiert.

- *Rekursion*: Deklarationen enthalten keine direkte Rekursion. Sie wird erreicht, indem man Nachrichten an `self` schickt.

1.1 Überblick

Ein Gesamtbild der Arbeit befindet sich in Abb. 1.1 (Seite 5). O'SMALL wird nach **RFI** übersetzt, welches selbst zwei Teilsprachen besitzt.

Da die verwendeten Typen komplizierter sind als gewöhnlich, wird ihnen ein eigenes Kapitel gewidmet. Sie werden dort mit ordnungssortierter Logik formalisiert.

Kapitel 4 stellt **R**, eine Verbundsprache, vor. Die Semantik wird mithilfe eines Termersetzungssystems angegeben. Ein Typinferenzalgorithmus wird angegeben.

Kapitel 5 stellt **RF**, eine Erweiterung von **R** um den λ-Kalkül und 'let', vor. Die Semantik wird mithilfe eines Reduktionssystems, das das Termersetzungssystem aus Kapitel 4 erweitert, angegeben. Zwei äquivalente Algorithmen für die Typinferenz mit 'let' schlagen die Brücke vom formalen Teil der Arbeit zum praktischen. Eine Typdisziplin mit Typschemata wird vorgestellt. Sie bildet den Ausgangspunkt für das nächste Kapitel.

Kapitel 6 stellt **RFI**, eine Erweiterung von **RF** um imperative Sprachanteile vor. Die Semantik wird durch Inferenzregeln (*natürliche Semantik*) angegeben. Die Typinferenz wird den imperativen Konstrukten angepaßt.

Kapitel 7 enthält die Sprachdefinition von O'SMALL und die Übersetzung in **RFI**, die auf der Wrappersemantik beruht. Die Übersetzungsfunktion benutzt eine einfache Analyse, die es erlaubt, den Grad des Polymorphismus der Zielprogramme zu maximieren. Des weiteren werden in diesem Kapitel die Resulate der Typinferenz anhand von O'SMALL-Programmen bewertet.

2. Introduction

There is a growing demand in the world for solutions which computers provide, and we need the capability of developing big software systems that are controllable, maintainable, and expandable. It is the task of computer science to provide methods and tools that will help us in this respect. Programming is often called an art, and creativity and inspiration may yield beautiful programs. On the other hand, functionality plays a major rôle in this context and, thus, programming is rather an engineering art. Therefore, we also speak of *software engineering.* Software engineering divides the development of a system into the three phases analysis, design, and implementation. Programming languages are central in software engineering because not only do they completely determine implementation but they also have a strong influence on the first phases. Object-oriented analysis and design [5, 19] were inspired by object-oriented programming languages. In any case, the decisions taken in the analysis and design phases must be expressible in the programming language.

The object-oriented approach promises alleviation of these problems. Object-oriented languages such as SMALLTALK [31] undoubtedly have the reputation of leading to controllable, maintainable, and expandable systems. However, they are not regarded as very safe in the sense that object-oriented programs are free of errors. They are well-suited for rapid prototyping, not for end products. These reservations may be due, on the one hand, to the efficiency loss of nowadays compilers. On the other hand, the safety issue may be raised by the absence of static typing, which means that type errors are not recognized until run time.

Before continuing our discussion, let us explain some terminology. Our notion of *object-oriented programming* [103] is the following: the concepts of *objects*, *object classes*, and *class inheritance* must be supported. An object has a set of operations, called *methods*, and an internal state contained in so-called *instance variables. Data abstraction* is gained by forbidding direct access to instance variables. Objects communicate with each other by sending messages. The result of a message sent to an object (the *receiver*) is not completely determined by the actual parameters, but depends on the state of the receiver. Objects can be modeled as records of methods, and message sending as *record selection.* Object classes can serve as templates for creating objects. They specify operations and may also contain their implementation. Class inheritance is a mechanism for the composition of

specifications and implementations. There is late binding for operations modified in subclasses. Note that the language in this work does not contain any specifications.

A *type error* occurs when values meet that do not match. An example is the addition of an integer and a boolean. In object-oriented languages, type errors consist mainly of messages that are not "understood" by their receiver.

Let us now continue the discussion. If type errors are not discovered before run time, a programming language deserves the predicate 'unsafe'. Many errors occur for the first time when the data base management system has already been sold or the probe is already on its way into space. Exhaustive testing or complete verification are impossible in practice. Therefore, type errors should be recognized at compile time. Static type checkers usually guarantee that an accepted program does not produce type errors at run time. Almost all interesting problems in computer science are undecidable and so is the question of whether a program is free of type errors for all inputs. This implies that static type systems can only express sufficient conditions which again means that they can only accept a subset of type error free programs. Thus, safety must be paid for with less flexibility in programming. It is the task of research in type systems to keep the price as low as possible.

We often use the notions of '*type checking*' and '*type inference*' synonymously although they are nuanced. Type checking takes a program – possibly with full type declarations – and decides whether the type information is correct. Type inference gathers missing type information from the context. Therefore, type checking is always a part of type inference. We differentiate between *polymorphic* and *monomorphic* type systems. Classic imperative languages like C [54] and Pascal [51] have a monomorphic type system which is rather inflexible. C and the object-oriented language C++ [90] get the necessary flexibility but are not type safe. In a monomorphic type system, there are base types (integers, booleans, ...) and types that can be constructed from these (pairs of integers, lists of booleans, ...). If, in one of those languages, one has a procedure for sorting lists of integers one must copy and modify the code in order to obtain a procedure for sorting lists of booleans. In polymorphically typed languages one can use *one* sorting procedure for all data types that are totally ordered.

The author has learned to appreciate the advantages of object-oriented languages [31] and polymorphic type systems [35, 97] in various projects [17, 18, 37, 41, 109]. The idea of combining both offered itself. However, the combination turned out to be much more difficult than expected: records had to be treated in a flexible way in order to model objects and inheritance and late binding had to be accounted for. Yet, the hardest problem seemed to be caused by the imperative features of object-oriented programming languages. The polymorphic type systems of functional languages rely on the fact that variables cannot change their values. Apparently, the combination of polymorphic type systems and imperative features leads to a considerable complication.

As a target language for the amalgamation of object orientation and polymorphic type inference we chose O'SMALL [38] because it is free of superfluous details and contains all necessary elements. O'SMALL is untyped and possesses no type declarations, which is why all types must be inferred. Since the semantics of the language is independent of static type checking we can consider the latter as an optional analysis. Rejected programs can still be used – at one's own risk of course. In other words, we have a descriptive [78, 76, 77] type system, *not* a prescriptive one. The type information created by the type inferencer is valuable equally well for rejected as for accepted programs.

> *The goal of this work is a polymorphic type inference system for* O'SMALL *which infers* all *types and accepts as many sensible programs as possible.*

How can we achieve our goal? One way is giving type inference rules directly for O'SMALL language constructs. This would be complicated since O'SMALL, despite its small size, still has many syntactic constructions. The way we have chosen is shown in Fig. 2.1. We translate O'SMALL into a simpler language containing records, functions, and imperative features. The language is called **RFI**. We infer and check the types of the translation. We retain enough information in the translation process so as to be able to gain useful documentation for the source program. Another advantage of formulating type inference on **RFI** is the similarity of this language to the λ-calculus. For the latter, we can rely on a treasure of literature on the topic.

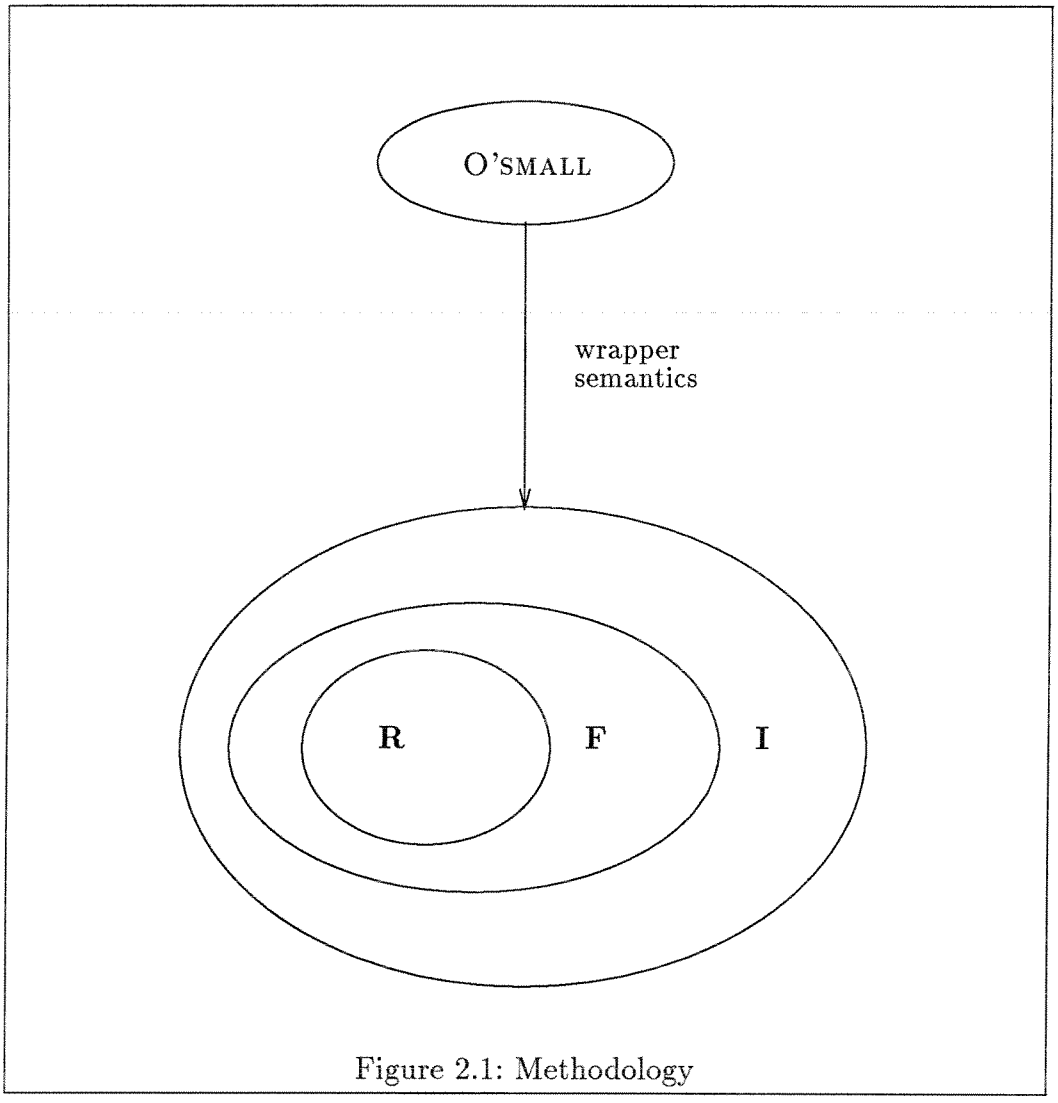


Figure 2.1: Methodology

```
class Point inheritsFrom Base
def var xComp := 0; var yComp := 0
in meth x() xComp
   meth y() yComp
   meth move(X,Y)
            xComp := X+self.x;
            yComp := Y+self.y
   meth distFromOrg()
            sqrt(sqr(self.x) + sqr(self.y))
   meth closerToOrg(point)
            self.distFromOrg < point.distFromOrg
ni

class Circle inheritsFrom Point
def var radius := 0
in meth r()          radius
   meth setR(r)      radius := r
   meth distFromOrg()
            max(0, super.distFromOrg - self.r)
ni

def var p := new Point;
    var c := new Circle
in p.move(2,2); c.move(3,3); c.setR(2);
   output p.closerToOrg(c);            {results in FALSE}
   p.move(0,-2); c.move(0,-2);
   output p.closerToOrg(c)             {results in FALSE}
ni
```

Figure 2.2: O'SMALL program with points and circles

We introduce O'SMALL and the idea of class inheritance by way of an O'SMALL program that can be found in Fig. 2.2. We examine the types that our type inferencer produces for this example program. The program is about points and circles in two-dimensional space. There are two class definitions: `Point` inherits from `Base` and `Circle` from `Point`. The class `Base` is a class "without contents".

Objects of class `Point` have two instance variables representing the Cartesian coordinates of the point. A point object created with `new` is at the origin, because its instance variables are initialized to zero. There are two methods – `x` and `y` – for inspecting the instance variables otherwise invisible from the outside. The method `move` changes the position of the receiver. In object-oriented terminology, the O'SMALL expression `p.m(a)` stands for sending of message `m` with argument `a` to receiver `p`, e.g. `c.move(3,3)` moves a circle object `c` by a certain amount. There is a method for calculating the distance from the origin and a method that returns `TRUE` if the receiver is closer to the origin than the argument. We can omit the empty parentheses when calling a method without parameters.

The class `Circle`, which inherits instance variables and methods from `Point`, has an additional instance variable for the radius, methods for reading and changing the radius, and it redefines `distFromOrg`. For the redefinition of the latter, the corresponding definition of the super class is referred to by `super.distFromOrg`. Thus, we can still retrieve what is just being redefined. The inherited method `closerToOrg` has not been redefined in the class `Circle` and does not have to be redefined in order to be consistent. In the body of `closerToOrg`, the message `distFromOrg` is sent to `self`. If the receiver of a `closerToOrg`-message is a circle, the redefined `distFromOrg`-method is chosen, although `closerToOrg` has not been redefined. This is called *late binding.*

The types of the point object `p` and the circle object `c` of Fig. 2.2 are as follows.

$$
\mathtt{p} \;:\; \left\langle \begin{array}{rcl} \mathtt{x} & : & \text{num} \\ \mathtt{y} & : & \text{num} \\ \mathtt{move} & : & \text{num} \to \text{num} \to \text{unit} \\ \mathtt{distFromOrg} & : & \text{num} \\ \mathtt{closerToOrg} & : & \langle \alpha \mid \mathtt{distFromOrg} : \text{num} \rangle \to \text{bool} \end{array} \right\rangle
$$

$$
\mathtt{c} \;:\; \left\langle \begin{array}{rcl} \mathtt{x} & : & \text{num} \\ \mathtt{y} & : & \text{num} \\ \mathtt{r} & : & \text{num} \\ \mathtt{setR} & : & \text{num} \to \text{unit} \\ \mathtt{move} & : & \text{num} \to \text{num} \to \text{unit} \\ \mathtt{distFromOrg} & : & \text{num} \\ \mathtt{closerToOrg} & : & \langle \beta \mid \mathtt{distFromOrg} : \text{num} \rangle \to \text{bool} \end{array} \right\rangle
$$

Record types are denoted by their list of components with the components' types and, optionally, an ellipsis. An ellipsis is labeled with a *row variable* on the left-hand side of a bar. The row variable stands for the infinite set of labels that are not mentioned explicitly. The actual parameter of the method `closerToOrg` may have more than a `distFromOrg` field. The absence of an ellipsis means that the record type is closed. The type 'unit' corresponds to the domain with one element. It is the result type of an "assignment expression". Such an expression is called "statement" in the programming language literature. If a method ends with an assignment, it has a type ending with unit. As type checking is performed on **RFI**, the types of methods appear in a curried version, although parameters are tuples in O'SMALL programs. The type of `x` is in fact unit→num, but, since the parentheses can be omitted when calling this method, we also omit the type on the left-hand side of the arrow. One could consider omitting the parentheses in parameterless method declarations as well.

Some properties of O'SMALL are important for type checking:

- *State*: Objects have assignable instance variables visible only in the class where they are declared (*encapsulated instance variables*). A less common feature is that every variable has to be initialized. This

contrasts to many other languages where uninitialized variables have the value *nil*.

- *Classes*: Classes are declared at the beginning of a program. They are not objects, i.e. they cannot be arguments of functions etc.
- *Inheritance*: O'SMALL has single inheritance à la SMALLTALK [31] using *pseudo variables* `self` and `super`. An extension to inheritance with explicit wrappers [39] permitting the modeling of certain cases of multiple inheritance is possible.
- *Parameter passing*: O'SMALL passes arguments *by value*. Objects are first-class citizens. They are not copied when they are passed.
- *Recursion*: There is no direct recursion in declarations. Recursion (including mutual recursion) is achieved by sending messages to `self`.

2.1 Overview

For a picture of the whole, refer to Fig. 2.1 on page 15. O'SMALL is translated into **RFI**, which itself contains two sublanguages. The organization is bottom-up.

Since our types are more complicated than usual, the first chapter is dedicated entirely to their formalization in order-sorted logic.

Chapter 4 introduces the language **R**, a language of records. Its semantics is given by a term rewriting system. A type inference algorithm that infers principal types[1] is given.

Chapter 5 introduces the language **RF**, an extension of **R** by the λ-calculus and 'let'. Its semantics is given by a reduction system extending the term rewriting system of chapter 4. Two equivalent algorithms for inferring types for 'let' bridge the gap between the formal and the practical part of this work. A type discipline using type schemes is presented. This type discipline will be the basis of the one in the next chapter.

[1] Principal types are also called "most general types".

Chapter 6 introduces the language **RFI**, an extension of **RF** by imperative features. Its semantics consists of inference rules (*natural semantics*). Imperative features receive a special treatment.

Chapter 7 defines the object-oriented language O'SMALL and its translation into **RFI** based on wrapper semantics. The translation function uses a simple analysis in order to maximize the degree of polymorphism in the target programs. The type inference system is assessed in terms of O'SMALL-programs.

3. Record Types as Feature Trees

Record calculi can be the basis for modeling objects in object-oriented languages. The type inference problem for those languages is still awaiting a satisfactory solution. Since systems with a general subtyping notion [2, 8, 10, 13, 14, 50, 89] are problematic, especially when confronted with imperative features [13], record types with row variables as developed by Wand [101, 102] and Rémy [79] are worth studying. Wand's record language possesses general concatenation. His record types are highly intuitive. However, he cannot infer principal types. Rémy's record language possesses record adjunction instead of concatenation. He can infer principal types but his record types are less intuitive.

We have studied record calculi with the overall aim of inferring types for O'SMALL [38]. O'SMALL is a class-based object-oriented language where classes are not first-class citizens. Record concatenation or adjunction are needed for modeling class inheritance. However, if classes are not first-class citizens the compiler can statically resolve all concatenations and adjunctions. The underlying record calculus can be reduced to "records and selection". We can use simple record types and have the principal type property at the same time.

Let us show by a simple example how record types with row variables are used. In the expression $1 + x.a$, the label a is selected on the variable x. The result of this selection is the argument of an integer addition. What is the type of x? We know that x must be a record with an a-component. The a-component must be an integer. Our first guess for the type of x is $\langle a : \text{int} \rangle$, a closed record type. Although this type is certainly correct it is not very flexible, we could even say that it is completely user-unfriendly.[1] The record type[2] that we infer for x is $\langle \alpha \mid a : \text{int} \rangle$. The variable α on the left-hand side of the bar is a so-called *row variable*. It stands for the other components that x may have.

Recursive types are necessary because of the peculiarities of the object-oriented programming style. The example $(x.a)x$ can be read as "send the message a to x with the argument x". This could be the object-oriented way

[1] Note that we have no subtyping. In a system with subtyping, the structured subtyping rules on records would allow us to have an x with further components. E.g. $\langle a : \text{int}, b : \text{bool} \rangle$ would be a subtype of $\langle a : \text{int} \rangle$.

[2] We are informal here; later, we will distinguish between types and type terms. Furthermore, the meaning of variables in type terms will be formalized.

of writing the addition $x + x$. The type inferred for x is $\mu\alpha.\langle\beta \mid a : \alpha \to \gamma\rangle$, a recursive type.

In ordinary type systems with finite types and no equality laws, one can identify types and type terms. We have already given an example of why we need infinite types and we will see that we have to impose equality laws on types as well. Therefore, we distinguish between *types* and *type terms*. Types are possibly infinite rational feature trees as introduced in section 3.1. Type terms come from a language containing primitives, an arrow, and records with disjoint adjunction as introduced in section 3.2. In section 3.3 we present a first-order theory over type terms and show that types are a model of it.

3.1 Types

Types are *feature trees* [87]. Fig. 3.1 shows examples of the types we have in mind. The nodes of the trees are labeled with *constructors* for function types, record types, and primitive types. The edges of the trees are labeled with so-called *features*. Depending on the constructor at a node, there are restrictions w.r.t. the number of subtrees and the features below that node. The arrow stands for function types and must have exactly two subtrees at features 1 and 2. The constructor r stands for record types. It may have any number of subtrees (including zero) at features coming from an infinite set $a, b, c, \ldots$ of record labels. Last but not least, there is a finite set of constructors standing for primitive types. Nodes labeled with these constructors must not have subtrees.

The last tree in Fig. 3.1 is the finite representation of an infinite tree. For reasons already stated above we want to admit infinite trees of that form. In the literature, they are called *rational trees* [22, 59] or *regular trees*.

We will describe types by first-order formulae over an order-sorted signature **RT**. We will build a corresponding theory RT. With this in mind, we now set up a first-order structure $\mathcal{T}$ (RT's standard model) whose universe is the set of all regular feature trees. We proceed along the lines of [87]. Basic definitions concerning feature trees can be found there or in section A.1.1.

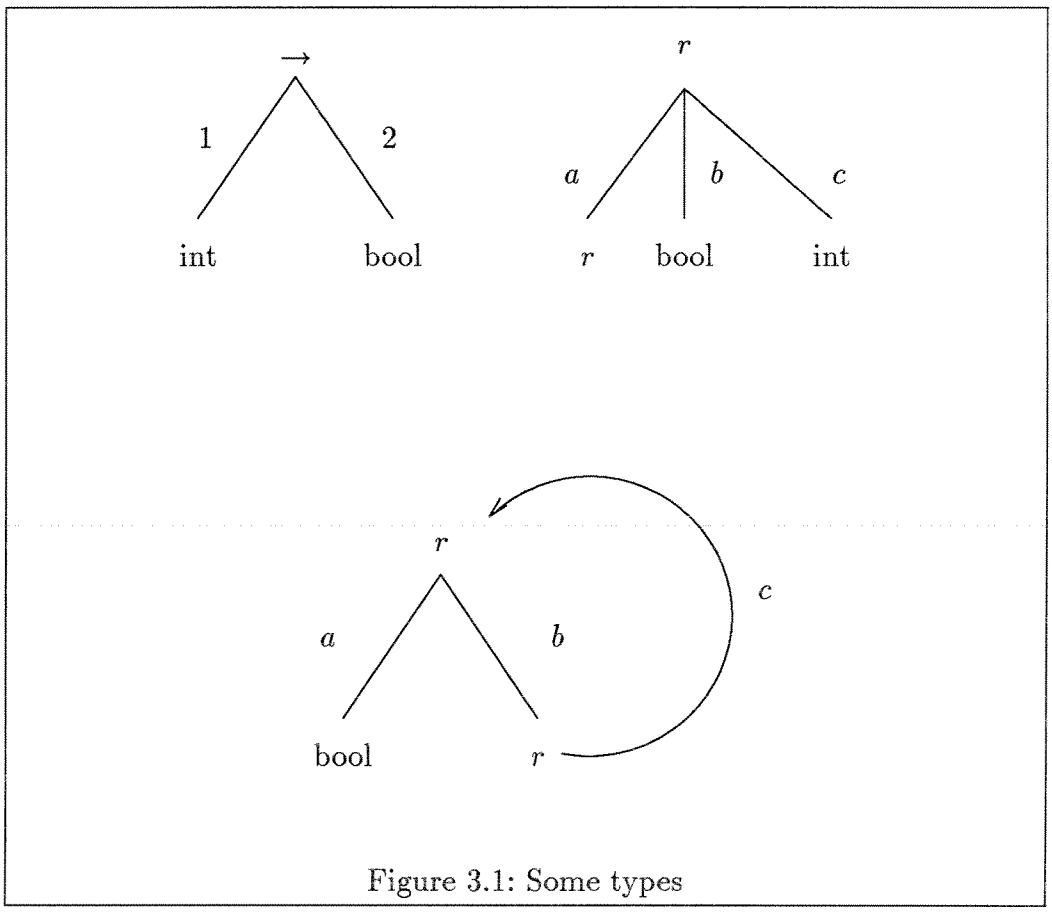


Figure 3.1: Some types

Let CON $= \{r, \rightarrow\} \cup$ PRIM be a finite alphabet of *constructors* with $\{r, \rightarrow\} \cap$ PRIM $= \emptyset$. We use the letter c to denote constructors. PRIM contains constructors for primitive types such as int, bool, and unit. Let FEA $= \{1, 2\} \cup$ LAB be an alphabet of *features* with $\{1, 2\} \cap$ LAB $= \emptyset$. We use letters f and g to denote features, F and G to denote sets of features, and a and b to denote labels. LAB is a countably infinite set of record *labels*.

Definition 3.1.1 (feature tree structure) We define the *feature tree structure* $\mathcal{T}$.

- The universe T of $\mathcal{T}$ is the set of all rational feature trees with the following restrictions:
  - $\sigma(\epsilon) = \rightarrow$ if and only if $dom(\sigma) \cap$ FEA $= \{1, 2\}$

  - $\sigma(\epsilon) = r$ if and only if $dom(\sigma) \cap \text{FEA} \subset \text{LAB}$
  - if $\sigma(\epsilon) \in \text{PRIM}$ then $dom(\sigma) = \emptyset$
- $\sigma \in c^{\mathcal{T}}$ if and only if $\sigma(\epsilon) = c$ (i.e. σ's root is marked with c),
- $(\sigma, \tau) \in f^{\mathcal{T}}$ if and only if $f \in dom(\sigma)$ and $\tau = f\sigma$ (i.e. τ is a direct subtree of σ at f).
- $\sigma \in F^{\mathcal{T}}$ if and only if $dom(\sigma) \cap \text{FEA} = F$

3.2 Type Terms

Since primitive types and function types are well-known, we focus on record type terms. Our introductory example contained the type term $\langle \alpha \mid a : \text{int} \rangle$. The vertical bar stands for *disjoint adjunction*, i.e. the labels on the right-hand side are disjoint from those on the left-hand side.[3] Our type term contains a variable α on the left-hand side of the bar. Type variables in this position are called *row variables* in the literature [79, 102]. For guaranteeing that α cannot be instantiated to a term that contains label a on top-level – this would be a violation of the disjointness condition – *many-sorted logic* would be useful. In many-sorted logic, every variable has a fixed sort.

Sorts are concerned with sets of labels. Sets of labels are partially ordered by the inclusion relation. Thus, it is only natural to use *order-sorted logic*, an extension of many-sorted logic. Order-sorted logic is able to reflect this partial order. We will see that order-sorted logic permits a natural and concise formalization of all problems to come in chapters 4 and 5.

The necessity to exclude finite sets of labels from row variables led us to order-sorted logic. Once we had introduced its machinery we recognized that we could encode even more information in the sorts. After all, not only (row) variables have sorts but every well-sorted term. We will be rewarded for the additional effort in the formulation of the type inference algorithm in section 4.3.

Definition 3.2.1 The set of *sorts* S contains:

- for every type constructor $c \in \text{PRIM}$, there is a *primitive sort* $\mathbf{p}_c$.

[3] This disjointness only holds for the top-level, not deeper in the terms.

- f the sort of function types,
- for every finite set of labels $A \subset \mathrm{LAB}$, A is the *closed (record) sort* of record types having exactly the labels in A on top-level,
- for all finite sets of labels $A \subset \mathrm{LAB}$ and $B \subset \mathrm{LAB}$, where $A \cap B = \emptyset$, (A, B) is the *open (record) sort* of record types having at least the labels in B but not those in A on top-level,
- $\top$ is the top sort. $\bot$ is the *inconsistent* sort (we call all other sorts *consistent*).

Basic definitions can be found in section A.1. The machinery for order-sorted logic and term rewriting can be retrieved from sections A.2 and A.3 respectively. There is also a notation index.

For our and, hopefully, the reader's convenience, we introduce a notation for finite sequences with indices like $a_1, \ldots, a_n$ or $a_1 : e_1, \ldots, a_n : e_n$ or $\alpha_1 \doteq \beta_1, \ldots, \alpha_n \doteq \beta_n$. We abbreviate these terms to $\overline{a}$, $\overline{a} : \overline{e}$, or $\overline{\alpha} \doteq \overline{\beta}$ respectively. Sometimes, a non-negative integer n or m will be referred to in the context of our abbreviating notation for sequences, although it is not made explicit in the notation.

We assume that the set of labels LAB is totally ordered ($\leq$). A record type term with labels $a_1, \ldots, a_n$ and entries $\sigma_1, \ldots, \sigma_n$ is denoted as $\langle \overline{a} : \overline{\sigma} \rangle$. In this notation, the labels $a_1, \ldots, a_n$ are always distinct. n is not mentioned explicitly and we assume that $n \geq 0$. Therefore, this notation includes the empty record type term, which is also denoted as $\langle\rangle$.

Definition 3.2.2 (Signature RT) We define the signature **RT** of type terms. Type terms are ranged over by variables $\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \ldots$ There are infinitely many variables of each consistent sort. In the following subsort declarations, A, B, C, and D are finite sets of labels.

$$
\begin{aligned}
\mathrm{s} &< \top && \text{for all sorts s} \\
\bot &< \mathrm{s} && \text{for all sorts s} \\
A < (B, C) &\Leftrightarrow && A \supseteq C, A \cap B = \emptyset \\
(A, B) < (C, D) &\Leftrightarrow && A \supseteq C, B \supseteq D
\end{aligned}
$$

The constructors for type terms build primitives, functions, records, and

disjoint adjunctions of records:[4]

$$
\begin{array}{rcll}
p_c & : & \mathbf{p}_c & \{\text{for all } c \in \text{PRIM} \\
. \rightarrow . & : & \top \times \top \rightarrow \mathbf{f} & \\
\langle \overline{a} : \dot{:} \rangle & : & \overline{\top} \rightarrow \{\overline{a}\} & \\
\langle . \mid \overline{a} : \dot{:} \rangle & : & (A, B) \times \overline{\top} \rightarrow (A \setminus \{\overline{a}\}, B \cup \{\overline{a}\}) & \begin{cases} \{\overline{a}\} \neq \emptyset \\ \{\overline{a}\} \subseteq A \end{cases}
\end{array}
$$

We have already mentioned that sorts express more than the absence of labels. Record sorts do not only express which labels are excluded but also which labels are included. In the open record sort (A,B), A is the set of labels that the record type term has not, B is the set of labels that it has at least. In the following table, we list some type terms and their potential sorts.[5]

type	sort
$\langle a : \text{int} \rangle$	$\{a\}$
α	$(\{a\}, \emptyset)$
$\langle \alpha \mid a : \text{int} \rangle$	$(\emptyset, \{a\})$
β	$(\{a, b\}, \emptyset)$
$\langle \beta \mid a : \text{int} \rangle$	$(\{b\}, \{a\})$

The first line is simply a record type term of a closed record sort. Assume that the row variable α has a sort that excludes label a. If this label were not excluded, α could not be on the left-hand side of the bar in the adjunction below. Since the variable α does not guarantee any labels, the second component of its sort is empty. The last two lines show how open record sorts where both components are non-empty sets come into being.

Proposition 3.2.3 The signature **RT** is regular.

It is sometimes interesting to know whether two sorts have a common subsort. We will see that this is decidable. We denote the least partial order on the set of sorts S induced by the subsort declarations of Def. 3.2.2 by $\leq$. Overloading the symbol $<$, we write $a < b$ for $a \leq b$ and $a \neq b$.

[4] $\{\overline{a}\}$ denotes the set $\{a_1, \ldots, a_n\}$.

[5] Note that the second line is a prerequisite of the third line.

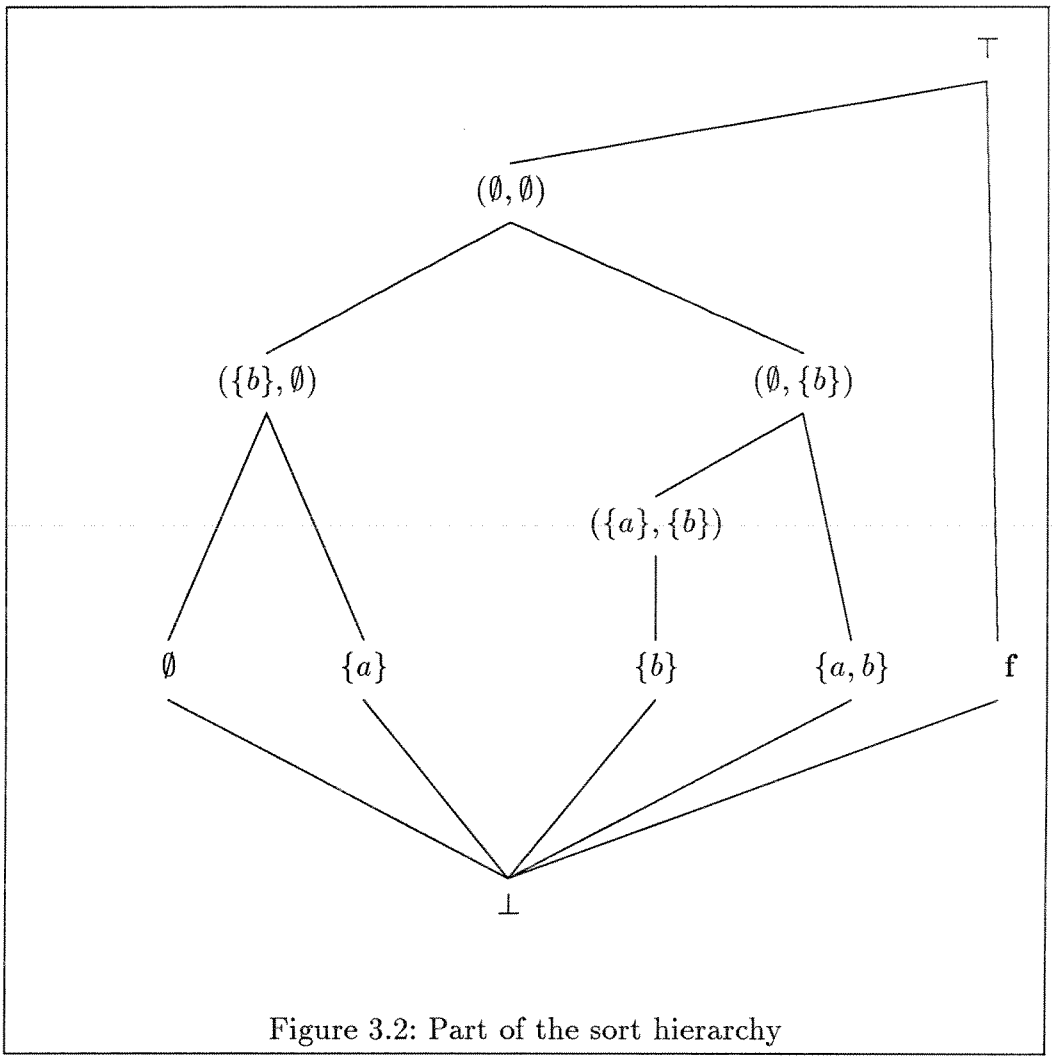


Figure 3.2: Part of the sort hierarchy

Proposition 3.2.4 $(S, \leq)$ is a lattice.

The next propositions show how to calculate the greatest lower bounds of two record sorts and when two sorts are incompatible.

Proposition 3.2.5

$$A \sqcap B = \bot \quad \Leftrightarrow \quad A \neq B \tag{3.1}$$

$$(A, B) \sqcap (A', B') = \bot \quad \Leftrightarrow \quad A \cap B' \neq \emptyset \vee A' \cap B \neq \emptyset \tag{3.2}$$

$$A \sqcap (B, C) = \bot \quad \Leftrightarrow \quad A \cap B \neq \emptyset \vee C \not\subseteq A \tag{3.3}$$

Proposition 3.2.6

$$(A,B)\sqcap(A',B') = \begin{cases} \bot & \text{if } A\cap B'\neq\emptyset \\ & \quad \vee A'\cap B\neq\emptyset \\ (A\cup A', B\cup B') & \text{otherwise} \end{cases} \tag{3.4}$$

$$A\sqcap(B,C) = \begin{cases} \bot & \text{if } A\cap B\neq\emptyset \vee C\not\subseteq A \\ A & \text{otherwise} \end{cases} \tag{3.5}$$

Fig. 3.2 shows part of the sort hierarchy. Just beneath the top element $\top$, there is the greatest record sort $(\emptyset,\emptyset)$. The sort $(\emptyset,\emptyset)$ contains all record types because it does not exclude any labels, the left-hand side is empty, and it does not enforce any labels, the right-hand side is empty, too. The open record sort $(\{b\},\emptyset)$ could be the sort of a row variable. Row variables, in general, exclude some labels but do not guarantee any. $(\emptyset,\{b\})$ is an open record sort that enforces the label b but does not exclude any labels. Beneath it, there is the sort $(\{a\},\{b\})$ which excludes label a and enforces label b. This could be the sort of an open record type having a fixed b-component and a row variable of the sort $(\{a,b\},\emptyset)$.

Proposition 3.2.7 The closed record sorts, the function sort, and the primitive sorts are exactly the minimal consistent ones.

We proceed by giving the denotations of our type terms.

Definition 3.2.8 We define an **RT**-algebra using the feature tree structure $\mathcal{T}$ of Def. 3.1.1.

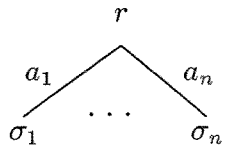
- For every sort $\mathbf{s}$ we give its denotation $\mathbf{s}^{\mathcal{T}}$.

$$\begin{aligned} \top^{\mathcal{T}} &= T \\ A^{\mathcal{T}} &= \{\sigma\in r^{\mathcal{T}} \mid f\in dom(\sigma)\Leftrightarrow f\in A\} \\ (A,B)^{\mathcal{T}} &= \{\sigma\in r^{\mathcal{T}} \mid f\in A\Rightarrow f\notin dom(\sigma),\ f\in B\Rightarrow f\in dom(\sigma)\} \\ \mathbf{f}^{\mathcal{T}} &= \{\sigma\in{\rightarrow}^{\mathcal{T}} \mid f\in dom(\sigma)\Leftrightarrow f\in\{1,2\}\} \\ \mathbf{p}_c^{\mathcal{T}} &= \{\sigma\in c \mid dom(\sigma)=\emptyset\} \qquad \text{for all } c\in \mathrm{PRIM} \end{aligned}$$

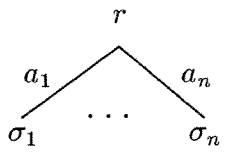
- It is easy to see that $\mathbf{r}<\mathbf{s}\in\mathbf{RT}$ implies $\mathbf{r}^{\mathcal{T}}\subseteq\mathbf{s}^{\mathcal{T}}$.

- We give the denotation of the function symbols.

$$\langle \overline{a} : \overline{\sigma} \rangle^{\mathcal{T}} \quad = \quad \overline{\sigma} \mapsto$$

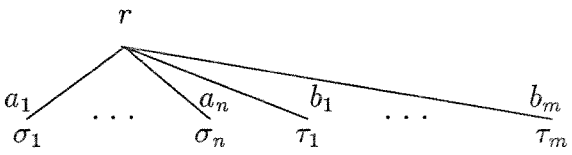


The sorts are constrained such that we can assume that σ has the form



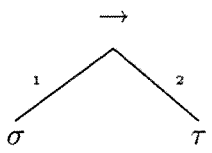
in the following equation, where $\{\overline{a}\} \cap \{\overline{b}\} = \emptyset$.

$$\langle \sigma \mid \overline{b} : \overline{\tau} \rangle^{\mathcal{T}} \quad = \quad \sigma, \overline{\tau} \mapsto$$



The denotation of $\rightarrow$ is

$$(\sigma \rightarrow \tau)^{\mathcal{T}} \quad = \quad \sigma, \tau \mapsto$$



For all $c \in \mathrm{PRIM}$, the denotation of $\mathbf{p}_c$ is

$$\mathbf{p}_c^{\mathcal{T}} \quad = \quad \mapsto \quad c$$

- It is easy to check that the sort constraints are fulfilled by the denotations of the function symbols.

Proposition 3.2.9 The denotation of an open sort can be represented by a partition of denotations of closed sorts.

Proof:

$$\begin{aligned}(A,B)^{\mathcal{T}} &= \{\sigma \in r^{\mathcal{T}} \mid l \in A \Rightarrow l \notin dom(\sigma),\ l \in B \Rightarrow l \in dom(\sigma)\} \\ &= \{\sigma \in r^{\mathcal{T}} \mid \exists C \text{ with } B \subseteq C, A \cap C = \emptyset\ (l \in dom(\sigma) \Leftrightarrow l \in C)\} \\ &= \biguplus_{B \subseteq C, A \cap C = \emptyset} C^{\mathcal{T}}\end{aligned}$$

□

Lemma 3.2.10 If $\mathbf{r} \sqcap \mathbf{s} = \bot$ then $\mathbf{r}^{\mathcal{T}} \cap \mathbf{s}^{\mathcal{T}} = \emptyset$.

Proof: If either $\mathbf{r}$ or $\mathbf{s}$ is $\bot$ the proof is trivial. Otherwise, if either $\mathbf{r}$ or $\mathbf{s}$ is $\top$, then the premise cannot be fulfilled. If one of them is $\mathbf{f}$ or $\mathbf{p}_c$, the proof is simple. The interesting case is if $\mathbf{r}$ or $\mathbf{s}$ are record sorts. Then, according to Proposition 3.2.5, there are three subcases. We show the emptyness of the intersection for each subcase.

$\mathbf{r} = A,\ \mathbf{s} = B, A \neq B$

$$\begin{aligned}\mathbf{r}^{\mathcal{T}} \cap \mathbf{s}^{\mathcal{T}} &= \{\sigma \in r^{\mathcal{T}} \mid l \in dom(\sigma) \Leftrightarrow l \in A\} \\ &\quad \cap \{\sigma \in r^{\mathcal{T}} \mid l \in dom(\sigma) \Leftrightarrow l \in B\} \\ &= \emptyset\end{aligned}$$

$\mathbf{r} = (A,B),\ \mathbf{s} = (A',B'),\ A \cap B' \neq \emptyset \vee A' \cap B \neq \emptyset$

$A \cap B' \neq \emptyset$

$$\begin{aligned}\mathbf{r}^{\mathcal{T}} \cap \mathbf{s}^{\mathcal{T}} &\overset{(3.2.9)}{=} \biguplus_{B \subseteq C, A \cap C = \emptyset} C^{\mathcal{T}} \ \cap \biguplus_{B' \subseteq D, A' \cap D = \emptyset} D^{\mathcal{T}} \\ &= \emptyset\end{aligned}$$

$A' \cap B \neq \emptyset$ This case is symmetric.

$\mathbf{r} = A,\ \mathbf{s} = (B,C),\ A \cap B' \neq \emptyset \vee C \not\subseteq A$

$$\begin{aligned}\mathbf{r}^{\mathcal{T}} \cap \mathbf{s}^{\mathcal{T}} &\overset{(3.2.9)}{=} A^{\mathcal{T}} \ \cap \biguplus_{C \subseteq D, B \cap D = \emptyset} D^{\mathcal{T}} \\ &= \emptyset\end{aligned}$$

□

3.3 The Theory

In many "ordinary" type systems, two types are different if they are syntactically different; e.g. int $\to$ (int $\to$ int) is different from int $\to$ int. In our type system, syntactically different type terms may denote the same type: e.g. the types $\langle\langle\rangle \mid \overline{a} : \overline{\alpha}\rangle$ and $\langle\overline{a} : \overline{\alpha}\rangle$ are the same. Therefore, in Def. 3.3.1, there are equations on type expressions.

Definition 3.3.1 (RT-equations) Each of the following equation schemes stands for a countably infinite number of equations.

$$\langle\langle\overline{a} : \overline{\alpha}\rangle \mid \overline{b} : \overline{\beta}\rangle \doteq \langle\overline{a} : \overline{\alpha},\ \overline{b} : \overline{\beta}\rangle \tag{3.6}$$

$$\langle\langle\alpha \mid \overline{a} : \overline{\beta}\rangle \mid \overline{b} : \overline{\gamma}\rangle \doteq \langle\alpha \mid \overline{a} : \overline{\beta},\ \overline{b} : \overline{\gamma}\rangle \tag{3.7}$$

All type terms used here are well-sorted. As a consequence, in equation (3.6), we have $\{\overline{a}\} \cap \{\overline{b}\} = \emptyset$. In equation (3.7), α must have a sort excluding labels $\{\overline{a}, \overline{b}\}$ and, again, we have $\{\overline{a}\} \cap \{\overline{b}\} = \emptyset$.

Proposition 3.3.2 The rewrite system obtained by orienting the equations of Def. 3.3.1 to the right is sort-decreasing, terminating, and confluent.

Proof: One easily verifies that orientation of the equations to the right yields rewrite rules (i.e. $\sigma \rightsquigarrow \tau$ is an **RT**-equation $\sigma \doteq \tau$, σ is not a variable, and $\mathcal{V}\tau \subset \mathcal{V}\sigma$) and that the rules are sort-decreasing. In order to show termination, we use the size of type terms. The *size* of an **RT**-term is defined as follows.

$$\begin{aligned} |\alpha| &= 1 \\ |\sigma \to \tau| &= 1 + |\sigma| + |\tau| \\ \langle\overline{a} : \overline{\sigma}\rangle &= 1 + \overline{|\sigma|} \\ \langle\sigma \mid \overline{a} : \overline{\tau}\rangle &= 1 + |\sigma| + \overline{|\tau|} \end{aligned}$$

Theorem A.3.7 states that if a rewrite system is sort-decreasing, then it is locally confluent if and only if all critical pairs converge. Theorem A.3.4

states that if a relation is locally confluent and terminating then it is confluent. Thus, we can show confluence by the convergence of all critical pairs. We now list all overlaps with their corresponding substitutions and critical pairs:

Rules (3.7) and (3.6) overlap in the following way:
$(\langle\langle\alpha \mid \overline{a} : \overline{\beta}\rangle \mid \overline{b} : \overline{\gamma}\rangle \rightsquigarrow \langle\alpha \mid \overline{a} : \overline{\beta}, \overline{b} : \overline{\gamma}\rangle, 1, \langle\langle\overline{c} : \overline{\delta}\rangle \mid \overline{a} : \overline{\epsilon}\rangle \rightsquigarrow \langle\overline{c} : \overline{\delta}, \overline{a} : \overline{\epsilon}\rangle)$
$\theta = [\langle\overline{c} : \overline{\delta}\rangle/\alpha, \overline{\epsilon}/\overline{\beta}]$ $\qquad (\langle\langle\overline{c} : \overline{\delta}\rangle \mid \overline{a} : \overline{\epsilon}, \overline{b} : \overline{\gamma}\rangle, \langle\langle\overline{c} : \overline{\delta}, \overline{a} : \overline{\epsilon}\rangle \mid \overline{b} : \overline{\gamma}\rangle)$

Rule (3.7) overlaps with itself in the following way:
$(\langle\langle\alpha \mid \overline{a} : \overline{\beta}\rangle \mid \overline{b} : \overline{\gamma}\rangle \rightsquigarrow \langle\alpha \mid \overline{a} : \overline{\beta}, \overline{b} : \overline{\gamma}\rangle, 1, \langle\langle\delta \mid \overline{c} : \overline{\epsilon}\rangle \mid \overline{a} : \overline{\zeta}\rangle \rightsquigarrow \langle\delta \mid \overline{c} : \overline{\epsilon}, \overline{a} : \overline{\zeta}\rangle)$

$\theta = [\langle\delta \mid \overline{c} : \overline{\epsilon}\rangle/\alpha, \overline{\zeta}/\overline{\beta}]$ $\qquad (\langle\langle\delta \mid \overline{c} : \overline{\epsilon}\rangle \mid \overline{a} : \overline{\zeta}, \overline{b} : \overline{\gamma}\rangle, \langle\langle\delta \mid \overline{c} : \overline{\epsilon}, \overline{a} : \overline{\zeta}\rangle \mid \overline{b} : \overline{\gamma}\rangle)$

One easily verifies that the critical pairs converge. □

Before we can list the non equational axioms of our theory, we introduce definitions and notations concerning conjunctions of equations.

Definition 3.3.3 Let f_i $(1 \leq i \leq n)$ be any type term constructors of Def. 3.2.2. A *determinant* for pairwise distinct variables $\alpha_1, \ldots, \alpha_n$ is a constraint of the form $\overline{\alpha} \doteq \overline{f(\overline{\beta})}$,[6] where, for all $1 \leq i \leq n$, $\mathrm{sort}(\alpha_i) \leq \mathrm{sort}(f_i(\overline{\beta}))$. We call $\{\overline{\alpha}\}$ the set of variables determined by $\overline{\alpha} \doteq \overline{f(\overline{\beta})}$.

We use $\tilde{\forall}\phi$ to denote the *universal closure* of a formula ϕ. We define the quantifier $\exists!\alpha\phi$ ("there exists a unique α such that") as an abbreviation for

$$\exists\alpha\phi \wedge \forall\alpha, \beta(\phi \wedge [\beta/\alpha]\phi \Rightarrow \alpha \doteq \beta).$$

We extend this quantifier to sets of variables V accordingly: $\exists!V\phi$.

Definition 3.3.4 The *theory of* **RT**-*terms* RT is given by equations (3.6) and (3.7) and the following axiom schemes.[7]

$$\tilde{\forall}(\exists!\{\overline{\alpha}\}(\overline{\alpha} \doteq \overline{f(\overline{\beta})})) \qquad \left\{\text{if } \overline{\alpha} \doteq \overline{f(\overline{\beta})} \text{ is a determinant} \right. \tag{3.8}$$

[6] Expanding our notation, we obtain $\alpha_1 \doteq f_1(\beta_{1_1}, \ldots, \beta_{1_{m_1}}) \wedge \ldots \wedge \alpha_n \doteq f_n(\beta_{n_1}, \ldots, \beta_{n_{m_n}})$

[7] Side conditions for the applicability of rules are written on the right-hand side of an opening brace.

$$\alpha \doteq \beta \;\Rightarrow\; \bot \qquad \left\{ \begin{array}{l} \mathrm{sort}(\alpha) \sqcap \mathrm{sort}(\beta) \;=\; \bot \end{array} \right. \tag{3.9}$$

$$\begin{array}{l} \langle \alpha \mid \overline{a} : \overline{\beta},\ \overline{b} : \overline{\gamma} \rangle \doteq \langle \delta \mid \overline{a} : \overline{\epsilon},\ \overline{c} : \overline{\zeta} \rangle \\ \quad \Rightarrow\; \exists \eta (\overline{\beta} \doteq \overline{\epsilon} \wedge \alpha \doteq \langle \eta \mid \overline{c} : \overline{\zeta} \rangle \wedge \delta \doteq \langle \eta \mid \overline{b} : \overline{\gamma} \rangle) \\ \qquad \left\{ \begin{array}{ll} \{\overline{b}\} \cap \{\overline{c}\} & = \emptyset \\ \mathrm{sort}(\eta) & = (\{\overline{a}, \overline{b}, \overline{c}\}, \{\}) \end{array} \right. \end{array} \tag{3.10}$$

$$\alpha \to \beta \doteq \alpha' \to \beta' \;\Rightarrow\; \alpha \doteq \alpha' \;\wedge\; \beta \doteq \beta' \tag{3.11}$$

Axiom (3.8) claims the existence of unique solutions for determinants. Axiom (3.9) states that type terms with incompatible sorts cannot be equal. Axiom (3.11) expresses simply the componentwise equality on function type terms.

Axiom (3.10) deserves some more explanation. When two record types with row variables are confronted, we perform an operation called *padding* by Wand. We can imagine that the labels of the two record types are aligned and that the missing ones are padded in. The axiom states that padding is possible if record type terms have the proper row variables. The set of common labels is $\{\overline{a}\}$. The labels that differ are padded "crosswise" into the row variables α and δ. A new row variable η is introduced.

Definition 3.3.5 We define a binary relation on type terms as

$$\sigma \approx \tau \;:\Leftrightarrow\; \mathrm{RT} \models \sigma \doteq \tau$$

The above relation is a *congruence* on type terms.

Theorem 3.3.6
The feature tree structure $\mathcal{T}$ is a model of the theory RT.

Proof: We must show that every equation and every axiom of RT are satisfied. It is easily checked that the equations of Def. 3.3.1 are satisfied. We will check the axioms one by one.

(3.8) Assume arbitrary feature trees of correct sorts for universally quantified variables. Then the determinant represents a unique regular tree (cf. [22]). This is immediately clear for primitive type terms, function type terms, and record type terms without concatenation. For type terms of the form $\langle \beta \mid \overline{a} : \overline{\alpha} \rangle$ it is essential that $\{\overline{a}\} \neq \emptyset$. Otherwise, we could have an equation $\gamma \doteq \langle \gamma \mid \rangle$ in a determinant that would not determine γ uniquely.[8] Also note that the sort restrictions regarding left-hand sides and right-hand sides of equations are important.

(3.9) Terms of incompatible sorts are different in our model (Lemma 3.2.10).

(3.10) This holds because the label sets are disjoint and because a labeled tree is a function.

(3.11) This holds because labeled trees are functions.

□

Proposition 3.3.7 (no junk)
Every type can be represented by a determinant.

Proof: Regular trees can be represented by systems of equations. The restrictions of $\mathcal{T}$ w.r.t. the features at certain constructors (Def. 3.1.1) correspond to the restrictions in the sort system of type terms (Def. 3.2.2).
□

We call a conjunction of equations with variables on left-hand sides and terms on right-hand sides *constraint.*

Definition 3.3.8 A variable α is called *isolated* in a constraint ϕ if there is an equation $\alpha \doteq \sigma$ in ϕ and this is the only occurrence of α in ϕ.

Definition 3.3.9 A constraint $\overline{\alpha} \doteq \overline{\sigma}$ is called a *solved form* if, for all $1 \leq i \leq n$,

- the variables α_i are distinct,
- $\mathrm{sort}(\sigma_i) \leq \mathrm{sort}(\alpha_i)$, and

[8] See also the discussion on *contractiveness* by Amadio and Cardelli [2].

- if σ_i is a variable β, then α_i is isolated and $\beta \neq \alpha_i$.

Proposition 3.3.10 (unique solutions) If in a solved form $\overline{\alpha} \doteq \overline{\sigma}$ we put in unique solutions for all variables $\mathcal{V}\overline{\sigma}\backslash\{\overline{\alpha}\}$, the solved form denotes unique solutions for all variables in $\{\overline{\alpha}\}$ in every model of RT.

Proof: Solved forms are like determinants except that they may contain additional bindings of some variables to other variables. Therefore, they have unique solutions according to axiom (3.8). □

3.4 μ-Terms

We have seen in proposition 3.3.7 that we can denote arbitrary types by determinants, i.e. certain conjunctions of equations. Type terms alone, i.e. without the help of equations, can only denote finite types. In order to be able to denote arbitrary types by type terms alone, we introduce μ-terms. We will see that with μ-terms we have exactly the same expressivity as with determinants.

Definition 3.4.1 (μ-term)
Let f be any type term constructor (Def. 3.2.2) with $f : \overline{\mathbf{s}} \rightarrow \mathbf{s}$. Then

$$\mu\alpha.f(\bar{\cdot}) \;:\; \overline{\mathbf{s}} \rightarrow \mathbf{s} \quad \left\{ \mathbf{s} \;\leq\; \mathrm{sort}(\alpha) \right.$$

is a μ-term. μ-terms are interpreted as

$$[\![\mu\alpha.f(\overline{\sigma})]\!]_{\mathfrak{a}} \;=\; [\![\alpha]\!]_{\mathfrak{b}}$$

where $\mathfrak{b}$ is a unique α-update of $\mathfrak{a}$ such that $[\![\alpha]\!]_{\mathfrak{b}} = [\![f(\overline{\sigma})]\!]_{\mathfrak{b}}$.

The existence and the uniqueness of the α-update comes from the existence of unique solutions of equations in determinants and the "binding mechanism" of μ.

The following proposition shows that μ-terms do neither increase nor decrease the expressiveness of our language.

Proposition 3.4.2 (μ-term elimination)

$$\alpha \doteq \mu\beta.f(\overline{\gamma}) \quad \models\!\!\!|_{\mathrm{RT}} \quad \exists\beta(\alpha \doteq \beta \wedge \beta \doteq f(\overline{\gamma})) \tag{3.12}$$

Proof:

$$\begin{aligned}
&(\alpha \doteq \mu\beta.f(\overline{\gamma}))^{\mathcal{T}} \\
&= \{\mathfrak{a} \mid [\![\alpha]\!]_{\mathfrak{a}} = [\![\mu\beta.f(\overline{\gamma})]\!]_{\mathfrak{a}}\} \\
&= \{\mathfrak{a} \mid [\![\alpha]\!]_{\mathfrak{a}} = [\![\beta]\!]_{\mathfrak{b}} \text{ where } \mathfrak{b} \text{ is a } \beta\text{-update of } \mathfrak{a} \text{ and } [\![\beta]\!]_{\mathfrak{b}} = [\![f(\overline{\gamma})]\!]_{\mathfrak{b}}\} \\
&= \{\mathfrak{a} \mid \mathfrak{b} \text{ is a } \beta\text{-update of } \mathfrak{a} \text{ and } [\![\alpha]\!]_{\mathfrak{b}} = [\![f(\overline{\gamma})]\!]_{\mathfrak{b}} \wedge [\![\alpha]\!]_{\mathfrak{b}} = [\![\beta]\!]_{\mathfrak{b}}\} \\
&= \{\mathfrak{a} \mid \mathfrak{b} \text{ is a } \beta\text{-update of } \mathfrak{a} \text{ and } [\![\alpha]\!]_{\mathfrak{b}} = [\![\beta]\!]_{\mathfrak{b}} \wedge [\![\beta]\!]_{\mathfrak{b}} = [\![f(\overline{\gamma})]\!]_{\mathfrak{b}}\} \\
&= \{\mathfrak{a} \mid \mathfrak{b} \text{ is a } \beta\text{-update of } \mathfrak{a} \text{ and } \mathfrak{b} \in (\alpha \doteq \beta \wedge \beta \doteq f(\overline{\gamma}))^{\mathcal{T}}\} \\
&= (\exists\beta(\alpha \doteq \beta \wedge \beta \doteq f(\overline{\gamma})))^{\mathcal{T}}
\end{aligned}$$

□

Proposition 3.4.3 (unfolding) $\quad \beta \doteq [\mu\alpha.\sigma/\alpha]\sigma \quad \models\!\!\!|_{\mathrm{RT}} \quad \beta \doteq \mu\alpha.\sigma$

Proof: In predicate logic, we can unfold terms if bindings are not destroyed, i.e. we have the general law $\beta \doteq [\tau/\alpha]\sigma \quad \models\!\!\!| \quad \exists\alpha(\alpha \doteq \tau \wedge \beta \doteq \sigma)$. This law will be used in the following sequence.

$$\begin{aligned}
&\beta \doteq [\mu\alpha.\sigma/\alpha]\sigma \\
&\models\!\!\!|_{\mathrm{RT}} \quad \exists\alpha(\alpha \doteq \mu\alpha.\sigma \wedge \beta \doteq \sigma) \\
&\models\!\!\!|_{\mathrm{RT}} \quad \exists\alpha(\exists\alpha(\alpha \doteq \alpha \wedge \alpha \doteq \sigma) \wedge \beta \doteq \sigma) \\
&\models\!\!\!|_{\mathrm{RT}} \quad \exists\alpha(\alpha \doteq \sigma \wedge \beta \doteq \sigma) \\
&\models\!\!\!|_{\mathrm{RT}} \quad \exists\alpha(\beta \doteq \alpha \wedge \alpha \doteq \sigma) \\
&\overset{3.4.2}{\models\!\!\!|_{\mathrm{RT}}} \quad \beta \doteq \mu\alpha.\sigma
\end{aligned}$$

□

The following process will transform equations containing type terms of arbitrary depth with μ to formulae containing only flat type terms without μ. Flat terms have at most depth one.

Definition 3.4.4 (Flattening) Let f be any type term constructor (Def. 3.2.2). Apply the following function to every equation in the constraint.

$$\begin{aligned} \mathrm{flt}(\alpha \doteq \beta) &= \alpha \doteq \beta \\ \mathrm{flt}(\alpha \doteq f(\overline{\sigma})) &= \exists \overline{\beta}(\alpha \doteq f(\overline{\beta}) \wedge \overline{\mathrm{flt}(\beta \doteq \sigma)}) \\ \mathrm{flt}(\alpha \doteq \mu\beta.f(\overline{\tau})) &= \exists \beta(\alpha \doteq \beta \wedge \mathrm{flt}(\beta \doteq f(\overline{\tau}))) \end{aligned}$$

We are using the abbreviating notation $\overline{\mathrm{flt}(\beta \doteq \sigma)}$ to stand for the conjunction of the results. In the last case, the variables have to be renamed appropriately in order to avoid capture.

Lemma 3.4.5 Flattening leaves the set of solutions invariant.

Proof: We show this for one equation. For the solutions of equations, the same brackets are used as for the solutions of frames. We have to show $[\![\mathrm{flt}(\alpha \doteq \sigma)]\!] = [\![\alpha \doteq \sigma]\!]$. We proceed by structural induction on σ.

$\sigma \equiv \beta$

This case is trivial.

$\sigma \equiv p_c$

This case is trivial.

$\sigma \equiv \tau \to \tau'$

$$\begin{aligned} & [\![\mathrm{flt}(\alpha \doteq \tau \to \tau')]\!] \\ &= [\![\exists \beta, \gamma(\alpha \doteq \beta \to \gamma \wedge \mathrm{flt}(\beta \doteq \tau) \wedge \mathrm{flt}(\gamma \doteq \tau'))]\!] \\ &\overset{\text{ind. hyp.}}{=} [\![\exists \beta, \gamma(\alpha \doteq \beta \to \gamma \wedge \beta \doteq \tau \wedge \gamma \doteq \tau')]\!] \\ &= [\![\alpha \doteq \tau \to \tau']\!] \end{aligned}$$

$\sigma \equiv \langle \overline{a} : \overline{\tau} \rangle$

$$\begin{aligned} & [\![\mathrm{flt}(\alpha \doteq \langle \overline{a} : \overline{\tau} \rangle)]\!] \\ &= [\![\exists \overline{\beta}(\alpha \doteq \langle \overline{a} : \overline{\beta} \rangle \wedge \overline{\mathrm{flt}(\beta \doteq \tau)})]\!] \\ &\overset{\text{ind. hyp.}}{=} [\![\exists \overline{\beta}(\alpha \doteq \langle \overline{a} : \overline{\beta} \rangle \wedge \overline{\beta} \doteq \overline{\tau})]\!] \\ &= [\![\alpha \doteq \langle \overline{a} : \overline{\tau} \rangle]\!] \end{aligned}$$

$\sigma \equiv \langle \tau \mid \overline{a} : \overline{\tau} \rangle$

This case is similar to the previous one.

$\sigma \equiv \mu\beta.f(\overline{\tau})$

$$
\begin{array}{rl}
& [\![\mathrm{flt}(\alpha \doteq \mu\beta.f(\overline{\tau}))]\!] \\
= & [\![\exists\beta(\alpha \doteq \beta \wedge \mathrm{flt}(\beta \doteq f(\overline{\tau})))]\!] \\
\overset{\text{ind. hyp.}}{=} & [\![\exists\beta(\alpha \doteq \beta \wedge \beta \doteq f(\overline{\tau}))]\!] \\
\overset{(3.4.2)}{=} & [\![\alpha \doteq \mu\beta.f(\overline{\tau})]\!]
\end{array}
$$

□

Why have we defined the congruence on type terms using the semantics? Would there not be a simpler way? Let us look at type terms with μ. A simpler definition would use an unfolding rule like

$$[\mu\alpha.\sigma/\alpha]\sigma \;=\; \mu\alpha.\sigma \tag{3.13}$$

Yet, a simple definition by finite unfoldings seems impossible as the following example [2] indicates. The type terms

$$
\begin{array}{rcll}
\sigma & = & \mu\alpha.\mathrm{int} \to \alpha & \text{and} \\
\tau & = & \mu\beta.\mathrm{int} \to \mathrm{int} \to \beta &
\end{array}
$$

are equivalent. They both expand to $\mathrm{int} \to \mathrm{int} \to \mathrm{int} \to \ldots$ Their equivalence cannot be shown by assuming $\alpha = \beta$. It would remain to show $\mathrm{int} \to \alpha = \mathrm{int} \to \mathrm{int} \to \alpha$. Expanding the μ's is not successful either. By the unfolding rule (3.13) we get

$$
\begin{array}{rcl}
\sigma & = & \mu\alpha.\mathrm{int} \to \alpha \\
& = & \mathrm{int} \to \mu\alpha.\mathrm{int} \to \alpha \\
& = & \mathrm{int} \to \mathrm{int} \to \mu\alpha.\mathrm{int} \to \alpha \\
& = & \mathrm{int} \to \mathrm{int} \to \sigma \\
\tau & = & \mu\beta.\mathrm{int} \to \mathrm{int} \to \beta \\
& = & \mathrm{int} \to \mathrm{int} \to \mu\beta.\mathrm{int} \to \mathrm{int} \to \beta \\
& = & \mathrm{int} \to \mathrm{int} \to \tau\ .
\end{array}
$$

The original problem of whether $\sigma = \tau$ remains if we only have the unfolding rule. Also Cardone and Coppo [16] define the congruence semantically, i.e. using regular trees.

4. A Record Language

This chapter is organized as follows. Section 4.1 contains the syntax and semantics of our record calculus **R**. Section 4.2 contains the type inference rules and the soundness result. Section 4.3 contains the type inference algorithm which is split into various phases. The phases are verified separately and the principal type property is shown.

4.1 Expressions

The following language is a simple record calculus that is supposed to be incorporated into larger languages.

Definition 4.1.1 (Syntax of R) The language **R** is defined by the following abstract syntax. Variables are denoted by lower case letters x, y, and z. Record selection is denoted by a dot.

$$\begin{array}{lcll} e & ::= & x & \text{variable} \\ & | & \langle \overline{a} \mapsto \overline{e} \rangle & \text{record} \\ & | & e.a & \text{selection} \end{array}$$

Variables are introduced in order to make the type inference system more interesting and to prepare for extensions of the language. The order of labels in a record plays no rôle. A record possessing a b-field is denoted as $\langle \overline{a} \mapsto \overline{e}, b \mapsto e' \rangle$. From the distinctness of labels it follows that $b \notin \{a_1, \ldots, a_n\}$. Similarly, $\langle \overline{a} \mapsto \overline{e}, \overline{b} \mapsto \overline{e'} \rangle$ is a record possessing fields for the labels $a_1, \ldots, a_n, b_1, \ldots, b_m$ with $\{\overline{a}\} \cap \{\overline{b}\} = \emptyset$.

Definition 4.1.2 (R-rewrite rules) The following rewrite rule scheme stands for a countably infinite number of rules.

$$\langle \overline{a} \mapsto \overline{e}, b \mapsto e' \rangle.b \quad \longrightarrow \quad e' \tag{4.1}$$

The rewrite rule scheme applies to all label sets $\{\overline{a}\}$ and labels b provided that they fulfill the distinctness condition claimed above. If a record has a field for the selected label the entry is the result of the selection. Otherwise, the selection term remains and represents an error.

The necessary machinery concerning confluence and termination can be retrieved from section A.3.

Proposition 4.1.3 The rewrite system of Def. 4.1.2 is terminating and confluent.

Proof: The *size* of an **R**-term is defined as follows.[1]

$$\begin{aligned} |x| &= 1 \\ |e.a| &= |e| + 1 \\ |\langle \overline{a} \mapsto \overline{e} \rangle| &= 1 + \overline{|e|} \end{aligned}$$

The rewrite rule scheme strictly decreases the size of the term. This implies termination. Local confluence is shown by the convergence of all critical pairs. Confluence follows from local confluence and termination (Theorem A.3.4). Since we have no critical pairs we have confluence. □

As a consequence of this proposition, every term has a unique normal form (Proposition A.3.2).

4.2 Typings

The main task of a type inference system is to guarantee the absence of type errors for accepted programs. The only type error in our simple language **R** is the selection of a label in a record that does not have this label. Since the rewrite system is terminating the easiest way of detecting type errors would be to simply perform rewriting steps until we reach a normal form and then simply check unresolved selections on records.[2] The contents of this and the following sections only makes sense if we regard **R** as a subset of **RF**. Eventually λ-abstraction and function application will be added and the language will have the power of a Turing machine. The presence of variables in **R** makes the extension to **RF** easy. The type inference rules are formulated in the style of [25, 58]. A *sequent* is a triple $\Gamma \vdash e : \tau$. We read "term e has type τ in the type environment Γ".[3] Extending our

[1] Here, the abbreviating notation stands for a sum: $\overline{|e|} = \sum_{i=1}^{n} |e|$.

[2] Selections on variables are acceptable.

[3] In chapter 3, we have made a difference between types and type terms. In this and the following chapters we will blurr the distinction again. The distinction is made in [16] where there are different sets of type inference rules depending on whether one infers types or type terms. Our type inference rules actually deal with type terms.

$$\frac{}{\Gamma \vdash x : \tau} \left\{ \Gamma(x) = \tau \qquad \text{(VAR)} \right.$$

$$\frac{\Gamma \vdash \overline{e} : \overline{\sigma}}{\Gamma \vdash \langle \overline{a} \mapsto \overline{e} \rangle : \langle \overline{a} : \overline{\sigma} \rangle} \qquad \text{(REC)}$$

$$\frac{\Gamma \vdash e : \langle \sigma \mid a : \tau \rangle}{\Gamma \vdash e.a : \tau} \qquad \text{(SEL)}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \tau} \left\{ \sigma \approx \tau \qquad \text{(EQ)} \right.$$

Figure 4.1: Type inference rules for records

abbreviating notation, we write $\Gamma \vdash \overline{e} : \overline{\sigma}$ for $\Gamma \vdash e_1 : \sigma_1 \ldots \Gamma \vdash e_n : \sigma_n$. A *type environment* is a finite mapping from term variables to types. It is written as $[\overline{x} : \overline{\tau}]$. A type is retrieved from the type environment Γ by $\Gamma(x)$. The inference rules are shown in Fig. 4.1.[4] With $fv(e)$ we denote the *free variables* of the term e. $dom(f)$ denotes the *domain* of a function.

Definition 4.2.1 Let e be a term and Γ a type environment with $dom(\Gamma) = fv(e)$. If, for some σ, we have $\Gamma \vdash e : \sigma$ then we say that (Γ, σ) is a *typing* of e. We call a term e *well-typed* if it has a typing.

Definition 4.2.2 A *type substitution* maps type variables to types. Type substitutions are ranged over by θ and ψ. They are capture avoiding. We write the application of a type substitution θ to a type σ as the juxtaposition $\theta\sigma$. We write $\theta\Gamma$ for the application of θ to every component of the type environment Γ.

Definition 4.2.3 Let $dom(\Gamma) = dom(\Gamma')$. The typing (Γ, σ) is *more general than* the typing (Γ', σ') if and only if there exists a type substitu-

[4]Side conditions for the applicability of rules are written on the right-hand side of an opening brace.

tion θ with $dom(\theta) \subseteq fv(\sigma)$ such that $\theta\sigma \approx \sigma'$ and, for all $x \in dom(\Gamma)$, $\theta(\Gamma(x)) \approx \Gamma'(x)$.

Proposition 4.2.4 The relation 'more general than' is a preorder on typings.

Definition 4.2.5 (Principal typing) We call a typing (Γ, τ) of e *principal* if and only if it is more general than all other typings of e.

Lemma 4.2.6 (Rewriting preserves types) If (Γ, σ) is a typing of an **R**-term e and $e \rightsquigarrow e'$, then (Γ, σ) is a typing of e'.

Proof: It suffices to show that we can infer the same type for the right-hand side of the rewrite rule as for the left-hand side. For the left-hand side, we have the proof tree

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \overline{e} : \overline{\sigma} \quad \Gamma \vdash e' : \tau}{\Gamma \vdash \langle \overline{a} \mapsto \overline{e}, b \mapsto e' \rangle : \langle \overline{a} : \overline{\sigma}, b : \tau \rangle}\,(\text{REC})}{\Gamma \vdash \langle \overline{a} \mapsto \overline{e}, b \mapsto e' \rangle : \langle \langle \overline{a} : \overline{\sigma} \rangle \mid b : \tau \rangle}\,(\text{EQ})}{\Gamma \vdash \langle \overline{a} \mapsto \overline{e}, b \mapsto e' \rangle.b : \tau}\,(\text{SEL})$$

One of the premises of this proof tree is $\Gamma \vdash e' : \tau$. This would be the root of the proof tree for the right-hand side of the rewrite rule. □

Theorem 4.2.7 (Well-typed terms do not go wrong) Every well-typed ground term reduces to a normal form not containing any selections.

Proof: By Lemma 4.2.6, a normal form of a well-typed term is also well-typed. From the type inference rules for **R**, we see that every subterm of a well-typed term is also well-typed. Choose any innermost selection subterm, i.e. a term $e.b$ where e does not contain any selection. Then e must be of the form $\langle \overline{a} \mapsto \overline{e} \rangle$. There are two cases:

$b \in \{\overline{a}\}$

In this case, we can apply rule (4.1). This contradicts our assumption that the term is in normal form.

$b \notin \{\overline{a}\}$

In this case, we cannot infer a type for our subterm. This contradicts our assumption that the whole term is well-typed.

□

4.3 Type Reconstruction

The algorithm that finds a typing for a term consists of two phases. The first phase creates constraints as it decomposes the input term. The second phase solves the constraints.

The algorithm works on a structure called "frame". A frame consists of quantified variables, a conjunction of so-called *scopes*, and equational constraints.[5]

Definition 4.3.1 A *frame* is a triple written as $\exists\overline{\alpha}(\rho;\ \phi)$, where

- $\exists\overline{\alpha}$ is the existential quantification of type variables,
- ρ is a conjunction of scopes. A scope is a pair written as $\Gamma \vdash \omega$, where Γ is a type environment and ω is a conjunction of proof obligations. A proof obligation is a pair written as $e : \alpha$, where e is a term and α a type variable,
- ϕ is a conjunction of equational constraints $\alpha \doteq \tau$, where α is a type variable and τ a type.

Frames have semantics. Intuitively, a frame has a solution if all proof obligations can be fulfilled using the type environment.

Definition 4.3.2 Let $\exists\overline{\alpha}(\rho;\ \phi)$ be a frame where $\rho = \overline{\Gamma} \vdash \overline{\omega}$. A type substitution θ is a *solution* of this frame if and only if there is a type substitution ψ such that θ and ψ agree everywhere except possibly on $\overline{\alpha}$ such that for all $\Gamma \vdash \omega$ in ρ and for all $e : \alpha$ in ω

1. $\psi\Gamma \vdash e : \psi\alpha$,

[5] At this point, it suffices to work with one scope because there is no λ-abstraction, yet. Having several scopes will be useful when the language is extended.

2. for all $\alpha \doteq \tau \in \phi \;:\; \psi\alpha \approx \psi\tau$.

We denote the set of all solutions of the frame $\exists\overline{\alpha}(\rho;\ \phi)$ by $[\![\exists\overline{\alpha}(\rho;\ \phi)]\!]$.

The mapping ψ in the above definition expresses the usual interpretation [44] of the existential quantification $\exists\overline{\alpha}$ in the frame.

4.3.1 Constraint Extraction

The first phase creates constraints as it decomposes the input term. When the rules of the next definition are applied to a frame, its middle component is consumed while its rightmost component increases.

Definition 4.3.3

We define *frame simplification rules* for the language **R**.

$$\frac{\exists\overline{\alpha}((\Gamma \vdash x : \alpha \wedge \omega) \wedge \rho;\ \phi)}{\exists\overline{\alpha}(\omega \wedge \rho;\ \phi \wedge \alpha \doteq \tau)} \left\{ \Gamma(x) = \tau \right. \tag{4.2}$$

$$\frac{\exists\overline{\alpha}((\Gamma \vdash \langle \overline{a} \mapsto \overline{e} \rangle : \alpha \wedge \omega) \wedge \rho;\ \phi)}{\exists\overline{\alpha}, \overline{\beta}((\Gamma \vdash \overline{e} : \overline{\beta} \wedge \omega) \wedge \rho;\ \phi \wedge \alpha \doteq \langle \overline{a} : \overline{\beta} \rangle)} \left\{ \begin{array}{l} \overline{\beta} \text{ fresh} \\ \mathrm{sort}(\overline{\beta}) = \overline{\top} \end{array} \right. \tag{4.3}$$

$$\frac{\exists\overline{\alpha}((\Gamma \vdash e.a : \alpha \wedge \omega) \wedge \rho;\ \phi)}{\exists\overline{\alpha}, \beta, \gamma((\Gamma \vdash e : \beta \wedge \omega) \wedge \rho;\ \phi \wedge \beta \doteq \langle \gamma \mid a : \alpha \rangle)} \left\{ \begin{array}{l} \beta, \gamma \text{ fresh} \\ \mathrm{sort}(\beta) = \top \\ \mathrm{sort}(\gamma) = (\{a\}, \emptyset) \end{array} \right. \tag{4.4}$$

In rule (4.2), the type of the variable x is retrieved from the type environment. In rule (4.3), the record is split up as one would expect. For each record field, we introduce a new type variable of the maximal sort. In rule (4.4), the newly introduced variable γ has the correct sort that guarantees the disjointness of labels in record types. The newly introduced variable β has the top sort. Note that the frame simplification rules can be viewed as the upside-down version of rules (VAR), (REC), and (SEL) from Fig. 4.1.

Proposition 4.3.4 (Effectiveness)

The frame simplification rules transform every frame with terms of the language **R** to a frame of the form $\exists\overline{\alpha}(\top;\ \phi)$.

Proposition 4.3.5
The frame simplification rules leave the set of solutions invariant.

Proof: In this proof by case analysis on the frame simplification rules, we are using Def. 4.3.2 extensively. Each time we show that "θ is a solution of the frame on the top of the rule if and only if it is a solution of the frame on the bottom of the rule". Since we will be using a mapping ψ agreeing with θ except possibly on the existentially quantified variables, we will not mention this fact and only talk about ψ. We will also not mention that ψ must fulfill ω and ϕ because ω and ϕ appear in every frame simplification rule on the top and on the bottom. We abbreviate $\theta \in [\![\ldots \text{top frame} \ldots]\!]$ to "top" and $\theta \in [\![\ldots \text{bottom frame} \ldots]\!]$ to "bottom".

(4.2)

$$\begin{array}{lcl} \text{top} & \Leftrightarrow & \psi\Gamma \vdash x : \psi\alpha \\ & \overset{\text{(EQ)},\text{(VAR)}}{\Leftrightarrow} & \psi\alpha \approx \psi\tau,\ \Gamma(x) = \tau \\ & \Leftrightarrow & \text{bottom} \end{array}$$

(4.3)

$$\begin{array}{lcl} \text{top} & \Leftrightarrow & \psi\Gamma \vdash \langle \overline{a} \mapsto \overline{e} \rangle : \psi\alpha \\ & \overset{\text{(EQ)},\text{(REC)}}{\Leftrightarrow} & \psi\Gamma \vdash \overline{e} : \psi\overline{\beta},\quad \psi\alpha \approx \psi\langle \overline{a} : \overline{\beta} \rangle \\ & \Leftrightarrow & \text{bottom} \end{array}$$

(4.4)

$$\begin{array}{lcl} \text{top} & \Leftrightarrow & \psi\Gamma \vdash e.a : \psi\alpha \\ & \overset{\text{(EQ)},\text{(SEL)}}{\Leftrightarrow} & \psi\Gamma \vdash e : \psi\langle \gamma \mid a : \alpha \rangle \\ & \Leftrightarrow & \psi\Gamma \vdash e : \psi\beta \wedge \psi\beta \approx \psi\langle \gamma \mid a : \alpha \rangle \\ & \Leftrightarrow & \text{bottom} \end{array}$$

□

4.3.2 Constraint Resolution

The second phase operates on the third component of frames. It tries to resolve the constraints that were created by the first phase (Def. 4.3.3).

The equations in the third component of frames are of the form $\alpha \doteq \sigma$, where α is a type variable and σ is a type.

Before we start the resolution proper, we will "flatten" (Def. 3.4.4) the types on right-hand sides and replace all μ-types by sets of equations.[6] Now, that we have a conjunction of flat equations without μ-types, we can proceed to constraint resolution. We assume that all existential quantifiers have been moved to the top while avoiding name conflicts. Newly introduced variables in the next phase are assumed to be existentially quantified and the quantifiers, in turn, to be moved to the top. Types are always kept in normal form during the resolution process. The algorithm consists in applying rules (4.5) through (4.17) to the constraint ϕ. The order of application does not matter. For better readability, we write equations connected by $\wedge$ one above the and omit the $\wedge$-sign.

The first five rules eliminate repeated occurrences of the same variable on the left-hand side.

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \alpha\end{array}}{\phi} \tag{4.5}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \beta\end{array}}{\begin{array}{l}[\beta/\alpha]\phi \\ \alpha \doteq \beta\end{array}} \left\{ \begin{array}{l} \alpha \neq \beta \\ \mathrm{sort}(\alpha) \geq \mathrm{sort}(\beta) \\ \alpha \in \mathcal{V}\phi \end{array} \right. \tag{4.6}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \beta \to \gamma \\ \alpha \doteq \delta \to \epsilon\end{array}}{\begin{array}{l}\phi \\ \alpha \doteq \beta \to \gamma \\ \beta \doteq \delta \\ \gamma \doteq \epsilon\end{array}} \tag{4.7}$$

[6] μ-types or non-flat types are not generated in the constraint extraction phase but they can sneak in via the type environment Γ in rule (4.2).

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \sigma \\ \alpha \doteq \sigma\end{array}}{\begin{array}{l}\phi \\ \alpha \doteq \sigma\end{array}} \tag{4.8}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \sigma \\ \alpha \doteq \tau\end{array}}{\bot} \left\{ \begin{array}{l} \mathrm{sort}(\sigma) \sqcap \mathrm{sort}(\tau) = \bot \end{array} \right. \tag{4.9}$$

The side conditions of rule (4.6) are worth a comment. The sorts of the variables α and β may be equal. Termination (Theorem 4.3.6) is ensured by the last condition requiring that the variable α is not isolated.

The next five rules eliminate repeated occurrences of the same variable on the left-hand side when record types are involved.

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \langle \overline{a} : \overline{\beta} \rangle \\ \alpha \doteq \langle \overline{a} : \overline{\gamma} \rangle\end{array}}{\begin{array}{l}\phi \\ \alpha \doteq \langle \overline{a} : \overline{\beta} \rangle \\ \overline{\beta} \doteq \overline{\gamma}\end{array}} \tag{4.10}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \langle \overline{a} : \overline{\beta}, \overline{b} : \overline{\gamma} \rangle \\ \alpha \doteq \langle \delta \mid \overline{b} : \overline{\epsilon} \rangle\end{array}}{\begin{array}{l}\phi \\ \alpha \doteq \langle \overline{a} : \overline{\beta}, \overline{b} : \overline{\gamma} \rangle \\ \delta \doteq \langle \overline{a} : \overline{\beta} \rangle \\ \overline{\gamma} \doteq \overline{\epsilon}\end{array}} \tag{4.11}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \langle \beta \mid \overline{a} : \overline{\gamma} \rangle \\ \alpha \doteq \langle \delta \mid \overline{a} : \overline{\epsilon} \rangle\end{array}}{\begin{array}{l}\phi \\ \alpha \doteq \langle \beta \mid \overline{a} : \overline{\gamma} \rangle \\ \beta \doteq \delta \\ \overline{\gamma} \doteq \overline{\epsilon}\end{array}} \tag{4.12}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \langle \beta \mid \overline{a} : \overline{\gamma} \rangle \\ \alpha \doteq \langle \delta \mid \overline{a} : \overline{\epsilon}, \overline{b} : \overline{\xi} \rangle\end{array}}{\begin{array}{l}\phi \\ \alpha \doteq \langle \beta \mid \overline{a} : \overline{\gamma} \rangle \\ \beta \doteq \langle \delta \mid \overline{b} : \overline{\xi} \rangle \\ \overline{\gamma} \doteq \overline{\epsilon}\end{array}} \tag{4.13}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \langle \beta \mid \overline{a} : \overline{\gamma}, \overline{b} : \overline{\delta} \rangle \\ \alpha \doteq \langle \epsilon \mid \overline{a} : \overline{\zeta}, \overline{c} : \overline{\psi} \rangle \quad \left\{ \begin{array}{l} \{\overline{b}\} \neq \emptyset, \{\overline{c}\} \neq \emptyset \\ \{\overline{b}\} \cap \{\overline{c}\} = \emptyset \end{array} \right.\end{array}}{\begin{array}{l}\phi \\ \alpha \doteq \langle \iota \mid \overline{a} : \overline{\gamma}, \overline{b} : \overline{\delta}, \overline{c} : \overline{\psi} \rangle \\ \beta \doteq \langle \iota \mid \overline{c} : \overline{\psi} \rangle \\ \epsilon \doteq \langle \iota \mid \overline{b} : \overline{\delta} \rangle \\ \overline{\gamma} \doteq \overline{\zeta}\end{array}} \tag{4.14}$$

Rule (4.10) analyzes two closed record types. Closed record types can only be equal if their set of labels is equal. If the set of labels is not equal rule (4.9) can be applied. Rule (4.11) analyzes a closed and an open record type. The open record type must be padded in order to get the same label set. Rule (4.12) compares two open record types having the same set of explicit labels. Rules (4.13) and (4.14) compare two open record types having different sets of explicit labels. The missing labels are padded as needed.

The following three rules eliminate all equations $\alpha \doteq \sigma$ for which $\text{sort}(\alpha) \geq \text{sort}(\sigma)$ does not hold. adaptable$(\mathbf{s}, \sigma)$ is true if $\mathbf{s} \neq \bot$ and the variables in σ can be replaced by other variables such that $\text{sort}(\sigma) = \mathbf{s}$. adapt$(\mathbf{s}, \sigma)$ is the smallest substitution with sort maximal $\overline{\gamma}$ such that $\text{sort}([\overline{\gamma}/\overline{\beta}]\sigma) = \mathbf{s}$.

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \beta\end{array}}{\begin{array}{l}\phi \\ \beta \doteq \alpha\end{array}} \left\{ \text{sort}(\alpha) < \text{sort}(\beta) \right. \tag{4.15}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \sigma\end{array}}{\bot} \left\{ \begin{array}{l} \text{sort}(\alpha) \sqcap \text{sort}(\sigma) = \mathbf{s} \\ \text{not adaptable}(\mathbf{s}, \sigma) \end{array} \right. \tag{4.16}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \sigma\end{array}}{\begin{array}{l}\phi \\ \alpha \doteq [\overline{\gamma}/\overline{\beta}]\sigma \\ \overline{\beta} \doteq \overline{\gamma}\end{array}} \left\{ \begin{array}{l} \text{sort}(\alpha) \sqcap \text{sort}(\sigma) = \mathbf{s} \\ \text{adaptable}(\mathbf{s}, \sigma) \\ [\overline{\gamma}/\overline{\beta}] = \text{adapt}(\mathbf{s}, \sigma) \\ \mathbf{s} \neq \text{sort}(\alpha) \\ \mathbf{s} \neq \text{sort}(\sigma) \end{array} \right. \tag{4.17}$$

Rule (4.15) simply puts the variable with a smaller sort on the left-hand side. Two type terms may have incompatible sorts simply because some of the internal variables have sorts that are "too big". If two type terms have incompatible sorts even after adapting the variables, rule (4.16) is applicable. If the variables in the type terms can be replaced by variables with smaller sorts, rule (4.17) can be applied.

This concludes the constraint resolution rules. The ensemble of these rules represent an algorithm whose correctness we are going to prove now.

Theorem 4.3.6 (Termination)
There is no infinite chain of applications of rules (4.5) through (4.17).

Proof: In order to show the termination of the resolution step we define functions r_1 through r_5 that map the constraint ϕ into well-founded

domains. We denote multisets by $\{\!|\ \ldots\ |\!\}$. We define a size function on constraints as the lexicographical order on $(r_1, r_2, r_3, r_4, r_5)$, where

$$
\begin{aligned}
r_1 &= \{\!|\ \mathrm{sort}(\alpha) \mid \alpha \text{ occurs inside type term in } \phi\ |\!\} \\
r_2 &= \{\!|\ |\sigma| \mid \alpha \doteq \sigma \text{ is an occ. of an eq. in } \phi \text{ and } \sigma \text{ is no variable}\ |\!\} \\
r_3 &= \{\!|\ \mathrm{sort}(\alpha) \mid \beta \doteq \alpha \text{ is an occurrence of an equation in } \phi\ |\!\} \\
r_4 &= |\{\!|\ \alpha \doteq \sigma \text{ is an occurrence of an equation in } \phi\ |\!\}| \\
r_5 &= \{\alpha \mid \alpha \text{ occurs in } \phi, \alpha \text{ is not isolated}\}
\end{aligned}
$$

We will now show for each of the rules in question that their application to a constraint decreases its size. Since the order is lexicographic, it suffices to find one component with $>$ where all previous components are $\geq$.

rule	r_1	r_2	r_3	r_4	r_5
4.5	$=$	$=$	$>$		
4.6	$\geq$	$=$	$\geq$	$=$	$>$
4.7	$=$	$>$	$\geq$	$=$	$>$
4.8	$\geq$	$\geq$	$\geq$	$>$	
4.9	$\geq$	$\geq$	$\geq$	$>$	
4.10	$=$	$>$			
4.11	$>$				
4.12	$>$				
4.13	$>$				
4.14	$>$				
4.16	$\geq$	$\geq$	$\geq$	$>$	
4.15	$=$	$=$	$>$		
4.17	$=$	$=$	$>$		

Termination follows from the existence of a minimal element in the codomains of the functions and the absence of infinite chains for multiset replacements [26]. Sorts contain no infinite decreasing chains for fixed programs because the set of labels in a frame is finite. □

One may wonder why we bother to introduce a countably infinite number of labels while we must argue with finite label sets in frames for the termination of our resolution step. An infinite number of labels is needed for the

incrementality of the algorithm. We want to infer the type of a program once and for all, no matter where this program is used. It is exactly due to the unknown uses of a program that the labels cannot be reduced to a finite number.

Theorem 4.3.7 (Effectiveness) After the application of rules (4.5) through (4.17), we either have a solved form or failure.

Proof: We first show that variables on left-hand sides of equations are distinct. We assume that a variable occurs more than once on the left-hand side and show that this contradicts our assumption that the algorithm has already terminated.

$\alpha \doteq \beta \wedge \alpha \doteq \sigma$
: We must have $\alpha \neq \beta$ since, otherwise, rule (4.5) would be applicable. If $\text{sort}(\alpha) \geq \text{sort}(\beta)$ then rule (4.6) is applicable. If $\text{sort}(\alpha) < \text{sort}(\beta)$ then rule (4.15) is applicable. If a common subsort exists we can apply rule (4.17), otherwise, rule (4.16).

$\alpha \doteq \beta \rightarrow \gamma \wedge \alpha \doteq \sigma$
: If σ were a variable, this would have been treated in the previous case. Thus, σ can be a function type and rule (4.7) is applicable. Otherwise, rule (4.9) is applicable.

$\alpha \doteq \langle \overline{a} : \overline{\beta} \rangle \wedge \alpha \doteq \langle \overline{b} : \overline{\gamma} \rangle$
: If $\{\overline{a}\} = \{\overline{b}\}$ then rule (4.10) is applicable, otherwise, rule (4.9).

$\alpha \doteq \langle \overline{a} : \overline{\beta} \rangle \wedge \alpha \doteq \langle \delta \mid \overline{b} : \overline{\gamma} \rangle$
: If $\{\overline{a}\} \supseteq \{\overline{b}\}$ then rule (4.11) is applicable, otherwise, rule (4.9).

$\alpha \doteq \langle \beta \mid \overline{a} : \overline{\gamma} \rangle \wedge \alpha \doteq \langle \delta \mid \overline{b} : \overline{\epsilon} \rangle$
: If $\{\overline{a}\} = \{\overline{b}\}$ then rule (4.12) is applicable, otherwise, rule (4.13) or (4.14).

Next we show that the sorts of the right-hand sides of equations are subsorts of the respective left-hand sides. In the following cases, we always assume that there remains an equation $\alpha \doteq \sigma$ and that $\text{sort}(\alpha) \geq \text{sort}(\sigma)$ does not hold. The results are contained in the following table. Above, we list the sort relation of α and σ. On the left, we list the possible forms of σ. The table entries consist of the applicable rules or a comment that a combination is impossible.

	$\mathrm{sort}(\alpha) < \mathrm{sort}(\sigma)$	$\mathrm{sort}(\alpha) \sqcap \mathrm{sort}(\sigma) = \mathbf{s}$ $\mathbf{s} \neq \bot$ $\mathbf{s} < \mathrm{sort}(\alpha)$ $\mathbf{s} < \mathrm{sort}(\sigma)$	$\mathrm{sort}(\alpha)$ $\sqcap$ $\mathrm{sort}(\sigma)$ $= \bot$
β	(4.15)	(4.17)	(4.16)
$\beta \to \gamma$	impossible	impossible	(4.16)
$\langle \overline{a} : \overline{\beta} \rangle$	impossible	impossible	(4.16)
$\langle \beta \mid \overline{b} : \overline{\gamma} \rangle$	(4.16 or 4.17)	(4.17)	(4.16)

It remains to show that, in equations $\alpha \doteq \beta$, the variables are distinct and α is isolated. If the variables are not distinct, rule (4.5) is applicable. Otherwise, rule (4.6) is applicable. □

Lemma 4.3.8 The application of rules (4.5) through (4.17) leaves the set of solutions invariant.

Proof: We show invariance for any model $\mathcal{A}$ of the theory RT by case analysis over the rules. Since the proofs for rules (4.5), (4.6), (4.8), (4.15), and (4.17) are trivial we concentrate on the remaining ones.

(4.7)

$$
\begin{aligned}
&(\phi \wedge \alpha \doteq \beta \to \gamma \wedge \alpha \doteq \delta \to \epsilon)^{\mathcal{A}} \\
&\quad = \quad \{a \mid a \in (\phi)^{\mathcal{A}} \wedge [\![\alpha]\!]_a = [\![\beta \to \gamma]\!]_a \wedge [\![\alpha]\!]_a = [\![\delta \to \epsilon]\!]_a\} \\
&\quad \overset{(3.11)}{=} \quad \{a \mid a \in (\phi)^{\mathcal{A}} \wedge [\![\alpha]\!]_a = [\![\beta \to \gamma]\!]_a \wedge [\![\beta]\!]_a = [\![\delta]\!]_a \wedge [\![\gamma]\!]_a = [\![\epsilon]\!]_a\} \\
&\quad = \quad (\phi \wedge \alpha \doteq \beta \to \gamma \wedge \beta \doteq \delta \wedge \gamma \doteq \epsilon)^{\mathcal{A}}
\end{aligned}
$$

(4.9)

$$
\begin{aligned}
&(\phi \wedge \alpha \doteq \sigma \wedge \alpha \doteq \tau)^{\mathcal{A}} \\
&\quad \subseteq \quad \{a \mid [\![\alpha]\!]_a = [\![\sigma]\!]_a \wedge [\![\alpha]\!]_a = [\![\tau]\!]_a\} \\
&\quad = \quad \{a \mid [\![\alpha]\!]_a = [\![\sigma]\!]_a \wedge [\![\sigma]\!]_a = [\![\tau]\!]_a\} \\
&\quad \subseteq \quad \{a \mid [\![\sigma]\!]_a = [\![\tau]\!]_a\} \\
&\quad \overset{(3.9)}{=} \quad (\bot)^{\mathcal{A}}
\end{aligned}
$$

(4.16)

This case is similar to the previous one.

(4.10)

$$\begin{aligned}
&(\phi \wedge \alpha \doteq \langle \overline{a} : \overline{\beta} \rangle \wedge \alpha \doteq \langle \overline{a} : \overline{\gamma} \rangle)^{\mathcal{A}} \\
&= \quad \{a \mid a \in (\phi)^{\mathcal{A}} \wedge [\![\alpha]\!]_a = [\![\langle \overline{a} : \overline{\beta} \rangle]\!]_a \wedge [\![\langle \overline{a} : \overline{\beta} \rangle]\!]_a = [\![\langle \overline{a} : \overline{\gamma} \rangle]\!]_a\} \\
&\overset{(3.10)}{=} \quad \{a \mid a \in (\phi)^{\mathcal{A}} \wedge [\![\alpha]\!]_a = [\![\langle \overline{a} : \overline{\beta} \rangle]\!]_a \wedge [\![\overline{\beta}]\!]_a = [\![\overline{\gamma}]\!]_a\} \\
&= \quad (\phi \wedge \alpha \doteq \langle \overline{a} : \overline{\beta} \rangle \wedge \overline{\beta} \doteq \overline{\gamma})^{\mathcal{A}}
\end{aligned}$$

(4.11), (4.12), (4.13),(4.14)

These cases are similar to the previous one.

□

4.3.3 The Rebuilding Phase

So far, we have either obtained a solved form or failure. If we have a solved form, the type of our original expression and the types of free variables in the environment are now spread over a set of equations. We want to have each of the interesting types in one equation or, in other words, we want one type (term) to express everything without referring to other equations. For this purpose we have to substitute the variables occurring on right-hand sides by their definitions, i.e. we rebuild the types. In the case of ordinary types, this is trivial. With recursive types, we must be careful not to fall into the trap of an infinite loop. Thus, we introduce μ-terms if we detect recursion. The *rebuilding* phase consists of the application of the following three rules. Let f be any n-ary type constructor. $fv(\phi)$ denotes the variables in ϕ that are not bound by μ.

$$\begin{array}{l} \phi \\ \alpha \doteq \sigma \\ \hline [\sigma/\alpha]\phi \\ \alpha \doteq \sigma \end{array} \quad \begin{cases} \alpha \notin fv(\sigma) \\ \alpha \in fv(\phi) \end{cases} \tag{4.18}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq f(\overline{\sigma})\end{array}}{\begin{array}{l}[\mu\alpha.f(\overline{\sigma})/\alpha]\phi \\ \alpha \doteq \mu\alpha.f(\overline{\sigma})\end{array}} \left\{ \alpha \in fv(\overline{\sigma}) \right. \tag{4.19}$$

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \alpha\end{array}}{\phi} \tag{4.20}$$

Note that rule (4.20) is the same as rule (4.5) of the resolution phase.

Proposition 4.3.9 The rebuilding phase terminates.

Proof: Rule (4.18) strictly decreases the set of non-isolated variables in ϕ (cf. r_5 in the proof of Theorem 4.3.6). Rules (4.19) and (4.20) do not increase this set and they strictly decrease the set of variables occurring freely simultaneously on the left-hand side and the right-hand side of one and the same equation. □

Lemma 4.3.10
The rebuilding phase leaves the set of solutions invariant.

Proof: The invariance of rules (4.18) and (4.20) is clear. We consider rule (4.19).

$$\begin{array}{lll}
[\![[\mu\alpha.f(\overline{\sigma})/\alpha]\phi \wedge \alpha \doteq \mu\alpha.f(\overline{\sigma})]\!] & & \\
\quad = & [\![\phi \wedge \alpha \doteq \mu\alpha.f(\overline{\sigma})]\!] & \\
\quad \overset{(3.4.2)}{=} & [\![\phi \wedge \exists\beta(\alpha \doteq \beta \wedge \beta \doteq f(\overline{\sigma}))]\!] & \\
\quad = & [\![\phi \wedge \alpha \doteq f(\overline{\sigma}))]\!] &
\end{array}$$

□

Proposition 4.3.11 After the rebuilding phase, free[7] variables on right-hand sides do not occur on left-hand sides in the constraint.

[7]This means: not bound by μ.

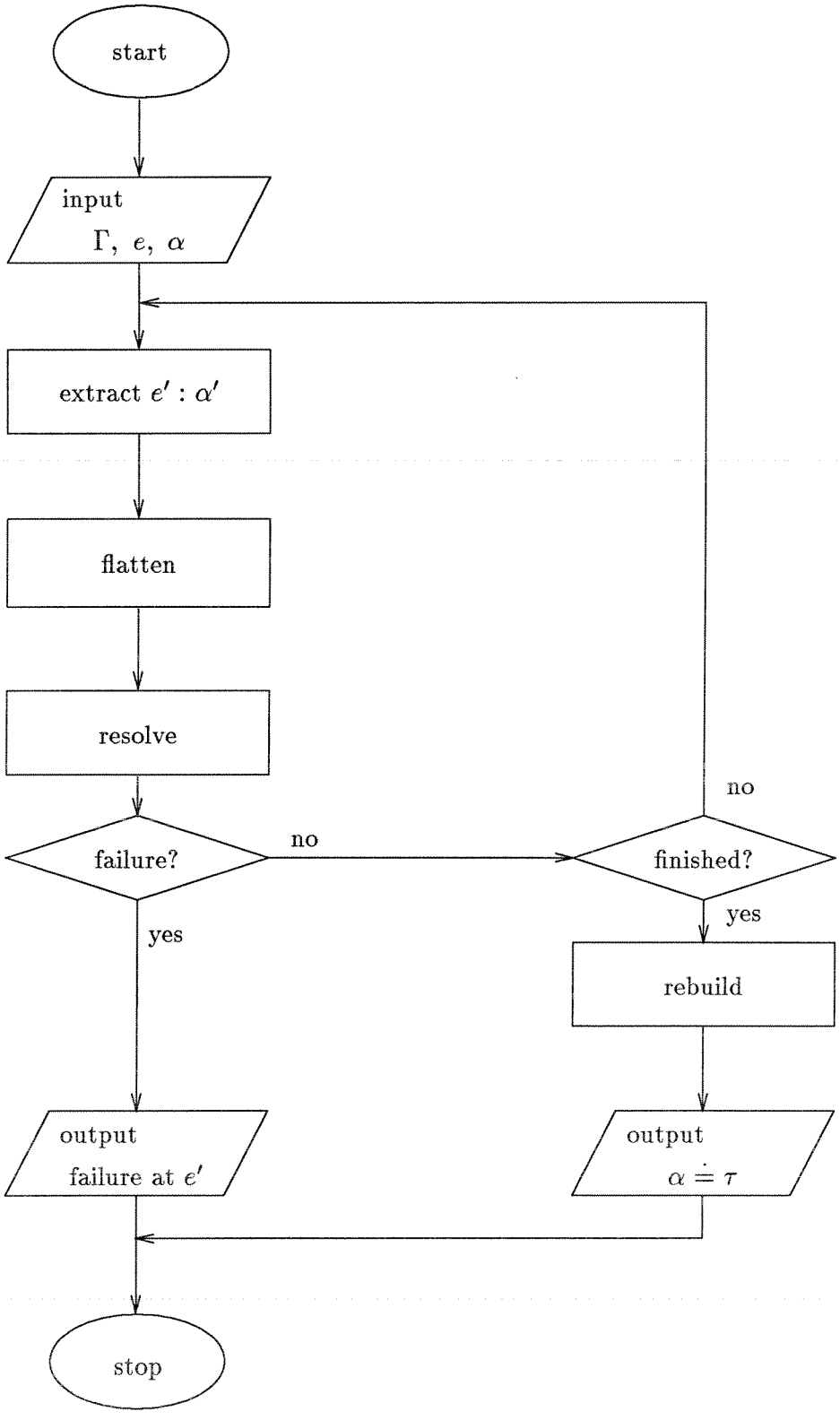


Figure 4.2: The practical algorithm (Homage to flow charts)

Proof: We lead a proof by contradiction. Each case will show an applicable rule contradicting the assumption that the rebuilding phase has already terminated. Assume that there is an equation $\alpha \doteq C(\beta)$ in the constraint, where C is some context not binding β, and β occurs on the left-hand side in the constraint.

$\alpha = \beta$
: If the context C is empty, rule (4.20) would be applicable. For any non-empty context, rule (4.19) is applicable.

$\alpha \neq \beta$
: Now, there must be another equation $\beta \doteq \sigma$ in the constraint. We have two cases depending on the form of σ:

  $\sigma \equiv \gamma$
  : If $\gamma = \beta$, then rule (4.20) is applicable. Otherwise, rule (4.18) is applicable.

  $\sigma \equiv f(\overline{\tau})$
  : If $\beta \in fv(\overline{\tau})$, then rule (4.19) is applicable. Otherwise, rule (4.18) is applicable.

□

As a consequence of this proposition, the solutions for a variable are completely characterized by one equation. However, if we are looking for several variables at the same time, as it is the case for the "initial" variables in the type environment, there may be common free variables in the solutions for different variables. An example for this can be found in section 5.3.

The simplest description of the whole type inference algorithm is just the sequential application of all the phases that we have described. After the input of a type environment Γ, the expression e, and a type variable α, the algorithm extracts all the constraints, flattens them, and tries to resolve them. If it succeeds, it outputs the binding of the variable α. Otherwise, it outputs "failure". If it succeeds, the information is sufficient. However, if it fails there is no way of telling where it failed. Therefore, we propose an interleaving algorithm as it is depicted in Fig. 4.2. This algorithm extracts the constraints of only *one* proof obligation. Then, it proceeds to the resolution phase. A failure can be detected as early as possible, and the

algorithm can report where the failure happened. In Fig. 4.2, it says that it happened while extracting the constraint for expression e'. If there is a fixed strategy of picking the proof obligations to extract, e.g. left to right, one can see until which point the program could be type checked.

Theorem 4.3.12 (Principal types) Let $fv(e) = \{\overline{x}\}$. If we start the algorithm with the frame $\exists([\overline{x} : \overline{\beta}] \vdash e : \alpha;\ \top)$, where $\beta_1, \ldots, \beta_n$, and α are distinct, it computes a principal typing if e has a typing, or stops with failure.

Proof: We consider the non-interleaving version of the algorithm because the interleaving version can be treated analagously. The failure case is not considered because it is much simpler. Let us recall what we know about our algorithm.

- The frame simplification rules extract all proof obligations until we have a frame of the form $\exists\overline{\alpha}(\top;\ \phi)$ (proposition 4.3.4).
- The flattening phase eliminates all μs and non-flat type terms in ϕ.
- The resolution phase leads to a solved form (theorem 4.3.7).
- The rebuilding phase gives us an equation $\alpha \doteq \sigma$ where α is the type variable of the input frame and σ is a type term, possibly containing μ, that completely characterizes the type of e (proposition 4.3.11). Similarly it contains equations $\beta \doteq \tau$ for all concerned variables $\beta \in \{\overline{\beta}\}$.

Therefore, we can say that the algorithm computes a typing (Γ, σ) supposing that the concerned variables in $[\overline{x} : \overline{\beta}]$ have been replaced by their solutions τ.

All phases of the algorithm leave the set of solutions invariant (lemmata 3.4.5, 4.3.5, 4.3.8, and 4.3.10). The type terms σ and $\overline{\tau}$ are functions from assignments (mappings from free type variables to types) to types (axiom (3.8) and proposition 3.4.2). Propositions 3.3.7 and 3.4.2 guarantee that assignments and substitutions are interchangeable because assignments cannot introduce types that are not expressible by substitutions.

In the sequel, we consider σ and omit $\overline{\tau}$ because it is analogous. We must show that the typing found by our algorithm is a principal typing. Restricted to our type σ, this means that for any other typing of e with a type σ', there exists a substitution θ such that $\theta\sigma \approx \sigma'$. We show the existence of θ constructively.

Let α be a new type variable. Assume that the variables in σ and σ' are disjoint. We build the combined system of equations

$$\mathrm{flt}(\alpha \doteq \sigma) \wedge \mathrm{flt}(\alpha \doteq \sigma'),$$

apply constraint resolution to it and obtain a solved form $\overline{\alpha''} \doteq \overline{\sigma''}$. Since the solutions of σ' are a subset of those of σ and since the flattening and resolution phases are invariant, the solved form $\overline{\alpha''} \doteq \overline{\sigma''}$ has the same solutions as σ'. On the left-hand side of $\overline{\alpha''} \doteq \overline{\sigma''}$, there is a subset of the free variables of σ that is now bound to some type terms. This is the substitution θ we have been looking for. □

5. An Applicative Language

A look at Fig. 2.1 (page 15) reveals that, so far, only the language **R** with its type inference has been presented. What remains to be done is the extension to the language **RFI**. This extension is divided into three steps:

- sections 5.1-5.3 contain the extension to **RF**,
- section 5.4 adds the 'let'-construct, and
- chapter 6 contains the extension to **RFI**.

The type inference for **RF** is a combination of the type inference for **R** (chapter 4) and the polymorphic type inference for the λ-calculus with 'let' by Hindley [43] and Milner [63] augmented by recursive types [16]. As in the previous sections, the language will contain no type declarations. In the untyped λ-calculus, the fixed-point operator can be defined [4] and therefore, at the end of this section, we will have a full functional language with records.

5.1 Expressions

We extend the record language **R** by λ-abstraction and function application. In section 5.4, we will add let-declarations to **RF**. They can be regarded as syntactic sugar on the semantic level. However, on the level of type inference they play a crucial rôle.

Definition 5.1.1 (Syntax of RF)
The language **RF** is defined by the following abstract syntax.

e	::=	x	variable
	|	$\langle \overline{a} \mapsto \overline{e} \rangle$	record
	|	$e.a$	selection
	|	$\lambda x.e$	abstraction
	|	$e\ e$	application

There are variables, records, and λ-terms. The semantics of **R** is defined in Def. 4.1.2 (page 41) by an algebraic rewriting system. An *applicative* term rewriting system [55] is an algebraic rewriting systems containing a special binary operator called *application*. This operator is expressed by

juxtaposition and is left-associative. The following reduction relation can be regarded as the combination of an algebraic rewrite system and rules for the λ-calculus.

Definition 5.1.2 (RF reduction) The applicative rewriting system, together with β-reduction, defines a one-step *reduction relation* . $\rightsquigarrow$. on **RF**; it is the least relation satisfying the following rules.

$$\frac{}{(\lambda x.e_1)\ e_2 \ \rightsquigarrow\ [e_2/x]e_1} \tag{5.1}$$

$$\frac{}{\langle \overline{a} \mapsto \overline{e}, b \mapsto e \rangle.b \ \rightsquigarrow\ e} \tag{5.2}$$

$$\frac{e_1 \ \rightsquigarrow\ e_1'}{e_1\ e_2 \ \rightsquigarrow\ e_1'\ e_2} \tag{5.3}$$

$$\frac{e_2 \ \rightsquigarrow\ e_2'}{e_1\ e_2 \ \rightsquigarrow\ e_1\ e_2'} \tag{5.4}$$

$$\frac{e \ \rightsquigarrow\ e'}{\lambda x.e \ \rightsquigarrow\ \lambda x.e'} \tag{5.5}$$

An important question, which might be easily overlooked, is the confluence of the thus obtained reduction system. The confluence of the λ-calculus is well known [4] and the rewriting system of the record language is confluent (Proposition 4.1.3). One might be led to believe that the combination of two confluent systems is also confluent. This is not always the case but a theorem by Müller [69] helps us here.

Proposition 5.1.3 **RF** reduction is confluent.

$$\frac{\Gamma \cdot [x : \sigma] \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \to \tau} \quad \text{(ABS)}$$

$$\frac{\Gamma \vdash e : \sigma \to \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash e\ e' : \tau} \quad \text{(APP)}$$

Figure 5.1: Additional type inference rules for functions

Proof: Looking at the algebraic rewriting rules of Def. 4.1.2, we see that they are linear and not variable-applying. Thus, the confluence of the entire reduction system is a consequence of the confluence theorem by Müller [69]. □

5.2 Typings

The additional inference rules are shown in Fig. 5.1. In $\Gamma \cdot [x : \tau]$, the type environment Γ is either extended by an entry for x or an existing entry is overwritten.

Proposition 5.2.1 (Rewriting preserves types) If (Γ, σ) is a typing of an **RF**-term e and $e \rightsquigarrow e'$ then (Γ, σ) is a typing of e'.

Proof: Since we have shown a similar Proposition (4.2.6) for the algebraic rewrite rules, we restrict ourselves to β-reduction. The proof tree for the left-hand side is:

$$\frac{\dfrac{\Gamma \cdot [x : \sigma] \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \to \tau}\ \text{(ABS)} \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash (\lambda x.e)\ e' : \tau}\ \text{(APP)}$$

The right-hand side is $[e'/x]e$. On the left-hand side, we also have a proof for $\Gamma \cdot [x : \sigma] \vdash e : \tau$. In this proof, we can infer the type σ for x using

rule (VAR). From this proof, we can obtain a proof for $\Gamma \vdash [e'/x]e : \tau$ by replacing all occurrences of rule (VAR) for x by occurrences of the proof for $\Gamma \vdash e' : \sigma$. □

Terms with "unevaluated" applications or selections represent error elements. The next theorem states that well-typed terms can be reduced to terms that do not represent error elements.

Definition 5.2.2 (Error element) In the language **RF**, we call a term of one of the following forms *error element.*

$\langle \overline{a} \mapsto \overline{e} \rangle.b \quad \left\{ \; b \notin \{\overline{a}\} \right.$
$\langle \overline{a} \mapsto \overline{e} \rangle \; e'$
$(\lambda x.e).a$

Theorem 5.2.3 (Well-typed terms do not go wrong) If a well-typed **RF**-term reduces to a normal form, this normal form contains no error elements.

Proof: Error elements have no typing. Since every subterm of a well-typed term must have a typing, a well-typed term cannot contain error elements. Since rewriting preserves types (Proposition 5.2.1), the normal form of a well-typed term is also well-typed, and cannot contain error elements. □

As an example, we will infer a type for the term $\lambda x.(x.a)$ using the inference rules of Fig. 5.1 and Fig. 4.1.

$$\dfrac{\dfrac{\dfrac{}{[x : \langle \alpha \mid a : \beta \rangle] \vdash x : \langle \alpha \mid a : \beta \rangle}\,(\text{VAR})}{[x : \langle \alpha \mid a : \beta \rangle] \vdash x.a : \beta}\,(\text{SEL})}{[\,] \vdash \lambda x.(x.a) \; : \; \langle \alpha \mid a : \beta \rangle \to \beta}\,(\text{ABS})$$

The sort of α is $(\{a\}, \emptyset)$ and the sort of β is $\top$.

5.3 Type Reconstruction

We extend the algorithm of section 4.3, i.e. the constraint creation phase of section 4.3.1.

Definition 5.3.1 We extend Def. 4.3.3 by the following *frame simplification rules* for the language **RF**.

$$\frac{\exists\overline{\alpha}((\Gamma \vdash \lambda x.e : \alpha \wedge \omega) \wedge \rho;\ \phi)}{\exists\overline{\alpha},\beta,\gamma((\Gamma \cdot [x:\beta] \vdash e : \gamma) \wedge (\Gamma \vdash \omega) \wedge \rho;\ \phi \wedge \alpha \doteq \beta \to \gamma)} \left\{ \begin{array}{l} \mathrm{sort}(\beta) = \top \\ \mathrm{sort}(\gamma) = \top \\ \beta,\gamma \text{ fresh} \end{array} \right. \quad (5.6)$$

$$\frac{\exists\overline{\alpha}((\Gamma \vdash e\ e' : \alpha \wedge \omega) \wedge \rho;\ \phi)}{\exists\overline{\alpha},\beta,\gamma((\Gamma \vdash e : \gamma \wedge e' : \beta \wedge \omega) \wedge \rho;\ \phi \wedge \gamma \doteq \beta \to \alpha)} \left\{ \begin{array}{l} \mathrm{sort}(\beta) = \top \\ \mathrm{sort}(\gamma) = \mathbf{f} \\ \beta,\gamma \text{ fresh} \end{array} \right. \quad (5.7)$$

Lemma 5.3.2 The frame simplification rules leave the set of solutions invariant.

Proof: We show the implications from 'top' to 'bottom' and vice versa for the new rules in the same way as in the proof of Lemma 4.3.5.

(5.6)

$$\begin{array}{rcl} \text{top} & \Rightarrow & \eta\Gamma \vdash \lambda x.e : \eta\alpha \\ & \overset{\text{(ABS)}}{\Rightarrow} & \eta\Gamma \cdot [x:\beta] \vdash e : \eta\gamma,\ \eta\alpha \approx \eta(\beta \to \gamma) \\ & \Rightarrow & \text{bottom} \\ \text{bottom} & \Rightarrow & \eta\Gamma \cdot [x:\beta] \vdash e : \eta\gamma,\ \eta\alpha \approx \eta(\beta \to \gamma) \\ & \overset{\text{(ABS)}}{\Rightarrow} & \eta\Gamma \vdash \lambda x.e : \eta(\beta \to \gamma),\ \eta\alpha \approx \eta(\beta \to \gamma) \\ & \overset{\text{(EQ)}}{\Rightarrow} & \text{top} \end{array}$$

(5.7)

$$\begin{array}{rcl} \text{top} & \Rightarrow & \eta\Gamma \vdash e\ e' : \eta\alpha \\ & \overset{\text{(APP)}}{\Rightarrow} & \eta\Gamma \vdash e : \eta(\beta \to \alpha),\ \eta\Gamma \vdash e' : \eta\beta \\ & \Rightarrow & \text{bottom} \\ \text{bottom} & \Rightarrow & \eta\Gamma \vdash e : \eta\gamma,\ \eta\Gamma \vdash e' : \eta\beta,\ \eta\gamma = \eta(\beta \to \alpha) \\ & \overset{\text{(APP)}}{\Rightarrow} & \text{top} \end{array}$$

The other cases have been proved in the proof of Lemma 4.3.5. □

The constraint resolution and the rebuilding phase remain unchanged. All properties, including the principal type property remain. The proofs are analogous to those in chapter 4. Differing from **R**, we can have non-terminating computations in **RF**. Even programs that are accepted by the type checker are not *strongly normalizing* because we admit recursive types. An example of a typing for a function that may cause non-termination can be found below.

It is now time for some examples. The first one will yield an open and recursive record type. The term is $(x.a)\ x$. Since this term is not closed, we must put $x : \beta$ into the initial type environment. We show the evolution of the frame omitting the quantified variables. The simplification rules for the transitions are written on the left-hand side.

	$[x : \beta] \vdash (x.a)\ x : \alpha$	
(5.7)	$[x : \beta] \vdash x.a : \gamma_1 \wedge x : \beta_1$	$\gamma_1 \doteq \beta_1 \to \alpha$
(4.4)	$[x : \beta] \vdash x : \beta_2 \wedge x : \beta_1$	$\gamma_1 \doteq \beta_1 \to \alpha \wedge \beta_2 \doteq \langle \gamma_2 \mid a : \gamma_1 \rangle$
(4.2)	$[x : \beta] \vdash x : \beta_1$	$\gamma_1 \doteq \beta_1 \to \alpha \wedge \beta_2 \doteq \langle \gamma_2 \mid a : \gamma_1 \rangle$ $\wedge \beta_2 \doteq \beta$
(4.2)	$\top$	$\gamma_1 \doteq \beta_1 \to \alpha \wedge \beta_2 \doteq \langle \gamma_2 \mid a : \gamma_1 \rangle$ $\wedge \beta_2 \doteq \beta \wedge \beta_1 \doteq \beta$

The variables α and β, the interesting ones, are not quantified. All other variables are existentially quantified, when they are introduced. The flattening phase is the identity in this case. After the constraint resolution phase we obtain:

$$\begin{aligned} \gamma_1 &\doteq \beta \to \alpha \\ \beta &\doteq \langle \gamma_2 \mid a : \gamma_1 \rangle \end{aligned}$$

We have omitted the isolated variables. The rebuilding phase yields

$$\beta \doteq \mu\beta.\langle \gamma_2 \mid a : \beta \to \alpha \rangle$$

This is the type for x. The type of the whole expression α is unconstrained. However, it must be the same as the codomain type above.

The next example yields a recursive type that is not in its minimal form. The term is $\lambda x.x\ x$. Since this term is closed, we start the algorithm with an empty type environment. During the constraint extraction phase, $x : \beta$ will be inserted into the type environment.

	$[] \vdash \lambda x.x\ x : \alpha$	
(5.6)	$[x : \beta] \vdash x\ x : \gamma_1$	$\alpha \doteq \beta_1 \to \gamma_1$
(5.7)	$[x : \beta] \vdash x : \gamma_2 \wedge x : \beta_2$	$\alpha \doteq \beta_1 \to \gamma_1 \wedge \gamma_2 \doteq \beta_2 \to \gamma_1$
(4.2)	$[x : \beta] \vdash x : \beta_2$	$\alpha \doteq \beta_1 \to \gamma_1 \wedge \gamma_2 \doteq \beta_2 \to \gamma_1 \wedge \gamma_2 \doteq \beta$
(4.2)	$\top$	$\alpha \doteq \beta_1 \to \gamma_1 \wedge \gamma_2 \doteq \beta_2 \to \gamma_1$ $\wedge \gamma_2 \doteq \beta \wedge \beta_2 \doteq \beta$

The flattening phase is the identity in this case. After the constraint resolution phase we obtain:

$$\begin{aligned} \alpha &\doteq \beta_1 \to \gamma_1 \\ \gamma_2 &\doteq \gamma_2 \to \gamma_1 \\ \beta &\doteq \gamma_2 \end{aligned}$$

The rebuilding phase yields

$$\begin{aligned} \alpha &\doteq \beta_1 \to \gamma_1 \\ \beta &\doteq \mu\gamma_2.\gamma_2 \to \gamma_1 \end{aligned}$$

As a last example we take Curry's paradoxical combinator Y [4, page 131]. It is well-known that this example needs recursive types for the inference but the result type is not recursive. This is why in type inference systems without infinite types, the fixed point operator *can* be added as a typed constant. In those systems, it *must* be added if we need it. The term is $\mathsf{Y} \equiv \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$. Using a leftmost innermost strategy for the constraint extraction phase, we obtain the constraint

$$\begin{aligned} \alpha &\doteq \beta_1 \to \gamma_1 \\ \gamma_2 &\doteq \beta_2 \to \gamma_1 \\ \gamma_2 &\doteq \beta_3 \to \gamma_3 \\ \gamma_4 &\doteq \beta_4 \to \gamma_3 \\ \gamma_4 &\doteq \beta_1 \\ \gamma_5 &\doteq \beta_5 \to \beta_4 \\ \gamma_5 &\doteq \beta_3 \\ \beta_5 &\doteq \beta_3 \\ \beta_2 &\doteq \beta_6 \to \gamma_6 \\ \gamma_7 &\doteq \beta_7 \to \gamma_6 \end{aligned}$$

$$\begin{aligned}
\gamma_7 &\doteq \beta_1 \\
\gamma_8 &\doteq \beta_8 \to \beta_7 \\
\gamma_8 &\doteq \beta_6 \\
\beta_8 &\doteq \beta_6
\end{aligned}$$

The flattening phase is again the identity. After the constraint resolution phase we obtain:

$$\begin{aligned}
\alpha &\doteq \beta_1 \to \gamma_6 \\
\gamma_2 &\doteq \beta_3 \to \gamma_6 \\
\beta_1 &\doteq \gamma_6 \to \gamma_6 \\
\beta_3 &\doteq \beta_3 \to \gamma_6
\end{aligned}$$

Finally, the rebuilding phase yields

$$\begin{aligned}
\alpha &\doteq (\gamma_6 \to \gamma_6) \to \gamma_6 \\
\gamma_2 &\doteq (\mu\beta_3.\beta_3 \to \gamma_6) \to \gamma_6
\end{aligned}$$

The interesting type is that of α. The variable γ_6 is not constrained. The recursive type of γ_2 remains hidden. The recursive type of γ_2 is not in its minimal form. Its minimal form is $\mu\beta_3.\beta_3 \to \gamma_6$.

5.4 The let Construct

The let construct is important for type inference because it is the source of polymorphism [63]. From the evaluation point of view,

$$\text{let } x = e \text{ in } e' \quad \text{and} \quad (\lambda x.e')\ e$$

are equivalent. From the type inference point of view, they are different, and the reason is the following. Looking at $\lambda x.e'$ alone, we have a function for which we have to infer a type. We do not know to what arguments the function may be applied, i.e., we do not know what x will be substituted for. In the 'let' construct we know that $\lambda x.e'$ will only be applied to e. Therefore, we are not obliged to assume one consistent type for x everywhere in e' but we can imagine that the β-reduction has already happened (by program transformation) and instead infer a type for $[e/x]e'$, where $[e/x]e'$ denotes

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash [e/x]e' : \tau}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau} \quad \text{(LET)}$$

Figure 5.2: Type inference rule for 'let'

the capture avoiding substitution of e for x in e'. Note that we have let, *not* letrec, i.e. if x appears in e' it is global. There is no recursion. Therefore, we can add the inference rule (LET) of Fig. 5.2 to our rules. The proof that well-typed terms do not go wrong is straightforward but tedious [62]. The semantics of the 'let' construct is to evaluate e and then perform the β-reduction. In the premise, the sequent $\Gamma \vdash e : \sigma$ guarantees that e has a type at all. This part is only necessary if e does not occur in e'. Due to confluence, it does not matter if we evaluate e first and then β-reduce or vice versa. Thus, type checking $[e/x]e'$ is sufficient.

Can the application of rule (LET) lead to non-termination, or, are there any proof trees that contain an infinite number of applications of rule (LET)? The answer to the question is "no". This result seems to be folklore and the reasoning might be: there is no recursion because x does not appear in e and, thus, no danger. Since the exact proof is not so close at hand, we list it here.

Proposition 5.4.1 Each proof tree contains a finite number of applications of rule (LET).

Proof: Every variable in our term gets a number. We assume that there are no name conflicts. Every variable bound by λ gets the number 0. All variables x that are introduced by

$$\text{let } x = e \text{ in } e'$$

where e does neither contain further let-declarations nor further let-bound variables, get the number 1. Now, variables introduced by

$$\text{let } x = e \text{ in } e'$$

where e contains neither further let-declarations nor variables do not have a number yet. They get the number of the maximum of the numbers in e plus one.

We define a function letsize that uses the function number as described above.

$$\begin{aligned}
\text{letsize}(x) &= \{\!|\ \text{number}(x)\ |\!\} \\
\text{letsize}(\langle \overline{a} \mapsto \overline{e} \rangle) &= \bigcup_{e \in \overline{e}} \text{letsize}(e) \\
\text{letsize}(e.a) &= \text{letsize}(e) \\
\text{letsize}(\lambda x.e) &= \text{letsize}(e) \\
\text{letsize}(e\ e') &= \text{letsize}(e) \cup \text{letsize}(e') \\
\text{letsize}(\text{let } x = e \text{ in } e') &= \text{letsize}(e') \cup \text{number}(x)
\end{aligned}$$

Note that the union symbols denote the union on multisets, meaning that $\{\!|\ a\ |\!\} \cup \{\!|\ a\ |\!\} = \{\!|\ a, a\ |\!\}$ etc. By construction of the function number, we have that every let-bound variable has a number strictly greater than the maximum of the numbers occurring in the expression that defines it. Therefore, replacing

$$\text{let } x = e \text{ in } e'$$

by $[e/x]e'$ strictly decreases the function letsize (we take the multiset ordering) no matter if x occurs in e' or not. Also note that the substitution does not violate the invariant that the number of the let-bound variable x is strictly greater than the maximum of the numbers of let-bound variables in its defining term. □

Definition 5.4.2 Here comes the constraint extraction rule for let:

$$\frac{\exists \overline{\alpha}((\Gamma \vdash \text{let } x = e \text{ in } e' : \alpha \wedge \omega) \wedge \rho;\ \phi)}{\exists \overline{\alpha}((\Gamma \vdash e : \beta \wedge [e/x]e' : \alpha \wedge \omega) \wedge \rho;\ \phi)} \left\{ \begin{array}{l} \beta \text{ fresh} \\ \text{sort}(\overline{\beta}) = \overline{\top} \end{array} \right. \tag{5.8}$$

The invariance of the new rule is easy to see but tedious to show.

5.5 The let Construct Revisited

The rule (LET) for 'let' can be easily justified and the extension of the existing inference rules and the algorithm comes without extra machinery. In the general case, the substitution of let-bound variables by their defining terms can be quite costly since e' is in general not linear in x. We would have to repeatedly type check e. This is avoided by introducing so-called *type schemes* [63]. The checking of e leads to the same constraints everywhere. Thus, we can check e once and use its type as a type scheme. Before we present an alternative to rule (LET), we need some definitions.

Definition 5.5.1 A *type scheme* is an **RT**-type possibly containing *generic type variables*. Generic type variables come from a new set of variables and are denoted by $\tilde{\alpha}, \tilde{\beta}, \ldots$. We use $\tilde{\sigma}, \tilde{\tau}, \ldots$ to denote type schemes.

We denote the set of generic type variables of a type scheme $\tilde{\sigma}$ by $\mathcal{G}\tilde{\sigma}$. Type schemes stand for sets of types. For example, the idea behind the type scheme $\tilde{\alpha} \to \tilde{\alpha}$ is the set of all function types with identical domains and codomains. This set contains elements such as int→int, bool→bool, and $(\langle a : \text{int}\rangle \to \langle b : \text{int}\rangle) \to (\langle a : \text{int}\rangle \to \langle b : \text{int}\rangle)$. The set of types that a type scheme represents is formalized in the definition of the generalization relation.

Definition 5.5.2 The type scheme $\tilde{\sigma}$ *generalizes* the type σ, in symbols $\tilde{\sigma} \succ \sigma$, if there is a substitution θ from generic type variables to types such that $\theta\tilde{\sigma} = \sigma$. Overloading the symbol $\succ$, we define a more-general-than relation on type schemes.

$$\tilde{\sigma} \succ \tilde{\tau} \quad \text{if and only if} \quad \tilde{\tau} \succ \sigma \text{ implies } \tilde{\sigma} \succ \sigma$$

Note that $\_ \succ \_$ is a preorder on type schemes.

Definition 5.5.3 We define the *generalization* respecting the free type variables of the type environment Γ.

$$\lceil \sigma \rceil^{\Gamma} = \theta\sigma$$

where θ is a substitution from type variables to new generic type variables and $dom(\theta) = fv(\sigma) \backslash fv(\Gamma)$.

$$\frac{}{\Gamma \vdash x : \tau} \left\{ \tau = \lfloor \Gamma(x) \rfloor \right. \qquad \text{(VAR')}$$

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \cdot [x : \lceil \sigma \rceil^{\Gamma}] \vdash e' : \tau}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau} \qquad \text{(LET')}$$

Figure 5.3: Type inference rules for 'let' with type schemes

Definition 5.5.4 Instantiation of all generic type variables to new type variables.

$$\lfloor \tilde{\sigma} \rfloor = \theta \tilde{\sigma}$$

where θ is a substitution from generic type variables to new type variables and $dom(\theta) = \mathcal{G}\tilde{\sigma}$.

Note that the instantiation also depends on the type environment Γ since generic type variables are mapped to *new* type variables, where 'new' means not contained in Γ. This dependence of Γ is however not as important as for generalization, and we decided not to make it explicit in the notation (cf. [79]).

We have the necessary definitions for introducing the "efficient" inference rules for 'let' in Fig. 5.3. Rule (VAR') overwrites rule (VAR). These rules allow for the same typings as rule (LET). A formal derivation of a system with rule (LET) via a system with rules (VAR') and (LET') to an efficient algorithm can be found in [80].

Readers familiar with type inference systems will notice the absence of a generalization rule and instantiation rule [25]. Instead, generalization and instantiation are done precisely when variables are declared and used, respectively. In later sections, we will introduce imperative features. For purely functional type inference systems, both formulations are equivalent, whereas, for systems with imperative features, there are differences [96].

5.5.1 Type Reconstruction

In the last part, we have shown how let-polymorphism (that is neither top level nor letrec) can be inferred by an algorithm that works with frames. Frames have a semantics and all rules are semantics preserving. Furthermore, it is irrelevant which proof obligation we choose to process next or whether we prefer to apply a resolution rule to the constraint.

We now get much closer to the algorithm that one would use in practice for type inference with 'let', i.e. the Damas-Milner algorithm. We will leave the terrain of theory and logic while preserving the frame style for writing the algorithm. The algorithm to follow is more sequential than the previous one but still not completely. From here on, we dive into practical computer science. The preservation of the algorithm's style should facilitate the task of seeing that the previous algorithm and the following one compute exactly the same principal types. A frame consisted of existentially quantified variables, a conjunction of scopes, and a constraint. The new algorithm works on a pair consisting of a stack of scopes and a constraint. The quantifiers were omitted because there is no more semantics. The type environments contain type schemes or pairs of type environments and type variables. These pairs are used for the generalization of type variables.

Whereas in the let-rule with substitutions our algorithm conserved its character trait of not imposing any order on the proof obligations to work on or the resolution steps to take place, the new algorithm will force the constraint to be resolved at certain points and also impose an order on the scopes. When type schemes are created by generalization the constraint is brought into normal form and rebuilt. Extraction rules are always applied to the leftmost scope.

Definition 5.5.5 We list the most important simplification rules for **RF** with let. The rules for the other constructs are analogous.

$$\frac{(\Gamma \vdash \text{let } x = e \text{ in } e' : \alpha \wedge \omega) \bullet \rho;\ \phi}{\Gamma \vdash e : \beta \bullet (\Gamma \cdot [x : (\Gamma, \beta)] \vdash e' : \alpha \wedge \omega) \bullet \rho;\ \phi} \left\{ \begin{array}{l} \beta \text{ fresh} \\ \text{sort}(\beta) = \top \end{array} \right. \tag{5.9}$$

$$\frac{(\Gamma \vdash x : \alpha \wedge \omega) \bullet \rho;\ \phi}{\Gamma \vdash \omega \bullet \rho;\ \phi \wedge \alpha \doteq \lfloor \tau \rfloor} \left\{ \Gamma(x) = \tau \right. \tag{5.10}$$

$$\frac{(\Gamma \vdash x : \alpha \wedge \omega) \bullet \rho;\ \phi}{(\Gamma \cdot [x : \tau] \vdash \omega) \bullet \rho;\ \phi} \begin{cases} \Gamma(x) = (\Gamma', \beta) \\ \tau = \lceil \phi(\beta) \rceil^{\Gamma'} \end{cases} \tag{5.11}$$

$\phi(\alpha)$ denotes the lookup of a type variable α in the constraint ϕ. For this operation to be meaningful and useful, it is a prerequisite that the constraint is in rebuilt normal form.

6. An Imperative Language

The language of chapter 5 will be extended to contain imperative features. This extension is similar to going from the functional polymorphic sublanguage of SML (the essence of SML [67]) to full SML with references [65]. Imperative features are an essential part of object-oriented languages. Objects have an internal state that changes over time. This cannot be expressed in a purely functional framework.[1] The combination of state and polymorphic type checking has become a new research field of its own [24, 56, 83, 94, 93, 96, 107, 108]. Polymorphism has to be used with great care in the presence of references to the store. We will show this later by a standard example.

O'SMALL is a language that uses imperative features in a very restricted way. The internal state of objects is hidden in private variables (*instance variables*). These variables are accessed only by their class methods because we have *encapsulated instance variables* [88]. Although objects can be known globally in the system, these objects can only change their state as a result of messages sent to them. The reaction to these messages is the execution of their own methods. Imperative features have a "more local" character in object-oriented languages. Therefore, it is relatively easy to eliminate some of them already in the translation process (cf. section 7.3). For the remaining imperative features we can choose any of the recent imperative type inference systems [56, 94, 93, 96, 107].[2] The results of these approaches are clearly applicable to our language because it uses a subset of the imperative features treated in these approaches. We have chosen the imperative type discipline of Tofte [96]. This choice is discussed in section 8.2.

6.1 Expressions

The language we define in this section is a λ-calculus with imperative features [35, 64] and records.

[1] Of course, it can be modeled in a purely functional framework by passing a state argument around. However, we do not think that this is the proper way of expressing state in practice.

[2] [108] does not seem appropriate here (cf. section 8), although its simplicity is appealing.

Definition 6.1.1 (Syntax of RFI)
The language **RFI** is defined by the following abstract syntax.

e	::=	x	variable
	|	$\lambda x.e$	abstraction
	|	$e\ e$	application
	|	let $x = e$ in e	let-declaration
	|	ref e	new cell
	|	$!x$	dereferencing
	|	$u := e$	assignment
	|	$\langle \overline{a} \mapsto \overline{e} \rangle$	record
	|	$e.a$	selection

The first part is ordinary λ-calculus with "let". The following three clauses are the imperative features of our language. Some readers may note the absence of a sequence $e_1; e_2$ of expressions. The introduction of the sequence is not necessary because $e_1; e_2$ can be regarded as syntactic sugar for let $\_ = e_1$ in e_2 where the underscore stands for a dummy variable not used in e_2.

The dynamic semantics is specified in natural style. This approach has also been used in [1, 96, 94]. The evaluation function

$$\_, \_ \vdash \_ \rightsquigarrow \_, \_ : \text{store} \times \text{envir} \times \text{term} \rightarrow \text{value} \times \text{store}$$

is defined by rules (6.1) through (6.9). It describes the evaluation of expressions to values. The evaluation is performed in the context of an environment E and a store s. This is expressed by the turnstile symbol. Each evaluation results not only in a value but also in a, probably changed, store. This is the *side effect* of the evaluation. *Computable values*, ranged over by variable v, are primitive constants c and closures. A *closure* $\langle x, e, E \rangle$ is a triple consisting of a variable x, an expression e, and an environment E.

An *environment* is a finite mapping from term variables to values and locations. *Locations* come from a countable, totally ordered set and are ranged over by variables r. $E \cdot [x \mapsto v]$ denotes an environment E updated by v at x.

A *store* is a finite mapping from locations r to values. It is written as $[\overline{r} \mapsto \overline{v}]$. The notation for updating is the same as for environments. We now list the rules for the evaluation function.[3]

$$\frac{}{s,\ E \vdash x \ \rightsquigarrow\ v,\ s} \left\{ E(x) = v \right. \tag{6.1}$$

$$\frac{}{s,\ E \vdash \lambda x.e \ \rightsquigarrow\ \langle x, e, E \rangle,\ s} \tag{6.2}$$

$$\frac{\begin{array}{c} s,\ E \vdash e_1 \ \rightsquigarrow\ \langle x, e, E' \rangle,\ s' \\ s',\ E \vdash e_2 \ \rightsquigarrow\ v_2,\ s'' \\ s'',\ E' \cdot [x \mapsto v_2] \vdash e \ \rightsquigarrow\ v,\ s''' \end{array}}{s,\ E \vdash e_1\ e_2 \ \rightsquigarrow\ v,\ s'''} \tag{6.3}$$

$$\frac{\begin{array}{c} s,\ E \vdash e_1 \ \rightsquigarrow\ v_1,\ s' \\ s',\ E \cdot [x \mapsto v_1] \vdash e_2 \ \rightsquigarrow\ v_2,\ s'' \end{array}}{s,\ E \vdash \text{let } x = e_1 \text{ in } e_2 \ \rightsquigarrow\ v_2,\ s''} \tag{6.4}$$

$$\frac{\begin{array}{c} s,\ E \vdash e_1 \ \rightsquigarrow\ v_1,\ s' \\ s' \cdot [r \mapsto v_1],\ E \cdot [x \mapsto r] \vdash e_2 \ \rightsquigarrow\ v_2,\ s'' \end{array}}{s,\ E \vdash \text{let } x = \text{ref } e_1 \text{ in } e_2 \ \rightsquigarrow\ v_2,\ s''} \left\{ r \notin dom(s') \right. \tag{6.5}$$

$$\frac{}{s,\ E \vdash !x \ \rightsquigarrow\ v,\ s} \left\{ \begin{array}{lcl} E(x) & = & r \\ s(r) & = & v \end{array} \right. \tag{6.6}$$

$$\frac{s,\ E \vdash e \ \rightsquigarrow\ v,\ s'}{s,\ E \vdash x := e \ \rightsquigarrow\ v,\ s' \cdot [r \mapsto v]} \left\{ \begin{array}{lcl} E(x) & = & r \\ s(r) & = & v' \end{array} \right. \tag{6.7}$$

[3]It is obvious that this relation is a function.

$$\frac{\begin{array}{c} s,\ E \vdash e_1 \rightsquigarrow v_1,\ s_1 \\ s_1,\ E \vdash e_2 \rightsquigarrow v_2,\ s_2 \\ \vdots \\ s_{n-1},\ E \vdash e_n \rightsquigarrow v_n,\ s_n \end{array}}{s,\ E \vdash \langle \overline{a} \mapsto \overline{e} \rangle \rightsquigarrow \langle \overline{a} \mapsto \overline{v} \rangle,\ s_n} \tag{6.8}$$

$$\frac{s,\ E \vdash e \rightsquigarrow \langle \overline{a} \mapsto \overline{v},\ a \mapsto v \rangle,\ s'}{s,\ E \vdash e.a \rightsquigarrow v,\ s'} \tag{6.9}$$

Rule (6.1) is applicable if there is a value v in the environment for the variable x. The store remains unchanged. Rule (6.2) transforms any λ-abstraction into a closure. The closure is needed to evaluate the expression e in the environment of the declaration of the λ-abstraction. Rule (6.3) evaluates e_1 to a closure, then e_2 to a value, and, finally, performs the application. The corresponding, probably changing, stores are passed through in this order. In rule (6.4), the expression e_1 is evaluated and its value put into the environment. In rule (6.5), the expression e_1 is evaluated and its value is inserted into the store at a new location. Note that the address r continues to "live" in the store s''. In rule (6.6), the environment must contain an address that has a value v in the store. This value will be retrieved (dereferencing). Assignments as in rule (6.7) can only be applied to cells that have been introduced by rule (6.5) previously. In rule (6.8), the expressions in the record are evaluated from left to right. As opposed to chapters 4 and 5, the order of labels plays a rôle now. We take the syntactic order. Rule (6.9) is record selection as one would expect it.

Looking at rules (6.5), (6.6), and (6.7) one might think that a two-stage mechanism for cells is unnecessary. By "two-stage" we mean that addresses are retrieved in the environment E and then fetched in the store s. Why do we not directly change the entries in the environment? The answer lies in rule (6.3). When e_1 is evaluated we put the actual environment into its closure. In the subsequent "step" e_2 is evaluated. This evaluation may have side effects on the store. When, finally, the closure $\langle x, e, E' \rangle$ is "opened" in order to compute the function, we retrieve E'. We need the original E' here but the store may have changed in the meantime. Therefore, the two-step mechanism is used for discerning the different ways that environment and store change.

6.2 Typings

The following well-known example shows where polymorphic references go wrong.

```
let c = ref (λx.x)
in  c := (λx.x+1);
    !c true
end
```

The semicolon in $e_1; e_2$ is syntactic sugar for let $\_ = e_1$ in e_2. The underline is a dummy variable that cannot occur anywhere else. Using a naïve extension of the type inference rules of Fig. 5.3, we could infer a polymorphic type for c by generalizing all type variables, and the program would be accepted by the type checker. However, this program goes wrong.

In order to avoid problems with the generalization of type variables, Tofte [96] introduces *imperative type variables* for types that may occur in the store. These imperative type variables are not generalized in certain cases, namely when the expression to which a variable in a let-declaration is bound is expansive.

Definition 6.2.1 An expression is said to be *non-expansive* if it is a variable, a constant, a record, or a λ-abstraction. All other expressions, i.e. applications and let-expressions are said to be *expansive.*

Expansive expressions have the potential of creating new cells in the store or, in other words, to expand the domain of the store (whence the name).

Type variables connected to values that may appear in the store are imperative. Imperative type variables are opposed to *applicative type variables.* For dealing with imperative type variables and imperative types, Tofte has the following rules.

- Imperative types contain no applicative type variables.
- A value to be stored must have an imperative type.
- Substitution maps imperative type variables to imperative types.

- Applicative type variables may be mapped to types containing imperative type variables.

These rules call for a treatment with order-sorted logic.

Definition 6.2.2 The set of *sorts* S contains

- all sorts of definition 3.2.1,
- for every consistent sort $\mathbf{s}$ of definition 3.2.1 there is a corresponding *imperative sort* $\mathbf{s}_{imp}$,
- a new sort for reference types called $\mathbf{r}$.

Imperative sorts are written with an index $_{imp}$.

Definition 6.2.3 (Signature RFIT) We extend the signature **RT** (definition 3.2.2) by infinitely many variables for the sorts of definition 6.2.2 and the subsort declaration

$$\mathbf{s}_{imp} < \mathbf{s}$$

if $\mathbf{s}$ is a consistent sort from definition 3.2.1 and $\mathbf{s}_{imp}$ the corresponding imperative sort. Furthermore, we have

$$\mathbf{s} < \mathbf{t} \Rightarrow \mathbf{s}_{imp} < \mathbf{t}_{imp}$$

if $\mathbf{s}$ and $\mathbf{t}$ are consistent sorts from definition 3.2.1 and $\mathbf{s}_{imp}$ and $\mathbf{t}_{imp}$ are the corresponding imperative sorts. There is a new constructor with the following sort declarations:

$$. \mathit{ref} : \top \rightarrow \mathbf{r}$$

$$. \mathit{ref} : \top_{imp} \rightarrow \mathbf{r}_{imp}$$

For every sort declaration $f : \overline{\mathbf{s}} \rightarrow \mathbf{t}$ of definition 3.2.2 we add the sort declaration

$$f : \overline{\mathbf{s}_{imp}} \rightarrow \mathbf{t}_{imp}$$

By these definitions we obtain a derived sort hierarchy with a common bottom element $\bot$. Since the extension of the denotations of sorts and the theory is straightforward, we turn directly to the new type inference rules. In addition to the generalization of all free type variables, we need a generalization restricted to applicative type variables. The following definition is more general because we also have applicative types.

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \cdot [x : \lceil\sigma\rceil^{\Gamma}] \vdash e' : \tau}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau} \left\{ e \text{ non-expansive} \qquad \text{(LEN)} \right.$$

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \cdot [x :^{app} \lceil\sigma\rceil^{\Gamma}] \vdash e' : \tau}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau} \left\{ e \text{ expansive} \qquad \text{(LEX)} \right.$$

Figure 6.1: Imperative inference rules

Definition 6.2.4 A type σ is *applicative* if $\text{sort}(\sigma) \not\leq \top_{imp}$.

Definition 6.2.5 We define the *generalization* of applicative type variables respecting the free applicative type variables of the type environment Γ.

$$^{app}\lceil\sigma\rceil^{\Gamma} = \theta\sigma$$

where θ is a substitution from applicative type variables to new generic type variables and $dom(\theta) = fv_{app}(\sigma) \backslash fv_{app}(\Gamma)$.

Definition 5.5.4 of instatiation remains unchanged. Note however that imperative type variables are mapped to imperative types. This is guaranteed by the order-sorted formalism.

The type inference rules by Tofte come in addition to rules (ABS) and (APP) of Fig. 5.1 and rule (VAR') of Fig. 5.3. The new rules are contained in Fig. 6.1.

The new constructs ref, :=, and ! of definition 6.1.1 are considered as functions (assignment is written as infix). The initial type environment contains the type schemes:

$$\begin{aligned} \Gamma(ref) &= \tilde{\alpha} \to \tilde{\alpha} \quad \left\{ \text{sort}(\tilde{\alpha}) = \top_{imp} \right. \\ \Gamma(:=) &= \tilde{\alpha}\ ref \to \tilde{\alpha} \to unit \\ \Gamma(!) &= \tilde{\alpha}\ ref \to \tilde{\alpha} \end{aligned}$$

We conjecture that soundness and all other nice properties like principal types follow from the properties we have proved in chapters 4 and 5 and the proofs in [95]. Records and imperative features are orthogonal.

6.3 Type Reconstruction

Type reconstruction can be done with the same structures as in definition 5.5.5. Type environments may now contain either type schemes or what we call suspended generalizations.

Definition 6.3.1 A *suspended generalization* is a triple (Γ, e, α) where Γ is a type environment, e an expression, and α a type variable.

Definition 6.3.2 We list the new simplification rules for imperative types. The rules for the other constructs remain the same.

$$\frac{(\Gamma \vdash \text{let } x = e \text{ in } e' : \alpha \wedge \omega) \bullet \rho;\ \phi}{\Gamma \vdash e : \beta \bullet (\Gamma \cdot [x : (\Gamma, e, \beta)] \vdash e' : \alpha \wedge \omega) \bullet \rho;\ \phi} \begin{cases} \beta \text{ fresh} \\ \text{sort}(\beta) = \top \end{cases} \quad (6.10)$$

$$\frac{(\Gamma \vdash x : \alpha \wedge \omega) \bullet \rho;\ \phi}{\Gamma \vdash \omega \bullet \rho;\ \phi \wedge \alpha \doteq \lfloor \tau \rfloor} \begin{cases} \Gamma(x) = \text{entry} \\ \tau = \text{scheme(entry)} \end{cases} \quad (6.11)$$

where

$$\text{scheme(entry)} = \begin{cases} \text{entry} & \text{if entry is a type scheme} \\ \lceil \phi(\beta) \rceil^{\Gamma'} & \text{if entry} = (\Gamma', e, \beta),\ e \text{ non-expansive} \\ {}^{app}\lceil \phi(\beta) \rceil^{\Gamma'} & \text{if entry} = (\Gamma', e, \beta),\ e \text{ expansive} \end{cases} \quad (6.12)$$

Rule (6.10) binds x to a suspended generalization. This has the following effect. The algorithm works on the stack of scopes from left to right. When x is entered into the type environment in rule (6.10), the scope left to this one is not started yet and consequently ϕ does not contain any information on β. It is only after having finished with the scope $\Gamma \vdash e : \beta$ that ϕ contains constraints with respect to β and that $\phi(\beta)$ makes sense.

Rule (6.11) operates as before (rule (5.10)) in the case of a type scheme. In case of a suspended generalization it performs it and thus creates a new type scheme. Generalization is performed for each occurrence of x in its scope. This is correct but should be avoided in an implementation for the sake of efficiency.

The constraint simplification rules of section 4.3.2 are extended by

$$\frac{\begin{array}{l}\phi \\ \alpha \doteq \beta\ ref \\ \alpha \doteq \gamma\ ref\end{array}}{\begin{array}{l}\phi \\ \alpha \doteq \beta\ ref \\ \beta \doteq \gamma\end{array}} \tag{6.13}$$

It is obvious that all properties of the simplification rules remain unchanged by adding this harmless rule. No other rule is necessary because the rules that deal with sorts are general enough to take care of imperative types. E.g., the condition that imperative types may not contain applicative type variables is guaranteed by rules (4.16) and (4.17).

7. An Object-Oriented Language

This chapter concludes the construction of Fig. 2.1 on page 15 by defining the translation function from O'SMALL to **RFI**. After the completion of our system's description we assess it in terms of **RFI** and O'SMALL-programs.

7.1 Objects, Classes, and Wrappers

Objects are the focus of object-oriented languages. An *object* consists of an *interface* and an internal *state*. The interface of an object is a collection of methods. A *method* is similar to a *procedure* in procedural languages. In more elaborated object-oriented languages which feature the concept of specification, the interface does not consist of the methods themselves but of their specification. The implementation is hidden. The specification consists of the method names, the number of parameters, the types and so forth. The internal state of an object consists of a collection of so-called *instance variables*. Instance variables are invisible from the outside. They can only be accessed indirectly by methods. This way, *data abstraction* is achieved.

Objects communicate by *message passing*. A *message* consists of a *message selector* and a list of parameters. The *receiver* of the message is an object and it can react in one of the following ways:

- Hopefully, the receiver understands the message, i.e., there is a method with a *method name* that corresponds to the message selector, *and* the number of parameters is correct. In this case, the corresponding method is executed. Its result is returned to the *sender* of the message. The receiver's internal state may change as a *side effect* of the message received.

- The abnormal case is that the receiver does not "understand" the message, i.e., the number of parameters is wrong or there is no corresponding method at all. In this case, the program halts with an error. To avoid this to happen is the task of type checking.

A concept that has caused confusion in the past is that of a *class*. The original meaning of the word "class" is a "group having qualities of the same kind". In the world of object-oriented languages, we can translate

"class" by "object factory" or "object stencil". This definition in mind, we see that, indeed, objects that come out of the same factory must have qualities of the same kind. However, the distinction is important: objects only belong to a class if they are produced by it. Therefore, we may have two distinct classes that create objects with identical properties. Perhaps the most interesting feature of object-oriented languages is *class inheritance.* Class inheritance allows us to create new classes by modifying existing ones. If this modification is done properly, the result of using class inheritance is a *class hierarchy.*[1] It is hard to describe in formal terms how to use class inheritance properly. Yet, if it is done properly, the resulting class hierarchy resembles the hierarchies that categorize the world, e.g. the classification of animals and the like. Thus, class inheritance is a means of structuring the world. The created systems are easier to maintain. Hand in hand with modifying existing classes goes the concept of *code sharing.* It is true that code is shared and that this fact results in a better maintainability – again assuming an ideal programmer. However, it is an error to believe that class inheritance always saves time just because code is shared. On the contrary, a lot of consideration has to go into the creation of the proper classification.

Before deviating further, let us now get to the point and model all the flowery terms we have introduced above. The model looks quite simple but is the result of a long development. We use the concept of *wrappers* as introduced by Cook [21]. Cook used wrappers for the denotational description of object-oriented programming languages and we have based the denotational semantics of O'SMALL [38] on his work. We will now use them for translating O'SMALL into the intermediate language **RFI**. We start by explaining the above concepts one by one. The level of discussion remains informal in order to concentrate on the essential aspects. We assume familiarity with the λ-calculus.

An *object* is a *record* of methods and an internal state. Let us define the counter object c by

$$c \quad = \quad \langle \text{increment} \mapsto n := n + 1,\ \text{value} \mapsto n \rangle$$

We assume that the object c has an instance variable n that has been initialized to 0. Instance variables and their initialization are suppressed in

[1] We only use *single inheritance.*

the informal discussion of this section. The entries of the record contain the code of the methods. In this case, we have parameterless methods. Message passing is *record selection.* We can ask for the value of the above counter object by c.value. The result is 0. We can change the state of our object by c.increment. If we evaluate c.value now, we get 1 as result.

Programming is boring if there is no recursion. In functional or procedural languages, we can write recursive function declarations like

$$\text{fac} \quad = \quad \lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * \text{fac}(x-1)$$

The semantics of such a recursive definition is the least fixed point in an appropriate domain [42]. In the object-oriented world, functions cannot be applied in this direct manner. We always need to send a message to some receiver. If we want to program the factorial function in an object-oriented language, we must create an object that "understands" the message "fac". How can it use "fac" recursively? It must send "fac" to itself. We introduce a *pseudo variable* "self"[31] that stands for the receiver of the current message. An object that calculates the factorial function looks like this:

$$\langle \text{fac} \mapsto \lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * \text{self.fac}(x-1) \rangle$$

Note that this object has no internal state. In this object, "self" is an unbound variable. To obtain the object we really want, we have to use an old trick: we λ-abstract "self" and apply the *fixed-point operator* to the resulting function:

$$\mathsf{Y}(\lambda \text{self.}\ \langle \text{fac} \mapsto \lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * \text{self.fac}(x-1) \rangle)$$

Part of the last expression is a class. How can a class be (a syntactical) part of an object? Remember that a class is an object factory. In our example the class is this function:

$$\lambda \text{self.}\ \langle \text{fac} \mapsto \lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * \text{self.fac}(x-1) \rangle$$

We can create objects by applying the fixed-point operator to it. In the appropriate domains [42], the weakest fixed point is uniquely determined and we would always get the same object! Yes, but the underlying language contains state. The state may change and so may the objects. In the factorial example, all objects created by this class would indeed be the same. You

could construct an example of two counters which are the same at their creation and then differ according to the number of "increment"-messages sent to each of them.

Now let us take some existing objects and modify them. We can obtain a resettable counter for even numbers by

$$\langle \text{reset} \mapsto n := 0,\ \text{value} \mapsto 2 * n \rangle$$
$$\oplus\ \langle \text{increment} \mapsto n := n + 1,\ \text{value} \mapsto n \rangle$$

We use *left-preferential concatenation* for this purpose. Left-preferentiality means that, in the case of name conflicts, the record on the left-hand side overwrites the components of the right-hand side. In the example, there is one name conflict and the result of the concatenation is

$$\left\langle \begin{array}{rcl} \text{reset} & \mapsto & n := 0 \\ \text{increment} & \mapsto & n := n + 1 \\ \text{value} & \mapsto & 2 * n \end{array} \right\rangle$$

More meaningful examples will follow.

Our next task is to combine modification with *self-reference*. Since we have a *class-based* language, we perform all modifications on classes instead of directly on objects. Classes are functions that λ-abstract "self". Their body is a record. We cannot use $\oplus$ for concatenating two classes because $\oplus$ operates on records, not on functions. We have to "lift" the operator $\oplus$ in order to deal with the new situation. The following definition is generic.

Definition 7.1.1 We define the *lifting* of the binary operator $*$ by

$$a\ \boxed{*}\ b = \lambda s.(a\ s) * (b\ s)$$

Let us now combine the counter and the factorial functionality. We obtain the class of objects having the combined functionality by

$$(\lambda \text{self}.\ \langle \text{fac} \mapsto \lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * \text{self.fac}(x - 1) \rangle)$$
$$\boxed{\oplus}\quad (\lambda \text{self}.\ \langle \text{increment} \mapsto n := n + 1,\ \text{value} \mapsto n \rangle)$$

We obtain the result

$$\lambda \text{self}.\ \left\langle \begin{array}{rcl} \text{fac} & \mapsto & \lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * \text{self.fac}(x - 1) \\ \text{increment} & \mapsto & n := n + 1 \\ \text{value} & \mapsto & n \end{array} \right\rangle$$

In this example, "self" is only used once in the body. Note, however, that "self" is distributed by the lifting operation such that it is the same everywhere in the resulting object (no schizophrenia).

The last step of this introduction will lead us to the so-called wrappers. So far we can modify classes by extending their interface and replacing whole methods by new ones. To enhance the expressibility of inheritance, we also want to be able to modify existing methods incrementally. An example of this is Fig. 2.2. There, the distance from the origin of a circle is calculated in the same way as that of a point, except that we have to subtract the radius of the circle. We have to be able to refer to methods that are just being overwritten. This is realized by the *pseudo variable* "super". With this pseudo-variable we designate the object itself as if it belonged to the superclass of the class we are just defining. Before we do this complicated operation on the class level, let us exemplify it on the object level. We redefine the increment method of the object

$$\langle \text{increment} \mapsto n := n + 1,\ \text{value} \mapsto n \rangle$$

by applying the existing method twice:

$$\langle \text{increment} \mapsto \text{super.increment; super.increment} \rangle$$

As before, we have here an unbound variable. Therefore, we λ-abstract it:

$$\lambda\text{super}.\ \langle \text{increment} \mapsto \text{super.increment; super.increment} \rangle$$

The last expression is an object modifier. This modifier takes an argument (for "super") and can then be concatenated with another object. Here, the other object is the same as the argument for super. Therefore, an auxiliary definition is useful.

Definition 7.1.2 We define a binary operator for *record modification*: :

$$f \triangleright b = (f\ b) \oplus b$$

Using this operator, we can combine the modifier and the record

$$\langle \text{increment} \mapsto \text{super.increment; super.increment} \rangle \triangleright \langle \text{increment} \mapsto n := n + 1,\ \text{value} \mapsto n \rangle$$

and get the result

$$\langle \text{increment} \mapsto n := n + 1; n := n + 1,\ \text{value} \mapsto n \rangle$$

Now we lift the record modification operator to deal with classes. We call the modifiers that operate on classes *wrappers*. A wrapper is a function with "super" and "self" λ-abstracted. The wrapper that corresponds to the last modification is

$$\lambda\text{self}.\ \lambda\text{super}.\ \langle \text{increment} \mapsto \text{super.increment; super.increment} \rangle$$

It can be applied to the class

$$\lambda\text{self}.\ \langle \text{increment} \mapsto n := n + 1,\ \text{value} \mapsto n \rangle$$

by

$$(\lambda\text{self}.\ \lambda\text{super}.\ \langle \text{increment} \mapsto \text{super.increment; super.increment} \rangle)$$
$$\boxed{\triangleright} \quad (\lambda\text{self}.\ \langle \text{increment} \mapsto n := n + 1,\ \text{value} \mapsto n \rangle)$$

Wrappers can be used for *explaining* class inheritance. They can also be used explicitly in the language [39]. This has been found independently by Bracha and Cook [7].

We will now show how wrappers and classes interact using the example program of Fig. 2.2 on page 16. We will talk about the semantics of that program. We will be using the instance variables in the code, but we will not make the state of the objects explicit because we want to concentrate on the inheritance mechanism. We want to explain how *late binding* is realized by wrappers and especially the pseudo variables "self" and "super". The wrapper for the point class is

$$\begin{array}{l} \text{PointWrapper} = \lambda\text{self}.\ \lambda\text{super}. \\ \left\langle \begin{array}{rcl} \text{x} & \mapsto & \text{xComp} \\ \text{y} & \mapsto & \text{yComp} \\ \text{move(X,Y)} & \mapsto & \text{xComp} := \text{X+self.x}; \\ & & \text{yComp} := \text{Y+self.y} \\ \text{distFromOrg()} & \mapsto & \sqrt{(\text{self.x})^2 + (\text{self.y})^2} \\ \text{closerToOrg(point)} & \mapsto & \text{self.distFromOrg} < \text{point.distFromOrg} \end{array} \right\rangle \end{array}$$

In order to build a new class, we have to combine this wrapper with an existing class. The only initially existing class is the base class:

$$\text{BaseClass} \;=\; \lambda\text{self}.\langle\rangle$$

Now we define

$$\text{PointClass} \;=\; \text{PointWrapper} \boxed{\triangleright} \text{BaseClass} \;=\; \lambda\text{self}.$$

$$\left\langle \begin{array}{rcl} \text{x} & \mapsto & \text{xComp} \\ \text{y} & \mapsto & \text{yComp} \\ \text{move(X,Y)} & \mapsto & \text{xComp} := \text{X+self.x}; \\ & & \text{yComp} := \text{Y+self.y} \\ \text{distFromOrg()} & \mapsto & \sqrt{(\text{self.x})^2 + (\text{self.y})^2} \\ \text{closerToOrg(point)} & \mapsto & \text{self.distFromOrg} < \text{point.distFromOrg} \end{array} \right\rangle$$

A point object is created by application of the fixed-point operator to the point class: $p \;=\; \text{Y(PointClass)}$. As already mentioned, we do not consider the internal state here. In a formal framework it must, of course, appear somewhere.

Next in Fig. 2.2 comes the definition of the circle class as a subclass of the point class. Here is the corresponding wrapper that contains the added functionality:

$$\text{CircleWrapper} \;=\; \lambda\text{self}.\ \lambda\text{super}.$$

$$\left\langle \begin{array}{rcl} \text{r} & \mapsto & \text{radius} \\ \text{setR(r)} & \mapsto & \text{radius} := \text{r} \\ \text{distFromOrg()} & \mapsto & \max(0, \text{super.distFromOrg} - \text{self.r}) \end{array} \right\rangle$$

Now we define

$$\text{CircleClass} \;=\; \text{CircleWrapper} \boxed{\triangleright} \text{PointClass} \;=\; \lambda\text{self}.$$

$$\left\langle \begin{array}{rcl} \text{x} & \mapsto & \text{xComp} \\ \text{y} & \mapsto & \text{yComp} \\ \text{r} & \mapsto & \text{radius} \\ \text{setR(r)} & \mapsto & \text{radius} := \text{r} \\ \text{move(X,Y)} & \mapsto & \text{xComp} := \text{X+self.x}; \\ & & \text{yComp} := \text{Y+self.y} \\ \text{distFromOrg()} & \mapsto & \max(0, \sqrt{(\text{self.x})^2 + (\text{self.y})^2} - \text{self.r}) \\ \text{closerToOrg(point)} & \mapsto & \text{self.distFromOrg} < \text{point.distFromOrg} \end{array} \right\rangle$$

The β-reduction concerning "super" has already been performed. The notion of "self" in the new class is the same everywhere. The "closerToOrg"-method has not been mentioned in the circle wrapper. Note how, despite this fact, the "closerToOrg"-method now automatically calculates the correct distance from the origin of the circle object. A circle object is created by application of the fixed-point operator to the circle class: c = Y(CircleClass).

7.2 Keeping the Polymorphism

O'SMALL has class and wrapper definitions, local variable and instance variable declarations. The former are translated by let-declarations in **RFI** (cf. section 7.3). There are no references involved here. The latter pose a problem. Local variables and instance variables (let us simply call them variables from now on) can be subject to assignments in O'SMALL. A counter object must have an instance variable that can be updated to contain its current value. If we translate all variable declarations

$$\text{def } x := e \text{ in } e' \text{ ni}$$

by

$$\text{let } x = \text{ref } e \text{ in } e''$$

where e'' is the result of the substitution of all right-hand side occurrences of x in e' by $!x$, we are in trouble.[2] Since every variable is allocated in the store we lose most of our beloved polymorphism.

What can we do against that? We propose an intelligent translation function, i.e., a preprocessing step. We analyze the (finite) scopes of variables in our O'SMALL program. All declarations of variables x that get assigned new values by $x := e$ in their scope are translated the way we have shown above, i.e., by declaring references in **RFI** and dereferencing all occurrences of x on right-hand sides. All declarations of variables x

$$\text{def } x := e \text{ in } e' \text{ ni}$$

[2] e and e' have to be translated too. The formal translation function may be found in section 7.3.

that do not get assigned new values in their scope are translated simply by

let $x = e$ in e'

What can we gain with this procedure? Variables in O'SMALL are either primitive (booleans, integers,...) or objects. In the case of primitives, polymorphism is not needed. Primitive variables are typically changed by assignments. E.g., an instance variable n of a counter object might be incremented in one of the methods by $n := n + 1$. If the instance variables contain objects, then there are two possible ways of changing the state of the surrounding object:

- Suppose we have an instance variable x and a method f with a parameter y that contains the assignment $x := y$. This would mean replacing the object in the instance variable by a new one imported from outside.
- Suppose we have an instance variable x and a method f with a parameter y that is supposed to modify the contents of x. Suppose furthermore that the object in x understands a message "set" that changes its state. f could then change the state of the object in x by $x.\text{set}(y)$.

In the first case, the declaration of x must be translated by a reference to the store. The polymorphism would suffer at this point. In the second case, the declaration of x can be translated by a simple let-declaration (supposing there are no other assignments to x) and the polymorphism would remain intact. Our hope is that the programmer mainly uses the second alternative, i.e., changing objects by sending messages to them instead of replacing them altogether. If this programming style is applied then the changes of state would recursively propagate until the primitive level and assignments would be restricted to cases where polymorphism is not needed anyway.

One problem is that we do not know if this programming style is acceptable. Another problem is that the user of O'SMALL should not be forced to think in terms of the underlying intermediate language. If he programs in one of the two styles and sees the different reactions of the type inferencer he might start to ask questions. On the other hand, imperative type inference systems for languages like SML also have the problem that users do not understand them fully unless they get really involved with the subject.

p	::=	$\{k\}$ $[c]$	program
k	::=	class i inheritsFrom i d	direct class definition
		| wrapper i d	wrapper definition
		| class i i i	class definition from wrapper
d	::=	$[m]$	methods
		| def $[v]$ in $[m]$	instance variables and methods
c	::=	e	expression
	|	$i := e$	assignment
	|	if e then $[c]$ else $[c]$	conditional
	|	def $[v]$ in $[c]$	local variable
e	::=	b	expression (basic value)
	|	i	identifier
	|	$e.i(\{e\})$	message passing
	|	new e	object creation
	|	e * e	binary operator
v	::=	var $i := e$	variable declaration
m	::=	meth $i(\{i\})[c]$	method declaration

Figure 7.1: The abstract syntax of O'SMALL

7.3 Translation of O'SMALL

For the syntax description of O'SMALL, we use BNF-notation. Apart from the meta symbols ::= and | denoting definition and alternative, we use $\{e\}$ for zero or more repetitions of e and $[e]$ for one or more repetitions of e. Ordinary parentheses () are *not* metasymbols, they belong to O'SMALL. The concrete syntax of O'SMALL differs from the abstract syntax in order to facilitate parsing. We use `ni` as a closing parenthesis for `in`, semicolons in

sequences of complex expressions[3], commas between parameters, omission of parentheses for parameterless methods, and so forth. The abstract syntax is contained in Fig. 7.1. The *translation function* is denoted by angular brackets and has the type

$$\langle\!\langle \_ \rangle\!\rangle \ : \ \text{O'SMALL} \rightarrow \mathbf{RFI}$$

The use of the inheritance function $\boxed{\triangleright}$ implies the use of the concatenation operator (cf. definitions 7.1.1 and 7.1.2). Concatenation is not part of **RFI**. All concatenations can be performed by the compiler because classes are not first-class citizens in O'SMALL.

We assume that the translation is the identity on identifiers and that all name conflicts have been removed prior to translation. We are making extensive use of our abbreviating notation for sequences. We use the same variables for language constructs as in the abstract syntax. The only exception is the translation of 'def' where we use a generic g standing for either c or m. As already mentioned in section 6.2, we are using the results of an assignment analysis in the translation of 'def' and the right-hand side occurrences of the corresponding variables. Fig. 7.2 shows the translation function. The expression before the inheritance operator in the first clause is a wrapper. `self` and `super` are λ-abstracted. When methods are translated, they are curried. Correspondingly, messages (record selection) are translated into a curried version. By an underscore, we denote a distinct new variable for every c_i that is translated; it is a dummy variable that cannot be used.

O'SMALL programs can be type checked now, thanks to the translation function into **RFI** and the type inference for **RFI**. In the sequel, we will see how this is done in practice.

Fig. 7.3 contains a program in O'SMALL with explicit wrappers. The analysis of the local variable `a` permits us to translate it by a let-declaration. This gives us the necessary polymorphism for this example. The intermediate result of the translation is contained in Fig. 7.4. After the compiler has resolved the concatenations, we obtain the **RFI**-program of Fig. 7.5. The class A is declared as the application of the inheritance function to

[3]In many languages they are called 'statements', but here they also return a value, whence their name.

$$
\begin{array}{lcl}
\langle\!\langle \text{class } i_1 \text{ inheritsFrom } i_2\ d\ \ \overline{k}\ \overline{c} \rangle\!\rangle & & \\
\quad = \quad \text{let } i_1 = (\lambda \text{self}.\lambda \text{super}.\langle\!\langle d \rangle\!\rangle) \boxed{\triangleright}\ i_2 \text{ in } \langle\!\langle \overline{k}\ \overline{c} \rangle\!\rangle & & \\
\langle\!\langle \text{wrapper } i\ d\ \ \overline{k}\ \overline{c} \rangle\!\rangle & = & \text{let } i_1 = \lambda \text{self}.\lambda \text{super}.\langle\!\langle d \rangle\!\rangle \text{ in } \langle\!\langle \overline{k}\ \overline{c} \rangle\!\rangle \\
\langle\!\langle \text{class } i_1\ i_2\ i_3\ \ \overline{k}\ \overline{c} \rangle\!\rangle & = & \text{let } i_1 = i_2 \boxed{\triangleright}\ i_3 \text{ in } \langle\!\langle \overline{k}\ \overline{c} \rangle\!\rangle \\
\langle\!\langle \text{meth } i(\overline{i})\overline{c} \rangle\!\rangle & = & i \mapsto \lambda \overline{i}.\langle\!\langle \overline{c} \rangle\!\rangle \\
\langle\!\langle m\ \overline{m} \rangle\!\rangle & = & \langle \langle\!\langle \overline{m} \rangle\!\rangle \rangle \oplus \langle \langle\!\langle m \rangle\!\rangle \rangle \\
\langle\!\langle \text{def var } i := e \text{ in } \overline{g} \rangle\!\rangle & = & \begin{cases} \text{let } i = \text{ref } \langle\!\langle e \rangle\!\rangle \text{ in } \langle\!\langle \overline{g} \rangle\!\rangle & \text{if } i \text{ occurs on lhs} \\ \text{let } i = \langle\!\langle e \rangle\!\rangle \text{ in } \langle\!\langle \overline{g} \rangle\!\rangle & \text{otherwise} \end{cases} \\
\langle\!\langle \text{def } v\ \overline{v} \text{ in } \overline{g} \rangle\!\rangle & = & \langle\!\langle \text{def } v \text{ in def } \overline{v} \text{ in } \overline{g} \rangle\!\rangle \\
\langle\!\langle i := e \rangle\!\rangle & = & i := \langle\!\langle e \rangle\!\rangle \\
\langle\!\langle \text{if } e \text{ then } \overline{c} \text{ else } \overline{c'} \rangle\!\rangle & = & \text{if } \langle\!\langle e \rangle\!\rangle \text{ then } \langle\!\langle \overline{c} \rangle\!\rangle \text{ else } \langle\!\langle \overline{c'} \rangle\!\rangle \\
\langle\!\langle c\ \overline{c} \rangle\!\rangle & = & \text{let } \_ = \langle\!\langle c \rangle\!\rangle \text{ in } \langle\!\langle \overline{c} \rangle\!\rangle \\
\langle\!\langle b \rangle\!\rangle & = & b \\
\langle\!\langle i \rangle\!\rangle & = & \begin{cases} !i & \text{if } i \text{ was introduced by 'ref'} \\ i & \text{otherwise} \end{cases} \\
\langle\!\langle e.i(\overline{e}) \rangle\!\rangle & = & \langle\!\langle e \rangle\!\rangle.i\langle\!\langle \overline{e} \rangle\!\rangle \\
\langle\!\langle \text{new } e \rangle\!\rangle & = & \mathsf{Y}\langle\!\langle e \rangle\!\rangle \\
\langle\!\langle e\ *\ e' \rangle\!\rangle & = & \langle\!\langle e \rangle\!\rangle * \langle\!\langle e' \rangle\!\rangle
\end{array}
$$

Figure 7.2: The translation function

```
wrapper AWrap
   meth id(x) x

class A AWrap Base

def var a := new A
in
   a.id(3);
   a.id(true)
ni
```

Figure 7.3: The source program in O'SMALL

$$
\begin{array}{l}
\text{let AWrap} = \lambda\text{self}.\lambda\text{super}.\ \langle \text{id} \mapsto \lambda x.x \rangle \text{ in} \\
\text{let A} = \lambda\text{self}.\ (\text{AWrap self (Base self)}) \oplus (\text{Base self}) \text{ in} \\
\text{let a} = \text{Y A in} \\
\text{let \_} = (\text{a.id 3}) \text{ in} \\
\text{a.id true}
\end{array}
$$

Figure 7.4: The intermediate program

the wrapper and the base class. Sequences of expressions are translated by let-declarations with dummy variables. Since the translation has produced a target program that is purely functional, the effects are always empty. Therefore, they are omitted in the sequel. The types inferred for the target program are:

$$
\begin{array}{rcl}
\text{x} & : & \alpha \\
\text{a} & : & \langle \text{id} \mapsto \beta \to \beta \rangle \\
\text{A} & : & \gamma \to \langle \text{id} \mapsto \delta \to \delta \rangle \\
\text{AWrap} & : & \epsilon \to \zeta \to \langle \text{id} \mapsto \alpha \to \alpha \rangle
\end{array}
$$

Types for dummy variables are left out and, thus, we get a nice correspondence to the variables of the original program. In a type inference system

$$\text{AWrap} = \lambda\text{self}.\lambda\text{super}.\ \langle \text{id} \mapsto \lambda x.x \rangle \text{ in}$$

$$\begin{array}{l}\text{let A} = \lambda\text{self}.\ \langle \text{id} \mapsto \lambda x.x \rangle \text{ in}\\ \text{let a} = \text{Y A in}\\ \text{let \_} = (\text{a.id 3}) \text{ in}\\ \text{a.id true}\end{array}$$

Figure 7.5: The target program in **RFI**

for end users, the type information for classes and wrappers should come in a digested form: one could imagine presenting it as input and output requirements in the case of wrappers, and as the quality of being abstract or not in the case of classes. This information is easily obtained from the types.

7.4 Assessment

We now show the achievements and the limitations of our type inference system. Depending on the issue, we choose example programs in O'SMALL or **RFI**.

7.4.1 Recursive Types

Fig. 7.6 is a natural example[4] which demonstrates the need for recursive types. The objects of class `Pair` together with the relation `leq` define a preorder. The objects of class `OrderedPair` together with the relation `leq` and the equality `eq` define a partial order because the equality has been redefined and, now, `leq` becomes antisymmetric. The type of the object `p` is recursive:

[4]It is a modification of an example in [39].

```
class Pair inheritsFrom Base
def var xComp:=0
    var yComp:=0
in meth set(a,b) xComp := a; yComp := b
   meth x()      xComp
   meth y()      yComp
   meth leq(p)   (self.x + self.y)  <=  (p.x + p.y)
   meth eq(p)    self.x = p.x  and  self.y = p.y
ni

class OrderedPair inheritsFrom Pair
   meth eq(e)    self.leq(e)  and  e.leq(self)

def var p  := new OrderedPair
in
   p.set(7,3)
ni
```

Figure 7.6: O'SMALL program with recursive types

$$\mu t \;.\; \left\langle \begin{array}{rcl} \texttt{eq} & : & \langle \zeta \mid \texttt{leq} : t \to \text{bool}, \texttt{x} : \text{num}, \texttt{y} : \text{num} \rangle \to \text{bool} \\ \texttt{leq} & : & \langle \zeta \mid \texttt{leq} : t \to \text{bool}, \texttt{x} : \text{num}, \texttt{y} : \text{num} \rangle \to \text{bool} \\ \texttt{set} & : & \text{num} \to \text{num} \to \text{unit} \\ \texttt{x} & : & \text{num} \\ \texttt{y} & : & \text{num} \end{array} \right\rangle$$

The recursion in this type stems from the redefinition of the equality in the class `OrderedPair`. The argument `e` of `self.leq` must understand a message `leq` where the same `self` is an argument. The type check algorithm proceeds in the following way when it checks the method `eq` in the class `OrderedPair`. The pseudo-variable `self` gets the message `leq(e)` and, thus, we obtain the types $\texttt{e} : \alpha$ and $\texttt{self} : \langle \epsilon \mid \texttt{leq} : \alpha \to \beta \rangle$. Now, `e` gets the message `leq(self)` and, thus, $\texttt{e} : \langle \eta \mid \texttt{leq} : \gamma \to \delta \rangle$. Variables ϵ and η are row variables of the appropriate sorts. Now, the two types of `e` must be

unified: `e` : $\langle \eta \mid \texttt{leq} : \gamma \to \delta \rangle$ and `self` : $\langle \epsilon \mid \texttt{leq} : \langle \eta \mid \texttt{leq} : \gamma \to \delta \rangle \to \beta \rangle$. Also, γ must be unified with the type of `self`. The type of `self` already contains γ, and we get a recursive type.

When a class declaration is checked, there is no test of whether the type of `self` can be unified with the type of the record of methods actually being provided. This test is performed only if an object of this class is created. It is thus possible in a method to have messages to `self` that are not defined in this or any ancestor class. This results in an *abstract class* [30]. The type checker accepts abstract classes, but rejects the creation of their objects (see also section 7.4.3).

7.4.2 Imperative Features

O'SMALL is an object-oriented, not a functional language. Therefore, we found it inappropriate to reveal details on imperative vs. purely functional programming to the user. In O'SMALL, every variable is potentially imperative, i.e. we can assign a new value to every variable defined by `def`. Imperative features and let-polymorphism mostly exclude each other. The intelligent translation function avoids this problem in many cases. In Fig. 7.3 on page 99, we have the definition of the local variable `a`. In a simple translation function, the definition would be translated by `let a = ref ...`. The analysis allows us to translate it by `let a = ...` because there are no assignments to `a` in its scope. Fig. 7.7 is similar to the program of Fig. 7.3 but, now, there is an assignment. Fig. 7.8 contains the translation of Fig. 7.7 to **RFI**. The local variable must be introduced as a reference. The program is refused by the type checker. References to polymorphic functions are impossible.

There are limitations to our type system with respect to records that are only indirectly related to imperative features. The O'SMALL-program of Fig. 7.9, which does not produce a run time error, is refused by the type checker. The types of `a` and `b` cannot be unified because `b` has a method `n` that `a` does not have. The row variable mechanism, which works for arguments of methods, does not work for objects that are "simply there" because these objects have closed record types, and the different possible types are unified in our algorithm. It is impossible to give open record types to objects because then, record selection of absent fields would be accepted

```
class A inheritsFrom Base
   meth id(x) x

def var a := new A
in
   a.id(3);
   a := new A;
   a.id(true)
ni
```

Figure 7.7: An O'SMALL-program with an assignment

$$
\begin{array}{l}
\text{let A} = \lambda\text{self. } \langle \text{id} \mapsto \lambda x.x \rangle \text{ in} \\
\text{let a} = \text{ref (Y A) in} \\
\text{let \_} = \text{(a.id 3) in} \\
\text{let \_} = \text{(a} := \text{Y A) in} \\
\text{a.id true}
\end{array}
$$

Figure 7.8: The target program in **RFI**

by the type checker. Chapter 8 contains a short discussion of why there is no simple way out of this dilemma.

Again, in the general case, one is not sure if the assignment has taken place, but if the only message sent to a is m without arguments, this cannot go wrong. In this example, objects of the subclass B are of a subtype of the types of objects of the superclass A – if we define a subtype relation appropriately. The problem is that we do not know of any simple way of integrating an explicit subtype notion into the current framework. The work of Stansifer [89] may be valuable in this direction of research.

```
class A inheritsFrom Base
   meth m() 0

class B inheritsFrom A
   meth n() 0

def var a := new A
    var b := new B
in
   a := b;
   a.m
ni
```

Figure 7.9: O'SMALL program with closed record types

7.4.3 Abstract Classes

The recognition of *abstract classes* is possible because classes are functions from records to records. A class has a type $\sigma \to \tau$, where σ and τ may be different. The type σ on the left hand side of the arrow (let us call it the *expected type*) is the type of self reference. The type τ on the right hand side of the arrow (let us call it the *provided type*) is the type of the methods defined in the class. When a class is declared, the two types may be completely incompatible, and the type checker may still accept the class. When an object of the class is created, a fixed point operation takes place, and the two types are unified. Now, common components must be compatible. The *expected type* is always open, whereas the provided type is always closed. If the provided type has a component that is not in the expected type, no problem occurs. However, if the expected type has a component that is not in the provided type, there is a type clash resulting form the closedness of the provided type. The type checker has thus noticed that the class of the object just created is abstract. If a subclass defines the missing components appropriately, objects of the subclass can be created. Summing up, it may be said that abstract classes are recognized "lazily", i.e. the creation of instances of abstract classes is refused. In Fig. 7.10, class

```
class A inheritsFrom Base
   meth f(n)
      if  n=1  then  1  else  self.g(n-1) + 1  fi

class B inheritsFrom A
   meth g(n)
      if  n=1  then  1  else  self.f(n-1) + 1  fi

def var b := new B
in
   b.f(9)
ni
```

Figure 7.10: An abstract class

A is abstract. A has a subclass B that is not abstract. B has an instance b. The inferred types are as follows:

$$\begin{array}{lrcl} \texttt{A} & : & \langle \alpha \mid \texttt{g} : \text{int} \to \text{int} \rangle & \to \quad \langle \texttt{f} : \text{int} \to \text{int} \rangle \\ \texttt{B} & : & \langle \beta \mid \texttt{f} : \text{int} \to \text{int}, \texttt{g} : \text{int} \to \text{int} \rangle & \to \quad \langle \texttt{f} : \text{int} \to \text{int}, \texttt{g} : \text{int} \to \text{int} \rangle \\ \texttt{b} & : & \langle \texttt{f} : \text{int} \to \text{int}, \texttt{g} : \text{int} \to \text{int} \rangle & \end{array}$$

We have seen that the type inference can handle many typed object-oriented examples. It delivers valuable documentation that may be passed on to the end-user in a digested form. Yet, there are some limitations that forbid to assign objects of a "subtype" to a variable containing an object of a "super type".

7.4.4 Assessing O'SMALL

O'SMALL was developed for demonstrating the use of wrapper semantics. It has all the characteristic features of object-oriented languages, yet, after the completion of the type inference system, some extensions impose themselves. The often cited example of a stack is well-suited for showing some

```
class StackElem
def var value  := 0
    var bottom := true
    var next   := 0
in
   meth getValue()   value
   meth setValue(v)  value := v
   meth getBottom()  bottom
   meth setBottom(b) bottom := b
   meth getNext()    next
   meth setNext(n)   next := n
ni

class Stack
def var head := new StackElem
in
   meth push(elem)
            def var s := new StackElem
            in
               s.setValue(elem);
               s.setBottom(false);
               s.setNext(head);
               head := s
            ni
   meth top()
           if self.empty then output(99999); head.getValue
           else def var out := head.getValue
                in head := head.getNext; out ni
           fi
   meth empty() head.getBottom
ni
```

Figure 7.11: A stack example

deficiencies. Fig. 7.11 shows two class definitions defining stack elements and stacks. We will have to explain the clumsiness of this program.

Usually, *one* class is sufficient for stacks. The class `StackElem` is no more than three instance variables made accessible to the outside. This could be avoided by integrating these instance variables into class `Stack`. However, in method `push` we use `new StackElem`. If the three instance variables were integrated and we only had one class, we would not be able to say `new Stack` because of the scoping of O'SMALL: the class name of the current class is unknown when the class is being declared. A solution that is briefly discussed in [38] is the introduction of a new pseudo variable `current` that denotes just the current class. This pseudo variable would have the same visibility as the pseudo variables `self` and `super`, i.e. it would be visible in the method part, not in the instance variable part.[5]

Another problem is that this program does not pass the type checker. This has to do with the initialization of instance variables. In class `StackElem`, the instance variable `next` that contains the "link" to the next stack element is initialized with `0`. Therefore, the type inferencer would infer type num for `next`. One solution is the introduction of `nil` into the language. This has been done by Palsberg and Schwartzbach [73] for a language similar to O'SMALL. The drawback of this solution are messages sent to `nil`. Our policy in O'SMALL consists in initializing everything. This also makes type declarations superfluous. In the example, we would like to initialize `next` with `self`. For the type inferencer, this is a good solution. The visibility of `self` would have to be extended to the instance variable part – that is all.

Extending the visibility of `super` to the instance variable part probably does not make much sense. Extending the visibility of `current` to the instance variable part is problematic for the semantics: When an object is created, all its instance variables are initialized. If one of the instance variables were initialized to `new current`, this would lead to non-termination.

[5] Of course, there are further extensions to O'SMALL, like introducing a shorthand for "visible" instance variables or a mechanism of hiding some of the methods. These extensions would alienate the character of O'SMALL. They should be part of a bigger language that bases itself on some kernel language. The output of `99999` is an attempt to signal an error. In a bigger language one would need an exception raising and handling mechanism like in SML.

A solution, which has been adopted in ABCL [109], consists in delaying object creation to the point where the object receives its first message. This solution inherits the problems of mixing *lazy evaluation* with state: it is almost impossible for the human to know what the state will be at object creation.

Suppose that we have made the extensions discussed above. Then the program of Fig. 7.11 would pass the (extended) type checker. However, we would still not get enough polymorphism. The instance variable `value` of class `StackElem` is initialized with `0`. In the untyped version of O'SMALL this is no problem. We simply push any kind of elements on the stack and do not care about this initial value which is never accessed anyhow because it is in the "bottom of the stack". In the typed version, the initial value determines the type. In order to obtain polymorphic stacks we must be able to initialize them with different elements. Fortunately, the extension of the O'SMALL semantics and the type inference system to parameters of classes is straightforward. Examples of classes with parameters can be found in [38].

8. Conclusion

We have presented a polymorphic type system for a class based object-oriented language with state. Part of the algorithm has been proven correct. The parts with 'let' and imperative features have not been proven correct relying on previous work. O'SMALL existed before the idea of inferring types for it was born. We have, thus, designed the type inferencer after the language and not vice versa. Previous work by Rémy and Wand offered a solution for object-oriented programming languages without state. Wand's solution has no principal types but principal sets of types. We avoid this complication by simplifying the underlying record language. This simplification is possible because of the module-like characteristics of O'SMALL-classes. The translation function is enriched by an analysis of imperative constructs in order to use an existing type inference method for the λ-calculus with imperative features.

Since type inference is performed on **RFI** rather than on O'SMALL itself, one might suspect that information of the source language would be lost in the process. Surprisingly enough, this does not have to be a consequence, as we have seen for abstract classes: they are recognized. The type inference system provides a good documentation of otherwise untyped programs and helps finding errors even in small applications. It is of immediate practical use. Each class has to be checked at most once because we infer principal types. This is useful in incremental systems (rapid prototyping) where type inference can be part of the compiler. In commercial systems, where the code of classes should sometimes not be exposed, the type information of our system could be a much more reliable documentation than mere words.

Although O'SMALL is a relatively parsimonious language, the task of verifying its type inference system is complex. We have broken up the problem by first translating O'SMALL into a simpler language and then creating several sublanguages that could be treated separately. Besides ordinary let-polymorphism, our type system features open record types, recursive types, and imperative types. We have used order-sorted logic for their formalization. The type inference algorithm is divided into a constraint creation and a constraint resolution phase. The idea of constraints existed before [12], but the use of order-sorted logic in this context is new.

8.1 Alternatives

Let us discuss some of the decisions and their alternatives. We have chosen natural semantics for the imperative part in chapter 6, thus, changing from the stepwise term rewriting style of chapter 4 although there is an approach by Wright and Felleisen [106] with rewriting. In their approach, the store is part of the term and not an extra component. In order to make the semantics deterministic in the presence of side effects, they have to introduce so-called evaluation contexts. These contexts appear in the rewrite rules and seem to complicate proofs by structural induction. Therefore, we decided against this method despite its initial appeal of uniformness with previous chapters.

The limitations of Fig. 7.9 on page 104 suggest that one should introduce the notion of subtypes and take a least upper bound as a solution there: the type of a is $\langle \mathrm{m} : \mathrm{int} \rangle$, and the type of b is $\langle \mathrm{m} : \mathrm{int}, \mathrm{n} : \mathrm{int} \rangle$. With an appropriate subtyping order [15], the least upper bound of the two types is $\langle \mathrm{m} : \mathrm{int} \rangle$. This is shown in Fig. 8.1. In this example, the least upper

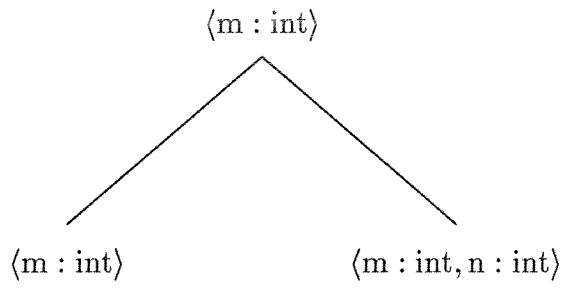


Figure 8.1: Least upper bound

bound is a good solution. However, in the general case, there exist many incomparable upper bounds and no least upper bound. Consider the type expressions $\langle \mathrm{a} : \alpha, \mathrm{b} : \alpha \rangle$ and $\langle \mathrm{a} : \mathrm{int}, \mathrm{b} : \mathrm{bool} \rangle$. Fig. 8.2 shows two of their upper bounds that are incomparable. Arbitrarily complicated examples can be constructed, and it is not clear which upper bound to take in the

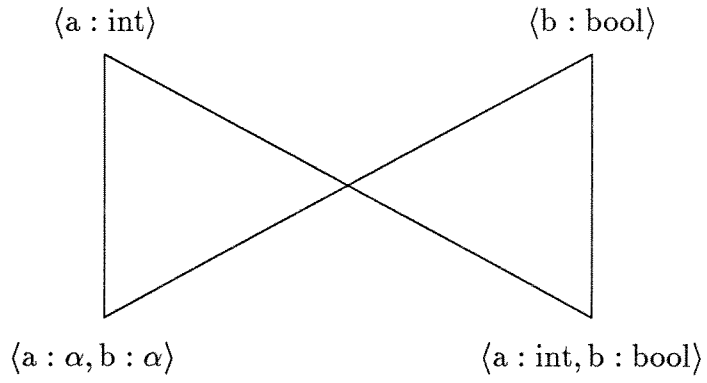


Figure 8.2: Incomparable upper bounds

general case. Even if an algorithm chose any of the upper bounds by a fixed strategy, it may be difficult to find out why one program is accepted and the other not. A way to compute a unique upper bound can be found by introducing a least upper bound operator $\sqcup$ into the type language. The *least upper bound* of the two type expressions is then

$$\langle a : \alpha, b : \alpha \rangle \sqcup \langle a : \text{int}, b : \text{bool} \rangle \quad = \quad \langle a : \alpha \sqcup \text{int}, b : \alpha \sqcup \text{bool} \rangle$$

We must be prepared to see complicated types if we use this method.

The design decisions of O'SMALL have been taken before the type inference project was envisaged. Not to include classes as first-class citizens made the semantics of O'SMALL simpler. It also simplified the type inference since we can infer principal types. Classes in O'SMALL are more restricted than in SMALLTALK. While SMALLTALK is an exploratory language, O'SMALL goes into the direction of a software engineering language. O'SMALL classes are declared at the beginning of a program and cannot be changed dynamically. A look at the scoping rules reveals that the relations "a class knows another class" or "a class may contain an instance variable of another class" are acyclic. This restriction facilitates reasoning about classes and makes them similar to modules in certain languages.

Classes cannot contain instance variables of each other in a (mutually)

recursive way. Allowing for this would imply a more complicated semantics for object creation in order to avoid non-termination. A solution, which has been adopted in ABCL [109], consists in delaying the creation of an object in an instance variable to the point where the object receives its first message. Not only the semantics, but also the type inference system, would become more complicated. We would have to consider 'letrec'- instead of 'let'-definitions.

A general pointer (reference) concept is absent in O'SMALL. This allows for the simple analysis in the translation phase. It remains to be seen if this restriction is acceptable for O'SMALL as the basis for a general purpose programming language. While a general pointer concept could be easily added to O'SMALL, it would make the assignment analysis during the translation impossible.

8.2 Related Work

The following list is by no means exhaustive and represents some of the work the author has considered during the design and construction of this type inference system. Most directly linked to this work, are type systems that deal with records, functions, or imperative features. We will briefly explain why subtypes have not been used. Type systems for SMALLTALK will be considered, and, finally, the O'SMALL-semantics of this work will be compared to previous ones.

Records and Functions

Records are the main data structure in object-oriented languages. In previous systems with polymorphic type inference (e.g. SML) it was impossible to write a function $\lambda x.(x.a)$ and give it a type such that it could be applied to all records that possess an a-component. A very general solution of this problem consists in defining a structural subtyping notion [11, 15, 66, 89]. Another central feature of object-oriented languages – at least as we understand it – is the internal state of objects. However, mixing a subtyping notion and imperative features can be problematic [13]. A thorough discussion of the subtyping problems in object-oriented programming languages

can be found in [9]. There, Bruce considers a purely functional language with type declarations. Although subtyping gives a certain flexibility to the type inference in object-oriented programming languages it is not exactly what one wants for subclassing [10].

A more specialized approach that avoids the problems with subtyping was first proposed by Wand [99]. He introduced the concept of row variables. The function $\lambda x.(x.a)$ has the type $\langle \alpha \mid a : \beta \rangle \to \beta$, where α is the row variable indicating that the value that gets eventually bound to x may have further labels. Wand's language has the following syntax:[1]

e	::=	x	variable
	|	$\langle \overline{a} \mapsto \overline{e} \rangle$	record
	|	$e \oplus e$	concatenation
	|	$e.a$	selection

However, principal types cannot be inferred for that language. The problem is that in the selection of a concatenation like $(x \oplus y).a$, we do not know whether the a-component must be present in x or in y. The solution proposed in [102] consists of inferring principal sets of types. These sets are finite but the combinatory explosion is problematic.

Rémy [79] introduced the notion of *fields*. Fields may be either instantiated to "present" or to "absent". Absent fields may still have types. We had problems of getting a good intuition for absent fields with a fixed type. In [81], Rémy presents the language

e	::=	x	variable
	|	$\langle \overline{a} \mapsto \overline{e} \rangle$	record
	|	$\langle a \mapsto e \rangle \oplus e$	adjunction
	|	$e.a$	selection

and shows that "record concatenation comes for free once record adjunction is provided". This result indicates that even extremely restricted languages can be powerful enough. The language we have examined in [40] has the same syntax. If we extend this language by λ-abstraction and function application, we can formulate the counterexample for principal types [100]. The symbol + stands for integer addition.

$$\lambda f.\lambda x.f(\langle a \mapsto 3 \rangle \oplus x) + f(\langle a \mapsto 3 \rangle \oplus \langle\rangle)$$

[1] We always omit λ-abstraction and application.

The adjunction of $\langle a \mapsto 3 \rangle$ to x must yield a term that has just an a-field. Thus, x must either be the empty record or a record with just an a-field. The types of this term are

$$(\langle a : \text{int} \rangle \to \text{int}) \to \langle\rangle \to \text{int}$$

and

$$(\langle a : \text{int} \rangle \to \text{int}) \to \langle a : \tau \rangle \to \text{int}$$

for any type τ. In the present framework, there is no type scheme that can generate just these types.[2]

When classes are top-level like in O'SMALL, we obtain the following calculus:

e	::=	x	variable
	$\mid$	$e.a$	selection
	$\mid$	r	simple record
r	::=	$\langle \overline{a} \mapsto \overline{e} \rangle$	record
	$\mid$	$r \oplus r$	concatenation

Since all labels are known at compile time, there are only concatenations of plain records. Since the labels are known at compile time, we can also let the compiler perform the concatenations. Therefore, we obtain our language $\mathbf{R}$, which has neither adjunction nor concatenation:

e	::=	x	variable
	$\mid$	$e.a$	selection
	$\mid$	$\langle \overline{a} \mapsto \overline{e} \rangle$	record

It is somehow amazing that this language should suffice for a full object-oriented language. The advantage of this simple language is the ease of formalization because we can get rid of fields and field variables. We obtain an algorithm that immediately and naturally deals with infinite label sets.

Harper and Pierce [36] investigated type inference for a record calculus with so-called *symmetric concatenation*, i.e. records must not have overlapping fields when they are concatenated. Although this system may be

[2] We considered a restricted language that contained fieldwise record adjunction instead of concatenation and thought that this would give us principal types [40]. The bug in [40] is in the adjunction axiom of the typing relation: the term $\rho' \cdot a[\alpha]$ is not well-sorted in general but well-sortedness is assumed throughout the paper.

useful for discovering inadvertent clashes of labels it is not well-suited for O'SMALL-style class inheritance where overwriting existing methods happens quite often. Rémy [82] has studied sorted algebras and equational theories in general, however, the study neither includes recursive types nor is it order-sorted as our's.

Ohori and Buneman [72] present a solution for inferring types for parametric classes. Their conditional type-schemes are similar to our open record types and recursive types. One major difference is that the type of the implementation, the instance variables, appear in the types of the methods. Although this type can be hidden by existential quantification it is still there. Therefore, an object has some hidden type. It is not simply a collection of methods like in our approach. Their approach does not include late binding.

Jategaonkar and Mitchell [49] present an interesting language with extended pattern matching. In their language, row variables also occur in the expressions. Using extended pattern matching they can express symmetric record concatenation. Since symmetric record concatenation does not allow overwriting of labels, the expressiveness of their language is in this respect comparable to that of our record calculus. By restricting record concatenation in this way they arrive at principal types for a type system with row variables in the same way as we. Their row variables are annotated with finite sets of labels. In our system this mechanism (and more) is contained in the sorts of the variables. One difference is that our system contains recursive types and theirs does not. Another difference lies in the unification algorithm using row variables. The unification algorithm of our system can be formulated naturally whereas their algorithm contains so-called double substitutions.

On the part of type inference for the λ-calculus, our system is fairly standard and relies on [43, 63, 12], although we use recursive types [20, 16]. Recursive types are also used by Ohori [70, 72].

Imperative Features

The imperative features resulting from the translation of O'SMALL programs are the same as those in the early ML language [32]. Variables are automatically dereferenced, too. It is noted that top-level letrefs must

be monomorphic whereas polymorphic *own-variables*[3] are useful. No type inference rule for letref was presented in [32]. In our language, we have nothing but polymorphic own variables or, stated differently, non-local assignments are impossible. Therefore, we are able to determine statically, whether there are assignments to a variable. If there are no assignments we can translate a variable definition by the definition of a local variable; we avoid the definition of a reference. This allows for more polymorphism.

The combination of state and polymorphic type checking is an active research field, where new approaches keep coming out [24, 56, 83, 94, 93, 96, 107, 108]. After considering the approach of Talpin and Jouvelot [94] because it is among the most powerful ones, we noticed that the additional complexity introduced by effects came on top of our extensible record types and recursive types. In a practical system such as ours, the importance of simplicity and readability of types cannot be underestimated. The additional power of Talpin and Jouvelot did not seem to make up for the additional complexity. This holds for **RFI** programs that are the targets of O'SMALL translations. It is not intended to be a general judgement of the approach.

The discipline by Wright [108] is conceptually even simpler than Tofte and has the advantage of not introducing any new types or effects. It forbids generalization of type variables of expressions that are applications. This comprises the application `ref e` and is, therefore, safe. Although Wright has empirically tested his discipline on a huge number of programs it does not work for our application. Typically, we have declarations like `let a = ref(FIX A)` or `let a = FIX A`, where `A` is a class. In the first case, our system cannot generalize the type variables. In the second case, that results from a clever translation from O'SMALL, we can generalize some type variables. In [108], the generalization would be impossible because `FIX A` is an application. It is also impossible to perform the η-expansion proposed by Wright because `FIX A` is a record, not a function.

In **RFI** programs that are translations from O'SMALL programs reference types do not reach the surface, i.e., they will not be reported as the type of an expression to the user because of automatic dereferencing in O'SMALL. Tofte [96] considers the two types $(\forall t.t \rightarrow t)$ref and

[3]To an own-variable, only local assignments are possible.

(int → int)ref. References to the latter can hold more functions making assignments easier whereas references to the former can hold functions that can be used in more situations. Therefore, there is no natural candidate for a principal type of the expression ref($\lambda x.x$).

In the context of data base languages, Ohori [71] discusses type checking for an imperative language. This language is translated by monads [68, 98] into pure λ-calculus. The extension of let-polymorphism to monads is an open problem.

Object-Oriented Languages

An approach that incorporates imperative features and is specialized for object-oriented programming languages comes from Palsberg and Schwartzbach [73]. Their language is almost identical to O'SMALL – including the concrete syntax! Their approach could be called (type inference by) abstract interpretation. Their notion of type differs completely from ours. Their types are finite sets of classes whereas here they are infinite. We determine the most general type for an expression whereas they compute a type that an expression has in a fixed program *including all its uses.* In other words, we infer the type of a class once and for all. It is valid for all uses. They have to restart their abstract interpretation from scratch each time a new line is added to the program. The information gained by abstract interpretation is finer than the one gained by type inference. It can accept more programs and is better suited for program optimization. The information gained by type inference is coarser because the notion of type must be simpler in order to be computable. The information of a most general type for an expression is a better documentation for the programmer. Furthermore, the most general type is only computed once. Therefore, type inference has an efficiency advantage when it comes to evolving systems.[4] Abstract interpretation and type inferencers are not in direct competition and the following scenario of their combined use can be imagined: a type inferencer is run on each class separately and only once. It is used during the program development phase. If there are parts in a complete software product that are refused by the type inferencer, abstract

[4] Although theoretical results on type inference predict a very bad worst case behavior [60, 53].

interpretation can guarantee the absence of type errors with a final check on the closed program. A prerequisite for the last step is that the user interaction is only via controlled input, i.e. for example by strings. Palsberg and Schwartzbach also investigated an O'SMALL like language with type declarations [74, 75].

SMALLTALK can be seen as a superset of O'SMALL where type inference is harder. Attempts to perform type inference for SMALLTALK started with Suzuki [91] and had the goal of efficient implementations [92]. Borning and Ingalls [6] presented a type inference system that relies on type declarations. Another approach to type checking SMALLTALK was made by Graver and Johnson [52, 33, 34]. The types of variables must be declared. All these systems have in common that not the whole language can be checked statically since SMALLTALK contains some low level elements.

Previous Semantics of O'SMALL

Fig. 8.3 compares the methodology of this work with that of [38] and [39]. Comparing this figure to Fig. 2.1, we regard the record languages as being without type inference because, in previous semantics, there has not been static type checking. Looking at the denotational semantics in [38, 39] from a purely syntactical point of view, we could say that the meta language used for the description there is **RF**. The denotational semantics can then be viewed as a translation from O'SMALL to **RF**. This translation has to contain all details about the management of the store. In this work, we have used a translation from O'SMALL to **RFI**. The target language has the concept of a store. The translation is thus freed from store management, and becomes much simpler.

8.3 Implementation

The algorithms described in this work have been implemented in CAML [57, 104]. The implementation is as close as possible to the description. The declarative nature of the formalization results in efficiency problems. There are several sources of inefficiency:

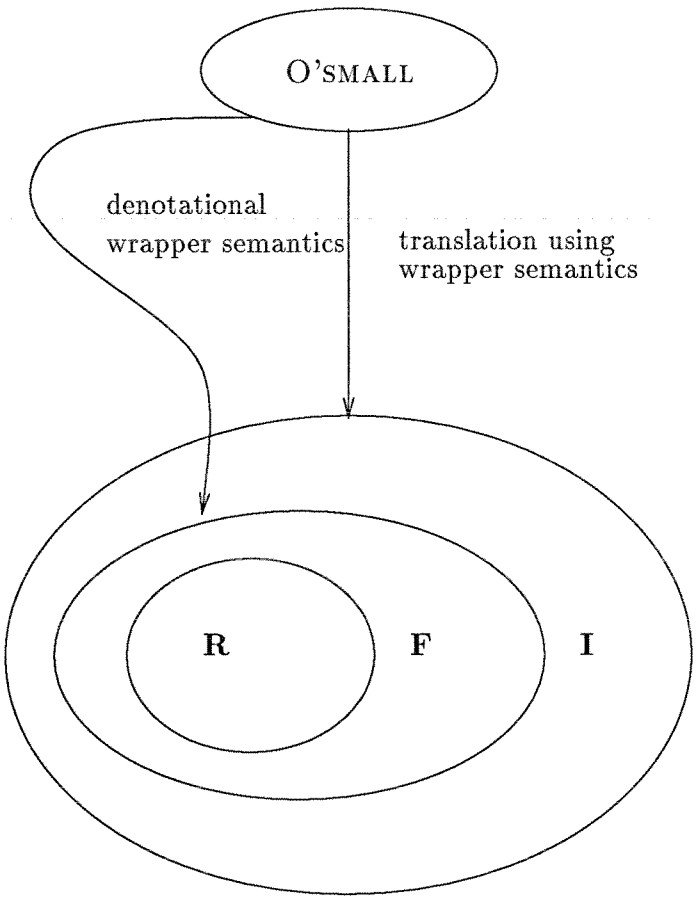


Figure 8.3: Methodologies compared

- The constraint extraction rules of definitions 5.5.5 and 6.3.2 perform generalization and instantiation for each right-hand side occurrence of variables.
- Instead of looking up variables in the constraint in normal form, it might be better to apply the substitutions directly to the type environment.
- Constraint resolution (rules (4.5) through (4.17)) introduces many unnecessary equations. E.g. when sorts are adapted, it would suffice to adapt the sort of variables instead of creating new ones and introducing new equations.
- There is no description of efficiently handling equations with isolated variables.

The speed tuning of the algorithm is necessary to make it applicable to large programs. However, we do not see any scientific interest here because the Damas-Milner algorithm exists already and any speed tuning would have to go in that direction.

8.4 The future

The system has not been tested on real size applications. While some people have programming styles that pose no problems, the limitations show that the common practice of assigning objects of a subclass to variables originally containing objects of a superclass is refused. Class libraries containing such assignments could still be checked in parts since the type inferencer can be "switched off". The correctness of the combination of type correct parts would have to be proved by other means in this case.

Turing and Church have shown that relatively few elements are necessary to create a universal programming language. While this is good news, we also know that most interesting properties of programs are undecidable. A designer of programming languages looks for new languages that are, on the one hand, powerful enough for the intended purpose and, on the other hand, restricted enough to encourage a good programming style and to make certain properties decidable [105]. The future will show if O'SMALL itself is a successful (prototypical) design in this sense and if this type inference system, which provides safety but places restrictions on the programming style, is acceptable.

> Quod si deficiant vires, audacia certe
> Laus erit: in magnis et voluisse sat est.
>
> Propertius Sextus (48–16 B.C.)

A. Appendix

A.1 Basic Definitions

Definition A.1.1 Let M be a set. A subset $H \subset M \times M$ defines a *preorder* (we often write $a \leq b$ for $(a, b) \in H$), if $\forall a, b, c \in M$:

$$\begin{aligned} a &\leq a \\ a \leq b \ \wedge \ b \leq c \ &\Rightarrow \ a \leq c \end{aligned}$$

If, in addition, $a \leq b \ \wedge \ b \leq a \Rightarrow a = b$ holds, $(M, \leq)$ is called a *partial order*. If, in a partial order, $\forall a, b \in M$:

$$a \leq b \ \vee \ b \leq a$$

we call it a *total order*. We write $a < b$ for $a \leq b \wedge b \neq a$.

Definition A.1.2 A partially ordered set $(S, \leq)$ is said to be *well-founded* if there are no infinite (strictly) descending sequences $s_1 > s_2 > s_3 > \ldots$ of elements of S.

Definition A.1.3 A *multiset* m over a set S is a total mapping $m : S \to \mathbb{N}_0$. A multiset is *finite* if almost all $s \in S$ are mapped to 0. Multisets are written inside slashed braces ($\{\!|$ _ $|\!\}$). We say that an element of S that is mapped to n has n *occurrences*.

For totally ordered S and finite multisets over S we define: $m_1 > m_2$ if m_2 results from m_1 by replacing one occurrence of one element s by finitely many occurrences of an element t, where $t < s$. The transitive closure of this relation is an (irreflexive) partial order.

A theorem by Dershowitz and Manna [26] states that a preorder on a set S is well-founded if and only if the induced multiset ordering on the set $M(S)$ of finite multisets over S is well-founded.

Let $(M, \leq)$ be a partially ordered set and $A \subset M$. An element $s \in M$ is called an *upper bound* of A if $\forall a \in A(a \leq s)$. Analogously, an element $t \in M$ is called a *lower bound* of A if $\forall a \in A(a \geq t)$.

Definition A.1.4 If $A \subset M$ possesses upper bounds and there is a least element s in the set of upper bounds, we call s the *least upper bound* of A, or in symbols $s = \text{lub}(A)$. Analogously, if $A \subset M$ possesses lower bounds and there is a least element t in the set of upper bounds, we call t the *greatest lower bound* of A, or in symbols $t = \text{glb}(A)$.

Definition A.1.5 A partially ordered set $(M, \leq)$ is called *lattice* if every subset of M consisting of two elements possesses a greatest lower bound and a least upper bound.

It is easy to see that the greatest lower bound and the least upper bound of two elements are uniquely determined. We denote the greatest lower bound and the least upper bound of two elements a and b by

$$a \sqcap b := \text{glb}(\{a, b\})$$
$$a \sqcup b := \text{lub}(\{a, b\}).$$

A.1.1 Feature Trees

This part is taken from [3, 87]. A *path* is a word over the set of all features. The symbol ϵ denotes the empty path. A path p is called a *prefix* of a path q if there exists a path p' such that $pp' = q$. We use FEA^* to denote the set of all paths.

A *tree domain* is a nonempty set $D \subseteq \text{FEA}^*$ of paths that is *prefix-closed*, i.e. if $pq \in D$ then $p \in D$. Note that every tree domain contains the empty path.

A *feature tree* is a partial function $\sigma : \text{FEA}^* \rightarrow \text{CON}$ whose domain $dom(\sigma)$ is a tree domain. The paths in the domain of a feature tree represent the nodes of the tree; the empty path represents its root. The *subtree* $p\sigma$

of a feature tree σ at a path $p \in dom(\sigma)$ is the feature tree defined by (in relational notation) $p\sigma := \{(q, c) \mid (pq, c) \in \sigma\}$. A feature tree σ is called a *subtree* of a feature tree τ if σ is a subtree of τ at some path $p \in dom(\tau)$, and a *direct subtree* if $p = f$ for some feature f. A feature tree σ is called *rational* if σ has only finitely many subtrees and σ is finitely branching (i.e., for every $p \in dom(\sigma)$, the set $\{pf \in dom(\sigma) \mid f \in \mathrm{FEA}\}$ is finite). Note that for every rational feature tree, there exist finitely many features $f_1, \ldots, f_n$ such that $dom(\sigma) \subseteq \{f_1, \ldots, f_n\}^*$.

A.2 Order-Sorted Logic

Many-sorted logic is the basis for algebraic specifications [29, 28, 61], rewriting techniques [46, 47], and unification theory [45, 85]. Its results extend to order-sorted logic [86, 84] under certain conditions. In many-sorted logic, the sorts are completely unrelated, while in the order-sorted case, there is a subsort relationship. The definitions follow the notation of [86].

Syntax We use lower case bold roman font for *sort symbols*, e.g. $\mathbf{p}$, $\mathbf{f}$. *Function symbols* are declared with their arity. If the arity is zero, we call them *constant symbols*. We use x, y, z for variables. Every variable x has a *sort*, $\mathrm{sort}(x)$, which is a sort symbol. A *subsort declaration* has the form $\mathbf{r} < \mathbf{s}$, where $\mathbf{r}$ and $\mathbf{s}$ are sort symbols. A *function declaration* has the form

$$f \_ \ldots \_ : \mathbf{s}_1 \times \ldots \times \mathbf{s}_n \to \mathbf{s}$$

where n is the arity of f. Function symbols do not have to be written in prefix notation; they may appear in "mixfix".

A *signature* Σ is a set of subsort and function declarations. The *subsort order* $\mathbf{r} \leq_\Sigma \mathbf{s}$ of Σ is the least preorder $\leq_\Sigma$ on the sort symbols of Σ so that $\mathbf{r} \leq_\Sigma \mathbf{s}$ if $\mathbf{r} < \mathbf{s} \in \Sigma$. The subsort order is extended componentwise to strings of sort symbols. If there is no danger of confusion, the index Σ is omitted.

Let Σ be a signature. A Σ-term of sort $\mathbf{s}$ is either a variable x so that $\mathrm{sort}(x) \leq_\Sigma \mathbf{s}$, or it has the form $f(s_1, \ldots, s_n)$ and there is a declaration

$$(f \_ \ldots \_ : \mathbf{s}_1 \times \ldots \times \mathbf{s}_n \to \mathbf{r}) \in \Sigma$$

such that $\mathbf{r} \leq_\Sigma \mathbf{s}$ and, for $i \in \{1, \ldots, n\}$, s_i is a Σ-term of sort $\mathbf{s}_i$.

A Σ-equation is an ordered pair of Σ-terms written as $s \doteq t$. A Σ-term is called *ground* if it contains no variables. $\mathcal{V}s$ is used for the set of variables occurring in the term s. $dom(f)$ is used for the *domain* of the function f. A Σ-substitution θ is a function from Σ-terms to Σ-terms such that

- if s is a Σ-term of sort $\mathbf{s}$, then θs is a Σ-term of sort $\mathbf{s}$,
- $\theta f(s_1, \ldots, s_n) = f(\theta s_1, \ldots, \theta s_n)$,
- $dom(\theta) := \{x \mid \theta x \neq x\}$ is finite.

A signature Σ is called *regular* if

1. the subsort order of Σ is a partial order and
2. every Σ-term s has a least sort $\mathrm{sort}(s)$, i.e. there is a unique function $\mathrm{sort}(\_)$ from the set of all Σ-terms to the set of sort symbols such that
   - if s is a Σ-term then s is a term of sort $\mathrm{sort}(s)$,
   - if s is a Σ-term of sort $\mathbf{s}$ then $\mathrm{sort}(s) \leq \mathbf{s}$.

Substitution [23] of a variable v by term e in term e' is denoted by $[e/v]e'$.

$$[e_2/v_2]([e_1/v_1]e)$$

is abbreviated to

$$[e_1/v_1, e_2/v_2]e\ .$$

Substitutions are always assumed to be *capture avoiding*, i.e. bound variables are appropriately renamed.

A *specification* $\mathcal{S} = (\Sigma, \mathcal{E})$ consists of a signature Σ and a set $\mathcal{E}$ of Σ-equations, called *axioms* of $\mathcal{S}$. Σ or $\mathcal{E}$ may be countably infinite.

Semantics Let Σ be a signature. A Σ-*algebra* $\mathcal{A}$ consists of *denotations* $\mathbf{s}^{\mathcal{A}}$ and $f^{\mathcal{A}}$ for the sort and function symbols of Σ such that:

- $\mathbf{s}^{\mathcal{A}}$ is a set.
- If $(\mathbf{r} < \mathbf{s}) \in \Sigma$, then $\mathbf{r}^{\mathcal{A}} \subset \mathbf{s}^{\mathcal{A}}$.
- $C_{\mathcal{A}} := \bigcup\{\mathbf{s}^{\mathcal{A}} \mid \mathbf{s} \text{ is a sort symbol of } \Sigma\}$ is called the *carrier* of $\mathcal{A}$.
- $f^{\mathcal{A}}$ is a mapping $D_f^{\mathcal{A}} \to C_{\mathcal{A}}$ whose domain $D_f^{\mathcal{A}}$ is a subset of $C_{\mathcal{A}}^n$, where n is the arity of f.
- If $(f \_ \ldots \_ : \mathbf{s}_1 \times \ldots \times \mathbf{s}_n \to \mathbf{s}) \in \Sigma$ and $a_i \in \mathbf{s}_i^{\mathcal{A}}$ $(i \in \{1, \ldots, n\})$, then $(a_1, \ldots, a_n) \in D_f^{\mathcal{A}}$ and $f^{\mathcal{A}}(a_1, \ldots, a_n) \in \mathbf{s}^{\mathcal{A}}$.

$C_{\mathcal{A}}^n$ denotes the cartesian product $C_{\mathcal{A}} \times \ldots \times C_{\mathcal{A}}$ with n factors.

A function symbol has only one denotation, although it may have more than one declaration in the signature. A model satisfies a declaration of a function symbol f if the domain of the denotation of f includes the declared domain, and the denotation of f maps every element of the declared domain to an element of the declared codomain [86].

Let V be a set of Σ-variables. A Σ-*assignment* is a mapping $\mathfrak{a} : V \to C_{\mathcal{A}}$ such that $\mathfrak{a}(x) \in (\text{sort}(x))^{\mathcal{A}}$ for all variables $x \in V$. Given a Σ-assignment $\mathfrak{a}$ and a Σ-term s, the *denotation* $[\![s]\!]_{\mathfrak{a}}$ of s in $\mathcal{A}$ under $\mathfrak{a}$ is defined as follows:

$$\begin{aligned} [\![x]\!]_{\mathfrak{a}} &= \mathfrak{a}(x), \\ [\![f(s_1, \ldots, s_n)]\!]_{\mathfrak{a}} &= f^{\mathcal{A}}([\![s_1]\!]_{\mathfrak{a}}, \ldots, [\![s_n]\!]_{\mathfrak{a}}) \,. \end{aligned}$$

Let x be a variable and X a set of variables. An assignment $\mathfrak{b}$ is called an *x-update* (X-update) of an assignment $\mathfrak{a}$ if $\mathfrak{b}$ and $\mathfrak{a}$ agree everywhere except possibly on x (X). Validity of Σ-equations in a Σ-algebra $\mathcal{A}$ is defined as follows:

$$\mathcal{A} \models s \doteq t \quad :\Leftrightarrow \quad \forall \Sigma\text{-assignments } \mathfrak{a} \;\; ([\![s]\!]_{\mathfrak{a}} = [\![t]\!]_{\mathfrak{a}}) \,.$$

If $\mathcal{A} \models s \doteq t$, we say that $s \doteq t$ is *valid* in $\mathcal{A}$ or that $\mathcal{A}$ *satisfies* $s \doteq t$. Our formulae only use conjunction and existential quantification. Let $\mathcal{A}$ be a Σ-algebra and $\mathfrak{a}$ a Σ-assignment. $(\phi)^{\mathcal{A}}$ are the solutions of ϕ in $\mathcal{A}$, i.e. the

set of all Σ-assignments such that

$$\begin{aligned}(\phi \wedge \psi)^{\mathcal{A}} &= (\phi)^{\mathcal{A}} \cap (\psi)^{\mathcal{A}} \\ (\exists v(\phi))^{\mathcal{A}} &= \{a \mid \exists d \in C_{\mathcal{A}}([d/v]a \in (\phi)^{\mathcal{A}})\}\end{aligned}$$

Let $\mathcal{S} = (\Sigma, \mathcal{E})$ be a specification and $\mathcal{A}$ a Σ-algebra. $\mathcal{A}$ is a *model* of $\mathcal{S}$ if $\mathcal{A}$ satisfies every equation of $\mathcal{E}$.

Definition A.2.1 We say that ψ is a *consequence* of ϕ (written $\phi \models \psi$) if and only if every interpretation which is a model of ϕ is also a model of ψ.

We use the symbol $\models$ for both the satisfaction relation ($\mathcal{A} \models \phi$) and for the *consequence relation* ($\phi \models \psi$). The symbol preceding "$\models$" determines the meaning.

Definition A.2.2 Let ϕ be a constraint of logical connectives and existential quantifiers. The *prenex normal form* [59] of ϕ is $\exists v_1, \ldots, v_m(\phi')$, where $v_1, \ldots, v_m$ are all existentially quantified variables of ϕ, and ϕ' contains no more existential quantifiers.

Let $\mathcal{A}$ and $\mathcal{B}$ be Σ-algebras. A mapping $\gamma : C_{\mathcal{A}} \to C_{\mathcal{B}}$ is called a *homomorphism* $\mathcal{A} \to \mathcal{B}$ if

1. $\gamma(\mathbf{s}^{\mathcal{A}}) \subseteq \mathbf{s}^{\mathcal{B}}$ for every Σ-sort symbol $\mathbf{s}$,
2. $\gamma(D_f^{\mathcal{A}}) \subseteq D_f^{\mathcal{B}}$ for every Σ-function symbol f,
3. $\gamma(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(\gamma(a_1), \ldots, \gamma(a_n))$ for every Σ-function symbol f and every tuple $(a_1, \ldots, a_n) \in D_f^{\mathcal{A}}$.

Let $\mathcal{A}$ and $\mathcal{B}$ be Σ-algebras. We say that a homomorphism $\gamma : \mathcal{A} \to \mathcal{B}$ is an $\mathcal{A} \to \mathcal{B}$ *covering* if the following two conditions are satisfied:

1. if $\mathbf{s}$ is a Σ-sort symbol and $b \in \mathbf{s}^{\mathcal{B}}$, then there exists an $a \in \mathbf{s}^{\mathcal{A}}$ such that $\gamma(a) = b$,
2. if f is a Σ-function symbol and $(b_1, \ldots, b_n) \in D_f^{\mathcal{B}}$, then there exists $(a_1, \ldots, a_n) \in D_f^{\mathcal{A}}$ such that $\gamma(a_i) = b_i$ for $i = 1, \ldots, n$.

Let Σ be a signature and V a set of Σ-variables. We construct the *term algebra* $\mathcal{T}_{\Sigma,V}$ by:

- $\mathrm{s}^{\mathcal{T}_{\Sigma,V}} := \{s \mid s \text{ is a } (\Sigma, V)\text{-term of sort s}\}$
- $D_f^{\mathcal{T}_{\Sigma,V}} := \{(s_1, \ldots, s_n) \mid f(s_1, \ldots, s_n) \text{ is a } (\Sigma, V)\text{-term}\}$
- $f^{\mathcal{T}_{\Sigma,V}}(s_1, \ldots, s_n) := f(s_1, \ldots, s_n)$

Let $\mathcal{A}$ be a Σ-algebra. An equivalence relation $\approx$ on the carrier of $\mathcal{A}$ is called a *congruence* on $\mathcal{A}$ if for every Σ-function symbol f

$$a_1 \approx b_1 \wedge \ldots \wedge a_n \approx b_n \;\Rightarrow\; f^{\mathcal{A}}(a_1, \ldots, a_n) \approx f^{\mathcal{A}}(b_1, \ldots, b_n)$$

provided $(a_1, \ldots, a_n) \in D_f^{\mathcal{A}}$ and $(b_1, \ldots, b_n) \in D_f^{\mathcal{A}}$.

A.3 Confluence and Termination

Definition A.3.1 An (algebraic) Σ-*rewrite rule* $s \rightsquigarrow t$ is a Σ-equation $s \doteq t$ such that s is not a variable, and every variable occurring in the right-hand side t occurs in the left-hand side s. An *(algebraic) rewriting system* is a specification $\mathcal{R} = (\Sigma, \mathcal{E})$ such that every equation in $\mathcal{E}$ is an (algebraic) rewrite rule.

The notion of algebraic rewriting rules [27] is used to differentiate these rules from the β-reduction rule of the λ-calculus.

The following notation and theorems come from [46]. Let $\_ \rightsquigarrow \_$ be a binary relation on some set. Then, $\_ \overset{*}{\rightsquigarrow} \_$ denotes the reflexive and transitive closure of $\_ \rightsquigarrow \_$, and $\_ \overset{*}{\leftrightsquigarrow} \_$ denotes the reflexive, symmetric, and transitive closure of $\_ \rightsquigarrow \_$. We write $s \downarrow t$ (read "s and t *converge*") if there is an r such that $s \overset{*}{\rightsquigarrow} r$ and $t \overset{*}{\rightsquigarrow} r$. The relation $\longrightarrow$ is called *locally confluent* if $r \rightsquigarrow s$ and $r \rightsquigarrow t$ implies $s \downarrow t$. The relation $\longrightarrow$ is called *confluent* if $r \overset{*}{\rightsquigarrow} s$ and $r \overset{*}{\rightsquigarrow} t$ implies $s \downarrow t$. The relation $\longrightarrow$ is called *terminating* if there are no infinite chains $s_1 \rightsquigarrow s_2 \rightsquigarrow \ldots$. An element s is called *normal* if there is no t such that $s \rightsquigarrow t$. An element t is called a *normal form* of s if $s \overset{*}{\rightsquigarrow} t$ and t is normal.

Proposition A.3.2 Let $\longrightarrow$ be a confluent relation. Then no element has more than one normal form. If $\longrightarrow$ is confluent and terminating, then every element has exactly one normal form.

Theorem A.3.3 Let $\longrightarrow$ be a confluent relation. Then $s \overset{*}{\leftrightsquigarrow} t$ if and only if $s \downarrow t$.

Theorem A.3.4 A relation is confluent if it is locally confluent and terminating.

The following definitions come from [86]. A syntactical Σ-object, O', is a *variant* of a Σ-object O if O' is obtainable from O by consistent variable renaming, i.e. there exist Σ-substitutions θ and ψ such that $O' = \theta O$ and $O = \psi O'$.

Definition A.3.5 An *overlap* of a rewrite system $\mathcal{R}$ is a triple

$$(s \rightsquigarrow t, \pi, s' \rightsquigarrow t')$$

such that

- $s \rightsquigarrow t$ and $s' \rightsquigarrow t'$ are variable disjoint variants of rules of $\mathcal{R}$, and π is a position of s such that s/π is not a variable,
- if $s/\pi = s$, then $s \rightsquigarrow t$ is not a variant of $s' \rightsquigarrow t'$,
- there exists an $\mathcal{R}$-substitution θ such that $(\theta s)/\pi = \theta s'$.

An overlap $(s \rightsquigarrow t, \pi, s' \rightsquigarrow t')$ is called a *variant* of an overlap

$$(u \rightsquigarrow v, \pi, u' \rightsquigarrow v')$$

if $u \rightsquigarrow v$ is a variant of $s \rightsquigarrow t$ and $u' \rightsquigarrow v'$ a variant of $s' \rightsquigarrow t'$.

Definition A.3.6 A *critical pair* of an overlap $(s \rightsquigarrow t, \pi, s' \rightsquigarrow t')$ of $\mathcal{R}$ is a pair $(\theta t, \theta(s[\pi \leftarrow t']))$ such that $(\theta s)/\pi = \theta s'$, $\theta(s[\pi \leftarrow t'])$ is an $\mathcal{R}$-term, and θ is an $\mathcal{R}$-substitution. A pair (s, t) is called $\mathcal{R}$-critical if (s, t) is a critical pair of an overlap of $\mathcal{R}$. We say that a pair (s, t) *converges* in $\mathcal{R}$ if $s \downarrow_{\mathcal{R}} t$.

The notations and results of term rewriting [46, 47] are generalized to order-sorted logic.

Proposition A.3.7 Let $\mathcal{R}$ be a sort decreasing rewriting system. Then $\mathcal{R}$ is locally confluent if and only if all critical pairs of $\mathcal{R}$ converge.

Bibliography

[1] S. Abramsky. Computational interpretation of linear logic. *Theoretical Computer Science*, 1992. to appear.

[2] R. Amadio and L. Cardelli. Subtyping recursive types. In *Symposium on Principles of Programming Languages*, pages 104–118, Orlando, Florida, Jan. 1991. ACM.

[3] R. Backofen and G. Smolka. A complete and recursive feature theory. Research Report RR-92-30, DFKI, Saarbrücken, July 1992. Short version appeared in the 1992 *Annual Meeting of the Association for Computational Linguistics.*

[4] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics.* North-Holland, 1981.

[5] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings Publishing Company, Inc., 1991.

[6] A. H. Borning and D. H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Symposium on Principles of Programming Languages*, pages 133–139. ACM, 1982.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *Object-Oriented Programming Systems, Languages and Applications and European Conference on Object-Oriented Programming*, pages 303–311. ACM, Oct. 1990.

[8] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Logic in Computer Science*, pages 112–128, 1989.

[9] K. B. Bruce. A paradigmatic object-oriented programming language: Design, static typing, and semantics. Technical Report CS-92-01, Williams College, Williamstown, MA 01267, Jan. 1992.

[10] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Conference on Functional Programming and Computer Architecture*, pages 273–280. ACM, 1989.

[11] L. Cardelli. A semantics of multiple inheritance. *Lecture Notes in Computer Science*, 173:51–67, 1984. revised in: Information and Computation, Vol. 76, 1988, pp. 138-164.

[12] L. Cardelli. Basic polymorphic typechecking. In *Science of Computer Programming*, pages 147–172. North Holland, 1987. Vol. 8.

[13] L. Cardelli. Structural subtyping and the notion of power type. In *Symposium on Principles of Programming Languages*, pages 70–79. ACM, Jan. 1988.

[14] L. Cardelli and J. C. Mitchell. Operations on records. *Lecture Notes in Computer Science*, 389:75–81, 1989. extended abstract.

[15] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, Dec. 1985.

[16] F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92:48–80, 1991.

[17] V. Claus, H. Fleischhack, A. Giesler, A. V. Hense, B. Koether, P. Kubitzsch, K. Lattemann, T. Nowak, S. Schmidt, P. Schneider, W. Strauch, A. Weber, S. Wiegand, and T. Wolters. Zwischenbericht der Projektgruppe Netzmodelle für Bürosysteme. Technical report, Universität Dortmund, 1985. in German.

[18] V. Claus, H. Fleischhack, H. Huwig, A. Giesler, A. V. Hense, B. Koether, P. Kubitzsch, K. Lattemann, T. Nowak, S. Schmidt, P. Schneider, W. Strauch, A. Weber, S. Wiegand, and T. Wolters. Abschlußbericht der Projektgruppe Netzmodelle für Bürosysteme. Technical report, Universität Dortmund, 1986. in German.

[19] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, Oct. 1989.

[20] A. Colmerauer. *Logic Programming*, chapter Prolog and Infinite Trees, pages 231–251. Academic Press, 1982.

[21] W. R. Cook. A denotational semantics of inheritance. Technical Report CS-89-33, Brown University, Dept. of Computer Science, Providence, Rhode Island 02912, May 1989.

[22] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[23] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[24] L. Damas. *Type Assignment in programming languages.* PhD thesis, University of Edinburgh, 1985. CST-33-85.

[25] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[26] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Commun. ACM*, 22:465–476, 1979.

[27] D. J. Dougherty. Adding algebraic rewriting to the untyped lambda calculus. In R. V. Book, editor, *Rewriting Techniques and Applications, 4th RTA-91, LNCS 488*, pages 37–48. Springer, 1991.

[28] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*, volume 1: Equations and Initial Semantics of *EATCS Monographs on Theoretical Computer Science.* Springer, 1985.

[29] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. Yeh, editor, *Data Structuring*, volume IV of *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978.

[30] A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, 1983. revised in 1989.

[31] A. Goldberg and D. Robson. *Smalltalk-80: the Language.* Addison-Wesley, 1989.

[32] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. *Lecture Notes in Computer Science*, 78, 1979.

[33] J. O. Graver. *Type-Checking and Type Inference for Object-Oriented Programming Languages.* PhD thesis, University of Illinois at Urbana-Champaign, 1989.

[34] J. O. Graver and R. E. Johnson. A type system for Smalltalk. In *Symposium on Principles of Programming Languages*, pages 136–150, San Francisco, Jan. 1990. ACM.

[35] R. Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, University of Edinburgh, Nov. 1986. Laboratory for Foundations of Computer Science Report Series.

[36] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Symposium on Principles of Programming Languages*, pages 131–142, Orlando, Fla., Jan. 1991. ACM.

[37] A. V. Hense. An O'small interpreter based on denotational semantics. Technical Report A 07/91, Universität des Saarlandes, Fachbereich 14, Nov. 1991.

[38] A. V. Hense. Wrapper semantics of an object-oriented programming language with state. In Ito and Meyer [48], pages 548–568.

[39] A. V. Hense. Denotational semantics of an object-oriented programming language with explicit wrappers. *Formal Aspects of Computing*, 5(3):181–207, 1993.

[40] A. V. Hense and G. Smolka. A verification of extensible record types. In Z. Shi, editor, *Proceedings of the IFIP TC12/WG12.3 International Workshop on Automated Reasoning*, pages 137–164, Beijing, P.R. China, 13–16 July 1992. International Federation for Information Processing, Elsevier, North-Holland, Excerpta Medica.

[41] A. V. Hense and R. Wilhelm. Evaluation of applicative style attributes using lazy memo functions. Technical Report Doc.: M.1.1.S.1.3-DI-6.0, European Strategic Programme for Research and Development in Information Technology, 1989. PROSPECTRA, Project Ref. No. 390.

[42] J. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[43] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.

[44] M. Höhfeld and G. Smolka. Definite relations over constraint languages. LILOG-Report 53, IBM Deutschland, Oct. 1988.

[45] G. Huet. *Résolution d'équations dans les langages d'ordre 1,2,...,ω.* PhD thesis, Université Paris 7, 1976. Thèse de doctorat d'état.

[46] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

[47] G. Huet and D. Oppen. Equations and rewrite rules: A survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, New York, 1980.

[48] T. Ito and A. R. Meyer, editors. *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science.* Springer-Verlag, Sept. 1991.

[49] L. Jategaonkar and J. Mitchell. Type inference with extended pattern matching and subtypes. *Fundamenta Informaticae*, 19:127–166, 1993.

[50] L. A. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes. In *Symposium on Lisp and Functional Programming*, 1988.

[51] K. Jensen and N. Wirth. *PASCAL User Manual and Report.* Springer-Verlag, 1975.

[52] R. E. Johnson. Type-checking Smalltalk. In *Object-Oriented Programming Systems, Languages and Applications*, pages 315–321. ACM, Sept. 1986.

[53] P. Kanellakis and J. Mitchell. Polymorphic unification and ML typing. In *16th Symp. Principles of Programming Languages*, pages 105–115, 1989.

[54] B. W. Kernighan and D. M. Ritchie. *The C Programming Language - Reference Manual.* Prentice-Hall, 1978.

[55] J. W. Klop. Term rewriting systems: a tutorial. *Bulletin of the European Association for Theoretical Computer Science*, 32, 1987.

[56] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *Symposium on Principles of Programming Languages*, pages 291–302. ACM, 1991.

[57] X. Leroy and P. Weis. *Manuel de reference du langage Caml.* InterEditions, Paris, 1993.

[58] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[59] M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Logic in Computer Science*, Edinburgh, 1988.

[60] H. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Symposium on Principles of Programming Languages*, pages 382–401, 1990.

[61] J. Meseguer and J. Goguen. *Algebraic Methods in Semantics*, chapter Initiality, Induction, and Computability. Cambridge University Press, 1985.

[62] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in Elf. Technical Report MPI-I-91-211, Max-Planck-Institut für Informatik, Saarbrücken, Aug. 1991.

[63] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[64] R. Milner. A proposal for Standard ML. In *Symposium on Lisp and Functional Programming*, pages 184–197, Austin Texas, 1984. ACM.

[65] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, 1990.

[66] J. C. Mitchell. Coercion and type inference. In *Symposium on Principles of Programming Languages.* ACM, 1984.

[67] J. C. Mitchell and R. Harper. The essence of ML. In *Symposium on Principles of Programming Languages*, pages 28–46. ACM, Jan. 1988.

[68] E. Moggi. Computational lambda-calculus and monads. *Logic in Computer Science*, pages 14–23, 1989.

[69] F. Müller. Confluence of the lambda calculus with left-linear algebraic rewriting. *Inf. Process. Lett.*, 41:293–299, Apr. 1992.

[70] A. Ohori. *A Study of Types, Semantics, and Languages for Databases and Object-oriented Programming.* PhD thesis, University of Pennsylvania, 1989.

[71] A. Ohori. Representing object identity in a pure functional language. In *International Conference on Database Theory*, 1991.

[72] A. Ohori and P. Buneman. Static type inference for parametric classes. *Object-Oriented Programming Systems, Languages and Applications*, pages 445–456, 1989.

[73] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Object-Oriented Programming Systems, Languages and Applications '91*, pages 146–161. ACM, Nov. 1991.

[74] J. Palsberg and M. Schwartzbach. Static typing for object-oriented programming. Technical Report DAIMI PB-355, Computer Science Department, Aarhus University, June 1991.

[75] J. Palsberg and M. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[76] F. Pfenning. Partial polymorphic type inference and higher-order unification. In Snowbird, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Utah, July 1988. ACM Press.

[77] F. Pfenning. On the undecidability of partial polymorphic type reconstruction. Technical Report CMU–CS–92–105, Carnegie Mellon University, Pittsburgh, Pennsylvania, Jan. 1992.

[78] F. Pfenning and P. Lee. LEAP: A language with eval and polymorphism. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development*, pages 345–359, Barcelona, Spain, Mar. 1989. Springer-Verlag. Lecture Notes in Computer Science 352.

[79] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Symposium on Principles of Programming Languages*, pages 77–88. ACM, 1989.

[80] D. Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Languages Fonctionnels.* PhD thesis, Université Paris 7, Feb. 1990.

[81] D. Rémy. Typing record concatenation for free. In *Symposium on Principles of Programming Languages*, pages 166–176. ACM, 1992.

[82] D. Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, INRIA, 1993.

[83] J. Reynolds. Syntactic control of interference, part 2. Technical Report CMU-CS-89-130, Carnegie Mellon University, 1989.

[84] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, volume 395 of *Lecture Notes in Computer Science.* Springer-Verlag, 1989.

[85] J. Siekmann. Unification theory. In *European Conference on Artificial Intelligence*, pages vi–xxxv, Brighton Centre, England, 1986.

[86] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. *Order-Sorted Equational Computation*, volume 2 of *Resolution of Equations in Algebraic Structures*, chapter 10, pages 297–367. Academic Press, 1989.

[87] G. Smolka and R. Treinen. Records for logic programming. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 240–254, Washington D.C., USA, 9–12 Nov. 1992. The MIT Press. Extended version to appear in *Journal of Logic Programming.*

[88] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Object-Oriented Programming Systems, Languages and Applications*, pages 38–45. ACM, Sept. 1986.

[89] R. Stansifer. Type inference with subtypes. In *Symposium on Principles of Programming Languages*, pages 88–97. ACM, Jan. 1988.

[90] B. Stroustrup. *The C++ programming language.* Addison-Wesley, 1986.

[91] N. Suzuki. Inferring types in Smalltalk. In *Symposium on Principles of Programming Languages*, pages 187–199. ACM, Jan. 1981.

[92] N. Suzuki and M. Terada. Creating efficient systems for object-oriented languages. In *Symposium on Principles of Programming Languages*, pages 290–296, 1984.

[93] J.-P. Talpin and P. Jouvelot. Type, effect and region reconstruction and its applications. In *International Workshop on Compilers for Parallel Computers*, pages 411–416, Paris, Dec. 1990.

[94] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science*, pages 162–173, 1992.

[95] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, May 1988. CST-52-88 also published as ECS-LFCS-88-54.

[96] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[97] D. Turner. Miranda: A non-strict functional language with polymorphic types. *Lecture Notes in Computer Science*, 201:1–16, 1985. Functional Programming Languages and Computer Architecture.

[98] P. Wadler. Comprehending monads. In *Symposium on Lisp and Functional Programming*, pages 61–78. ACM, 1990.

[99] M. Wand. Complete type inference for simple objects. In *Logic in Computer Science*, pages 37–44, 1987.

[100] M. Wand. Corrigendum: Complete type inference for simple objects. In *Logic in Computer Science*, page 132, 1988.

[101] M. Wand. Type inference for record concatenation and multiple inheritance. In *Logic in Computer Science*, pages 92–97, 1989.

[102] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, July 1991.

[103] P. Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. MIT Press, 1987.

[104] P. Weis and X. Leroy. *Le langage Caml.* InterEditions, Paris, 1993.

[105] R. Wilhelm. personal communication, Feb. 1993.

[106] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Department of Computer Science, Rice University, Apr. 1991.

[107] A. K. Wright. Typing references by effect inference. *Lecture Notes in Computer Science*, 582:473–491, Feb. 1992. ESOP' 92.

[108] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report TR 93-200, Rice University, Feb. 1993.

[109] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* MIT press, 1990.

Index