RICE UNIVERSITY

Soft Typing: An Approach to Type Checking for Dynamically Typed Languages

by

Mike Fagan

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Robert S. Cartwright, Chairman
Professor of Computer Science

Matthias Felleisen
Assistant Professor of Computer Science

Richard Tapia
Professor of Mathematical Sciences

Bruce Duba
Research Associate of Computer Science

Houston, Texas

August, 1992

Soft Typing: An Approach to Type Checking for Dynamically Typed Languages

Mike Fagan

Abstract

In an effort to avoid improper use of program functions, modern programming languages employ some kind of preventative type system. These type systems can be classified as either static or dynamic. Static type systems detect "ill-typed" program phrases at compile-time, whereas dynamic type systems detect "ill-typed" phrases at run-time.

Static typing systems have two important advantages over dynamically typed systems: First, they provide important feedback to the programmer by detecting a large class of program errors *before* execution. Second, they extract information that a compiler can exploit to produce more efficient code.

The price paid for these advantages, however, is a loss of expressiveness and modularity. It is easy to prove that a static type system for an "interesting" programming language necessarily excludes some "good" programs.

This paper focuses on the problem of designing programming systems that retain *all* the expressiveness of dynamic typing, but still offer the early error detection and improved optimization opportunities of static typing. To that end, we introduce a concept called *soft typing*.

The key concept of soft typing is that a type checker need not *reject* programs containing statically "ill-typed" phrases. Instead, the soft type checker inserts explicit run-time checks. Thus, there are two issues to be addressed in the design of soft typing systems. First, the typing mechanism must provide *reasonable* feedback to programmers accustomed to dynamically typed languages. Current static systems fail to satisfy the programmer's intuition about correctness on many programs. Second, a soft typing system must sensibly insert run-time checks (when necessary). This paper develops a type system and checking algorithms that are suitable for soft typing a significant class of programming languages.

Acknowledgments

A thesis is never the work of one person. In my case, this is especially true. A great many people contributed to this work. This section is an attempt to recognize and thank the people who analyzed, cajoled, consoled, and otherwise supported me in this effort.

First and foremost, I would like to thank my committee. Every Ph.D. candidate owes a debt to his committee, but in my case, the debt is exceptionally large. Thanks to Corky Cartwright, my committee chairman, who asked the appropriate questions, and provided a great deal of the intuition behind the ideas of this thesis. Corky never failed to see the "big picture", and his vision was the driving force for this thesis. Thanks to Matthias Felleisen for helping me break down large problems into smaller ones, and for reviewing my prose. Thanks to Bruce Duba for technical help on ML, as well as his interest in this work. Matthias and Bruce receive special commendation for their willingness to be added somewhat late in the game. Special thanks to Richard Tapia, who agreed to serve as my outside member. Richard has known me for years, and he *still* agreed to serve.

Additional thanks to Hans Boehm, who originally served on this committee, before he moved on from the Rice community. Hans provided very valuable insights into type checking unusual languages.

Special thanks to Tim Griffin, who first introduced me to the advanced unification theory. Tim provided many alternatives to the approach I initiated. The resulting synthesis was instrumental in analyzing the results in this thesis.

I would also like to thank the funding agencies for supporting this work. So, thank you IBM, thank you DARPA (PRISM project), and special thanks to the Texas Advanced Research Program, and the Advanced Technology Program.

Besides the faculty, I would also like to thank the students for their valuable feedback. Becky, Dorai, Rama, Mark, Andrew, Jerry, Mike, Marina — thanks a lot. I appreciated it. Also, special thanks to Dave Johnson, who always answered my TEX questions.

Anyone who has ever written a thesis also realizes that there are many people who contribute moral support. This kind of support is as necessary as technical support.

First, I acknowledge Keith Cooper and Linda Torczon, who first encouraged me to come back to graduate school. They convinced me I wasn't too old.

Second, special thanks to all my friends in the dance community. After 12 or 14 hours of left brain activity, the right brain really needs some exercise to maintain balance. While the list of people is too large to cover adequately, I'd like to mention Glen Hunsucker and his crew, Rick Archer and SSQQ, Mario and the X-team, Debbie, Jeannie, and Kay.

Extra special thanks to Carrie Estes. She doesn't have any interest in computer science, but she still provided lots of TLC.

In a similar fashion, my two oldest friends, Joe and Jeff, deserve some kind of medal for keeping me sane (or what passes for sanity in my case). I hope you guys know how much your encouragement has meant.

Last, but certainly not least, I must thank my family. Mom, Dad, Pattye, Steve, Kyle, John, and Paigee Beth — thanks so much. Without you, this whole project would never have been done.

I dedicate this thesis to my family.

Contents

Chapter 1

Introduction

Most modern programming languages support some notion of "type". While formal accounts of "type" vary, they all focus on the idea of confirming that applications of program functions to arguments are well defined. Most primitive program operations presume that their arguments satisfy certain constraints. A similar observation holds for functions and arguments in the realm of mathematics. The expression $f : A \to B$ indicates that function f maps elements from set A to elements of set B. The notation implies that f is defined for all elements of A, and maps these elements to elements of B. On the other hand, if x is *not* an element of A, then $f(x)$ has *no* meaning. In other words, functions in mathematics have a specified *domain of definition*, a set of values for which application is considered well-formed. The same intuition underlies the notion of typing in programming languages.

Like mathematical functions, program operations have domains of definition. Most program functions are can be meaningfully applied only to a specific set of values. If a function is applied to arguments outside of the intended set, the results are unpredictable. Consider the function **add1**. This function takes an integer n as input and produces the integer $n + 1$ as its value. For example, $\mathbf{add1}(5) = 6$. On the other hand, the application of of **add1** to the value "string" is meaningless. Attempting to apply **add1** to a string is an example of faulty application.

Since the programs submitted to a language translator may contain undefined applications, the language must provide some mechanism for coping with them. For this reason, most programming languages impose a type discipline on program text. This discipline can either be *static* or *dynamic*. The essential difference between static typing and dynamic typing lies in when type faults are detected. A static type system examines program text for dubious applications *before* any execution is attempted. If a program contains a possible type fault, it is *not* executed. Dynamic type systems, on the other hand, confirm that each application is appropriate during program execution.

Since static type analysis takes place *before* execution, a static type system must provide a set of rules determining whether or not a piece of program text is *well-typed.* The set of well-typed program expressions as defined by the type system must satisfy two criteria:

1. No well-typed program has any applications that will give run-time errors.

2. The property of being well-typed must be *decidable* for arbitrary program text.

Since well-typedness is decidable, the implementation of a statically typed language must provide a *type-checker*[1] to make that decision. In a statically typed language, programs deemed statically type correct by the type checker are eligible for execution. Programs *not* deemed statically type correct are systematically rejected. In other words, the type-checker acts as a filter on programs.

Advocates of static typing cite three advantages to programmers:

- Static typing performs a weak form of program verification. Clerical and simple conceptual errors often lead to type errors detectable by the type checker. Experimental results estimate the number of errors due to *detectable* type faults as something between 30% and 80% of all program errors [24].

- Assigning types to program variables and operations provides a succinct, intelligible form of program documentation, making programs easier to read and understand. Moreover, the type checker certifies that the type documentation is consistent.[2]

- Type information is useful in program optimization. Type information may permit a compiler to choose an optimal representation for some given data structure. Likewise, static analysis may prove that certain run-time checks are unnecessary, and therefore may be eliminated.

In contrast, the dynamic typing strategy detects type faults during program execution. All of the primitive operations check each argument before using it in a computation. If an argument is inappropriate, the operation signals a run-time error.

[1] The type checking process is almost always incorporated in the compilation or interpretation stage. Conceptually, however, the type checker can be considered a separate entity.

[2] Even in dynamically typed languages, a programmer could certainly provide type information. The difference is that a static type checker also checks the type information for consistency.

Unlike a statically typed language, there is no type checking guardian that excludes programs; any syntactically correct program can be executed. Dynamically typed languages are much more flexible and expressive than statically typed ones because they do not exclude any programs. The absence of restrictions on programs that can be executed provides the expressive power in dynamically typed languages. The expressiveness of dynamically typed languages offer the programmer:

Flexibility No type checker can decide whether or not an arbitrary program will generate any undefined applications. Therefore, the type checker must err on the side of safety and reject some programs that do not contain any semantic errors.

Generality: Some abstractions cannot be encapsulated as procedures because they do not type check. As a result, a programmer may be forced to write many different instances of the same abstraction. In Pascal, it is impossible to write a sort procedure applicable to arrays of different lengths; a separate sort procedure must be written for each array index set. In **ML**, it is impossible to write the polyadic `taut` function in example 2.9. A separate function must be written for each arity.

Semantic Simplicity The type system is a complex set of syntactic rules that a programmer must master to write correct programs. Otherwise, he will repeatedly trip over the syntactic restrictions imposed by the type checker.

Programmers accustomed to dynamically typed languages are reluctant to sacrifice this flexibility, even for the cited advantages of static typing.

The lack of flexibility in statically typed languages is inherent in any "interesting" programming language. An informal proof of this fact requires a more precise understanding of what is meant by " interesting programming language". Informally, the crucial property of a programming language is that it specifies a computation. The following definition solidifies this intuition.

> **Definition 1.1** *(What is a programming language?)* A programming language $\mathcal{L}$ is a triple $\langle L, V, [\![\cdot]\!] \rangle$. The first component, L, is a set of *terms*. The set L is called the *syntax*, or the *expressions* of $\mathcal{L}$. The second component, V, is a set of *values*. The third component, $[\![\cdot]\!]$ is a mapping from syntax to values. This component is called the *semantics*

of the language. Notationally, one indicates $[\![e]\!] = v$ to indicate that the semantics of expression e is the value v.

Intuitively, the expression e computes to the value v.

These definitions accommodate a great many programming languages. To narrow the possibilities a bit, this thesis focuses on "interesting" programming languages.

Definition 1.2 *(What is "interesting" ?)* An "interesting" programming language has the following features:

1. For any $e \in L$, there is a $v \in V$ such that $[\![e]\!] = v$. In other words, all terms have a value.
2. The set V contains at least three distinguished values: one for "erroneous" computations,one to indicate "non-terminating" computation, and some distinguished value for branches (see item 3). . Call the erroneous value **wrong**,the non-terminating value $\perp$, and the branch element T. Furthermore, there is at least one element $P_w \in L$ such that $[\![P_w]\!] = \textbf{wrong}$, and at least one element $P_r \in L$ such that $[\![P_r]\!] \neq \textbf{wrong}$.
3. For every $P, P_1, P_2 \in L$, $P' \in L$, where

$$P' \texttt{if } P \texttt{ then } P_1 \texttt{ else } P_2$$

and

$$[\![P']\!] = \begin{cases} \perp & [\![P]\!] = \perp \\ [\![P_1]\!] & [\![P]\!] = T \\ [\![P_2]\!] & \text{otherwise} \end{cases}$$

4. The set of programs $B = \{P \in L \mid [\![P]\!] \neq \perp\}$ is recursively enumerable (r.e.) but *not* recursive.

Informally, an "interesting programming language" is a programming language that permits non-terminating computation, branching, and has at least one program that has a run-time error. Furthermore, an "interesting programming language" has an undecidable halting problem.

The definition of interesting yields the following simple fact.

Lemma 1.1 *(The set of "Good" programs is not recursive)* The set G of all "good" programs, that is $G = \{P \in L \,|\, [\![P]\!] \neq \mathbf{wrong}\}$, is not recursive.

Proof Suppose G is recursive, using decision procedure D. Then, let $P \in L$ be any arbitrary program. By items 3 and 2, we can construct P'

$$P' = \texttt{if}\ P\ \texttt{then}\ P_w\ \texttt{else}\ P_w$$

Using D, we can decide if $[\![P']\!] = \mathbf{wrong}$. But, $[\![P']\!] = \mathbf{wrong}$ iff $[\![P]\!] \neq \bot$. So, we can use D to decide B. By assumption, B is not recursive. So, by contradiction, G is not recursive. □

Based on this preliminary framework, the analysis of statically typed languages and dynamically typed languages proceeds with definitions intended to capture the informal notions previously mentioned

Definition 1.3 *(Static type systems)* Let $\mathcal{L} = \langle L, V, [\![\cdot]\!]\rangle$, with type system $\mathcal{T}$. Let $L_W \subset L$ be a recursive set, called the *well-typed expressions*, such that for any $e \in L_W$, $[\![e]\!] \neq \mathbf{wrong}$. Then the programming language $\langle L_W, V, [\![\cdot]\!]\rangle$ is a statically typed language.

The key notion of *statically* typed language is that the set of expressions in the language is smaller than the original language.

Note that a statically typed language has the same semantics and set of values as the original programming language.

We can now easily prove:

Theorem 1.1 *(Fundamental Theorem of Static Typing)* Let $G = \{P \in L \,|\, [\![P]\!] \neq \mathbf{wrong}\}$. Then $G - L_W \neq \emptyset$. That is, there is a $P \in G, P \notin L_W$.

Proof By definition 1.3, if $P \in L_W$, $[\![P]\!] \neq \mathbf{wrong}$, so $P \in G$. This means $L_W \subseteq G$.By definition 1.3, if $P \in L_W$, $[\![P]\!] \neq \mathbf{wrong}$, so $P \in G$. This means $L_W \subseteq G$. By lemma 1.1, G is not recursive. But, by definition 1.3, the subset L_W is recursive. So $L_W \neq G$. Therefore, There is always some $P \in G - L_W$. □

We state the theorem informally as

> For any interesting programming language, a statically type discipline necessarily excludes some good programs.

As a corollary, we note

> For any interesting programming language, there will always be some programs that user must rewrite to accommodate a static type checker.

The basic philosophy of static typing is a conservative one. Statically typed languages exclude some programs that have no run-time to ensure that there are no programs that do contain errors. In practice, however, not even this principle is sacrosanct. To avoid massive inconvenience in certain cases, some static type systems permit programs that might possibly "go wrong" to be certified as type correct. These languages rely on features of the run-time system to avoid disasters. The classic case is ordinary division. The **divide** function takes two numbers as input. The second number, is supposed to be non-zero. Many static type systems, however, permit the type of `divide` to be $\mathsf{number} \times \mathsf{number} \rightarrow \mathsf{number}$. In such systems the expression `divide`$(5, 0)$ passes the type checker, even though a misapplication is present. Without this laxity by the type checker, programmers would be forced to write in their own run-time checks every time the `divide` operation is used. For type systems that permit some run-time checking, the term "type fault" is defined as "whatever the type checker says it is". In the context of this thesis, *any* inappropriate application is called a *type fault*.

In spite of the lack of expressiveness, statically typed languages still possess the three advantages cited previously. Clearly, one would certainly like to retain the expressiveness of dynamic typing but still gain a measure of the advantages of static typing. This desire is the motivation for this thesis. In this thesis, we introduce a concept we call *soft* typing as an approach to gaining the best of both static and dynamic typing. To do so, we alter our view of the function of a type-checker. Soft typing views the type-checker as a *program transformation* tool. From our point of view, a dynamically typed languages possess statically typed sublanguages. A given statically typed sublanguage (may) have useful properties. In particular, dynamically typed programs that are *not* statically correct *may* indicate the presence of errors. Also, the implementation and documentation benefits of static typing can be extended to the entire dynamically typed language if it is possible to transform an arbitrary dynamically typed program into a program in the statically typed sublanguage *that* *has the same meaning as the original program.* The goal of soft typing is to discover useful static sublanguages and methods for transforming programs.

A soft type checker inspects program fragments for potential run-time errors in the same fashion as a static type checker. In addition, the soft type checker will transform non-conforming programs into equivalent statically correct programs by inserting explicit run-time checks.

We design a type checker to be used as an auxiliary tool, not a filter. In a soft typing system, a programmer programs in his accustomed dynamically typed style, and prior to execution, he runs the soft type checker. The checker infers types wherever possible, and flags the places in the program where no type can be inferred, or where a possible run-time error may occur. In addition, the soft type checker inserts the appropriate dynamic checks. After the programmer inspects the places highlighted by the soft checker (and possibly makes corrections), he runs his program. Moreover, this paradigm does not require that a programmer use only one type checker. A soft typing system may be served by a *suite* of type checkers. The key point of this methodology is that the type checker does *not* stop the programmer from executing a program that it cannot guarantee. In this way, the expressiveness of dynamic type systems is maintained. The programmer, however, receives the error checking feedback from the type checking process. Furthermore, since the ultimate output of the soft type checker is a statically type correct program, the implementors can use the types to compile more efficient code.

1.1 Design Criteria for a Soft Type Checker

As presented, a soft typing system can be used as either a pure static system or a pure dynamic one. Programmers who insist on static typing would run the soft type checker and refuse to run programs not blessed by the type checker. Alternatively, programmers who are zealously attached to dynamic checking may choose to ignore the output of the type checker. Since soft typing encompasses both programming styles, one might assume that *any* type checker can serve as a soft type checker. There are both technical and pragmatic issues, however, that mitigate against the use of many common type checkers.

Before we begin our analysis, we state the design criteria for a soft typing system:

1. No syntactically correct programs are excluded.

2. Run-time safety is assured by run-time time checks if such checks cannot be safely eliminated

3. The type checking process must be unobtrusive

Qualitatively, we formulate two general principles to characterize an unobtrusive type system.

Minimal-Text Principle The system should require little extra text (in the form of declarations) for the type checker to function. In particular, the type checking system should function in the absence of programmer-supplied type declarations.

Minimal-Failure Principle The checker must pass "a large fraction" of dynamic programs that will not produce an execution error. We realize that not all programs can be certified, but a type checker that "cries wolf" too often is a burden rather than an aid.

As our first pragmatic observation, we note that any programming system that requires the programmer to declare types qualifies as unduly intrusive. We might be willing to provide *some* type information on an interactive basis to improve the accuracy of the typing process, but any system that requires declarations for routine program functions is not what we want. There are several static type checkers that do not require type information from the user. The most notable such checker is a component of many modern functional languages. ML is the canonical example of such a language [34, 17]. Other members of the family include Hope[7] and Miranda[51]. These languages may be viewed as variants of the simply typed lambda calculus.[3] Barendregt[4] calls this family of languages the *implicitly typed lambda calculi*, or the *Curry* family. The key property of the Curry family is that the type checking method does not require type declarations.

Another feature of the implicitly typed languages that plays a major role in soft typing is *parametric polymorphism*. Parametric polymorphism describes program procedures that are re-usable in regular ways. Routines with this property accept as input a wide variety of different, but structurally similar data, and the output type

[3]Some researchers believe that these languages forced fundamental extensions to the typed lambda calculus to accommodate polymorphic function definitions, but Wand's[52] research indicates this is not necessary.

of such a routine depends on the input type in a uniform way. The type behavior of parametrically polymorphic routines can be uniformly characterized by type variables (the type *parameter*). A few examples will clarify this concept.

Example 1.1 The identity function, $\lambda x.x$ has type $\alpha \rightarrow \alpha$. This means that the identity function can be thought of as having many different types. For example, if α is assigned the type bool, then the identity has type $\mathsf{bool} \rightarrow \mathsf{bool}$. Likewise, $\alpha = \mathsf{int}$ produces the type $\mathsf{int} \rightarrow \mathsf{int}$ as a type for the identity function.

Example 1.2 The **append** function on lists takes two lists, and joins one to the other. The function works on two lists of integers, giving a list of integers. It works equally on lists of booleans, yielding a list of booleans. The **append** function additionally accepts two lists of floating point numbers, returning a list of floating point numbers. These behaviors are instances of a quite general pattern, that can be characterized using type variables. The type of **append** is $(\mathsf{list}(\alpha), \mathsf{list}(\alpha)) \rightarrow \mathsf{list}(\alpha)$.

As stated earlier, Curry style languages supports both declaration free type checking and parametric polymorphism. A natural consideration, then, would be to base a soft typing system on a Curry style type checker. Unfortunately, there are far too many correct dynamically typed programs that do not pass the Curry style type checker. Furthermore, many of these examples are simple enough to be deemed "programming clichés". These clichés can be partitioned into two different classes. One class reflects the more flexible use of functions in non-uniform ways. The second class indicates the natural occurrence of recursive types.

1.1.1 Non-Uniformity

The non-uniformity common in dynamically typed languages is best illustrated with some examples.

Example 1.3 Consider the function

$\lambda x.$ `if` x `then` 1 `else nil`

The function takes a simple boolean value and either returns an integer 1, or the empty list. Clearly, no run-time error results as long as x is a boolean. Existing Curry style systems fail to assign a type to this function because the `if` expression is required to have the same type in both its `then` branch and its `else` branch. In these static systems, 1 and `nil` belong to different types.

Example 1.4 A similar problem exists with the function:

$(\lambda x.\texttt{cons}\,(1, x))\,[\texttt{true}, \texttt{false}]$

The expression builds the list $[1, \texttt{true}, \texttt{false}]$. Again, no run-time error results from the execution of this expression. The elements of the construction, however, belong to different types. Some are integer, some are boolean. In Curry static systems, non-uniform lists are systemically rejected.

The intuitive reason for the failure of these examples derives from the view of type embedded in the Curry tradition. Curry style languages ascribe to the view that a value has exactly one type. The Curry principle was popularized as the Hoare-Dahl-Dijkstra view of types[16].

Every value belongs to one and only one type

In contrast, programmers using a dynamically typed language (like Scheme) expect values to be uniquely constructed, but they frequently define functions that work on different kinds of data. For example, Scheme programmers conceives that the value **nil** (empty list) and values constructed with **cons** are fundamentally different. The programmers may, however, write a function that will work on either nil values or cons values. Stated another way, scheme programmers expect data values to have unique fundamental types based on how they were constructed. Program functions, however, may be defined to work on *unions* of these fundamental types. From a Scheme point of view, functions that work on list values are working on values of type $\mathsf{nil} \cup \mathsf{cons}$. The ML system, on the other hand, views both **nil** and **cons** values as being of type list. Consequently, there are no functions defined that work only on cons values, and no functions that work on nil or integers.

The concept of *union* typing confers reasonable types on both example 1.3 and example 1.4. For example 1.3, the function has type $\mathsf{bool} \rightarrow \mathsf{int} \cup \mathsf{nil}$. The reader also observes that 1 having type int, also has type $\mathsf{int} \cup \mathsf{nil}$, and similarly `nil` has both type nil and $\mathsf{nil} \cup \mathsf{int}$, so some degree of "uniformity" is maintained. Thus, parametric polymorphism and union typing integrate smoothly.

Example 1.4 also maintains the same smooth integration. One observes that the type of the expression is $\mathsf{cons}\,(\mathsf{int} \cup \mathsf{bool})$. In this manner, we have "uniform" (non-empty) lists of $\mathsf{int} \cup \mathsf{bool}$ values.

One additional advantage conveyed by union typing derives from the ability to make finer grained type distinctions. For example, the **hd** function, which extracts the first element of a non-empty list, has Curry type $\alpha\ \mathsf{list} \to \alpha$. This type is somewhat misleading. One might believe that **hd** produces some value on **nil**, when, in fact, the run-time system signals an error. The ML community refers to this phenomenon as a *partial* function. This situation does not occur in the union type framework because nil and cons are distinguishable. The **hd** function has type $\alpha\ \mathsf{cons} \to \alpha$.

The qualitative analysis of union types presents two distinct advantages in the soft typing context. One, many common dynamically typed programs not checkable by a Curry style type checker are type checkable with union types. Two, some of the run-time errors introduced in ML-like systems can be detected by a type checker supporting union types.

1.1.2 Recursive Types

Functional languages encourage a programmer to use recursion, so it is not surprising that recursive types occur with some frequency. The following examples illustrate typical programs that suggest recursive types. For the purpose of these examples, recursive types are notated by equations.

> **Example 1.5** The classic example of a program needing recursive types appears in the *Y combinator*, or fixed point operator:
>
> $$Y = \lambda F.\lambda x.(F\ (x\ x))\lambda x.(F\ (x\ x)).$$
>
> Curry style type checking systems, lacking recursive types, cannot assign a type to the subexpression $(x\ x)$. Consequently, no type can be assigned for Y. With recursive types, the type for x is $\alpha = \alpha \to \beta$, and the type for Y is $(\alpha \to \alpha) \to \alpha \to \alpha$.

When used in conjunction with union types, recursive types express the types of many common dynamically correct programs.

Example 1.6 Define a recursive function P as follows:

$$P \quad = \quad \lambda x. \quad \texttt{if } x = 0 \texttt{ then } 1 \texttt{ else cons}(P(x-1), \texttt{nil})$$

The output type for P is a recursive union type: $\tau_P = \mathsf{int} \cup \mathsf{cons}(\tau_P)$. Intuitively, the type indicated by τ_P is either an integer, or a non-empty list of integers, or a non-empty list of non-empty list of integers, and so on. A Curry-style checker cannot assign a type to P.

These examples are legal programs in any dynamically typed language — no run-time error is ever generated. To produce "equivalent" functions in a static language like ML, one must resort to defining new data types, as well as changing the program text to accommodate the newly defined constructor functions. This extra textual burden on the programmer is a violation of the minimal-text principle of the soft type design criteria.

This analysis indicates that recursive types, particularly in conjunction with union typing, are an integral part of dynamically typed programming style.

Having decided on unions, recursion and parametric polymorphism as desirable properties the technical challenge is a two-fold question:

1. Can unions, recursion, and parametric polymorphism be accommodated in an automated type assignment method ?
2. Are these constructs sufficient to insure that all dynamically programs can be automatically transformed into statically correct ones ?

1.2 Summary of Thesis Goals

The main goal of this thesis is to establish design criteria and methods for designing soft typing systems. We narrow our immediate goal to the design of a soft typing system for a family of prototypical programming languages. In particular, this thesis will establish, by construction, that there is at least one soft type system for the chosen language family satisfying the criteria of section 1.1. To satisfy this goal, the soft type system must exhibit three constructs:

A type language suitable for soft typing: By the minimal-failure principal, the type language must be designed with sufficient features to describe several typical dynamically typed language behaviors, including:

- Parametric Polymorphism
- Union Types
- Recursive Types

A type reconstruction algorithm for the type language: To meet the minimal text requirement, the programmer must be able to avoid type declarations.

A transformation method : The type system must admit an automatic transformation method for programs that are not in the associated statically typed language.

1.3 The General Plan

The exposition of the research results directed towards meeting the goals outlined in section 1.2 are divided into five chapters.

This current chapter describes the motivation for soft typing. It contains our design principles, as well as examples justifying our choices. It also summarizes the goals of the research.

Chapter 2 first establishes a family of programming languages. In order to make precise statements about the properties of the system, the specification of the programming language must include a formal semantics. The programming language family will be dynamically typed. Given a programming language family, the design of the a type system suitable for soft typing is the next task. The choice of type constructors is influenced by the data constructions of the language as well as the union, recursion and parametric constructions suggested by the considerations of section 1.1. The type language, together with the programming language serves to define a statically typed sublanguage. The statically typed sublanguage requires a type inference system, and a semantics for types such that the inference system and type semantics are sound with respect to the programming language semantics.

With a tentative type system in hand, Chapter 3 proceeds to investigate the automatic type assignment process. The type assignment problem has some consequences for the type language design. The type language presented in chapter 2 is too general. To automate the assignment process requires a mild condition on the use of unions types. Chapter 3 contains some example deductions for evidence of the mildness of the restriction. At this point, the design of the statically typed sublanguage is complete.

The final component of a soft typing system must address the is the transformation of dynamically typed programs that are *not* statically type correct into programs that are statically type correct, and have the same meaning as the original dynamically typed program. Chapter 4 indicates a method for our chosen programming languages and their associated statically typed sublanguages.

Chapter 5 contains a summary of the results, and a section on related work, to provide perspective for this work. Additionally, some thoughts on related research appear in this chapter.

Chapter 2

A Family of Programming Languages, Type Languages, and Type Inference Systems

This chapter concentrates on the specification of a programming language family, and the design of an associated family of type systems suitable for softly typing members of the language family. To simplify the analysis, we restrict ourselves to functional languages, but we expect to be able to extend our ideas to non-functional constructs in a similar manner to Tofte[50] or Duba[19].

The family of functional languages adopted herein is parameterized by the set of constants, or primitives. The set of constants, in turn, is guided by the data used by the program. Program data is a design consideration, and, as such is programmer-directed. Section 2.1 explains this dependency in more detail.

For a given set of constants, the programming language Exp is syntactically fixed. Section 2.2 defines the language syntax. Exp is a simple functional language with a `let` construct.

A language specification must also include semantics. In fact, we must ultimately give *two* semantics for members of our language family: one semantics when the language is viewed as statically typed, and one semantics when the language is viewed as dynamically typed. In this chapter, we concentrate on the statically typed language, and defer the dynamically typed semantics till chapter 4. The statically typed semantics for Exp expressions appear in section 2.3. The semantics are denotational.

Once the programming language is fixed, section 2.4 describes the design considerations for a type language to accommodate "most" Exp programs. The formal definition for the type language appears later in the section. The set of types proposed in this section is called the regular types.

The semantics for type language, appearing in section 2.5 employs the ideal model for types. The ideal model is explained in [31, 32]. A brief review of these results prefaces the actual semantic definition for the type language. We show that the semantic definition is well defined with respect to the the fixed point components.

Section 2.6 connects the programming language and the type language with a set of inference rules. The inference rules for ML are contrasted with the inference rules for the type language described in this chapter. In addition, section 2.6 contains the soundness proof for the regular type inference rules.

Finally, section 2.7 contains some example inferences to illustrate the system.

2.1 A view of program data

This section begins with a short methodological digression. Statically typed languages encourage a design philosophy that is somewhat different from the typical design philosophy encountered in dynamically typed languages. Advocates of statically typed languages cite the design of types as a crucial part of the program design process.

In contrast, programmers accustomed to dynamically typed languages focus on the design of the *data*, instead of the types. In some ways, this method is simpler. For example, a programmer may decide that his program needs integers, fractions and floating point numbers. His program may contain functions that act on various combinations of these data. A static type programmer, however, must decide how the various data must be put together into disjoint unions. That is, a function that acts on both integers and fractions has a different type from a function that operates on only integers. Consequently, from a type design point of view, the programmer must consider the effect of adding the type "integers or fractions" — a consideration that is absent from the data design viewpoint. The data designer assumes that the dynamic type system will provide the appropriate checking for types that are combinations (that is, unions) of different data types.

The types designed in this section is intended to support the paradigm of data design. Furthermore, the data required by a program also influences the programming language itself, as the appropriate primitives for data construction and analysis must be provided as constants in the language. For this reason, the data language and type language are quite similar. Furthermore, the importance of the data description system influences us to break tradition with the standard typing literature and describe the data language ahead of the programming language.

2.1.1 First order data

Even though functional programming languages emphasize the importance of functions as data values, the first order data is equally important. The view of first order data taken here is the *constructive* view established in Cartwright[10]. A similar view is taken by Mishra and Reddy[35]. The constructive view considers all first order data to be built from constructors. Furthermore, a constructor may restrict its arguments to be of a certain construction (in a crude form of dynamic typing). Finally, for each constructor, there is a unique set of *selectors* corresponding to each argument position of the constructor. The specification of the constructors, the valid constructions for arguments to the constructor, and the names of the associated selectors make up a constructive definition, or, *first order specification.* We consider the first order specification to be an integral part of the language specification. The notation in [10] gives a nice, succinct notation for the necessary information in a first order specification.

The syntax for a constructive definition, in full generality, can be described as a series of statements of the form:

$$\texttt{constructor}\ \texttt{C-name}[\texttt{S-name} : \texttt{C-name}[+\texttt{C-name}]^*]^*$$

The `C-name` and `S-name` are syntax for the names of constructors and selectors (as one might suppose).

An example will help clarify these concepts. To simplify matters, example 2.1 ignores the selector components at first, and then introduces the full first order definition later.

Example 2.1 *(Constructive data definitions)* Suppose the data needed consists of non-negative integers and lists. Then consider the following:

$\texttt{constructor}\ \texttt{0};$
$\texttt{constructor}\ \texttt{suc}(\texttt{0} + \texttt{suc});$
$\texttt{constructor}\ \texttt{nil};$
$\texttt{constructor}\ \texttt{cons}(\texttt{0} + \texttt{suc}, \texttt{nil} + \texttt{cons})$

The above definition indicates that `0` is a constructor requiring no arguments, and the `suc` constructor takes a single argument. Furthermore, the argument to `suc` must have been constructed with either a `0` or a previous application of the `suc` constructor. Similarly, `nil` is a constructor requiring no arguments, and `cons` requires two arguments. The first argument must be constructed via `0` or `suc`. The second argument's construction must be either a `nil` or some other `cons` construction. Some sample data elements:

`0`, `suc(suc(0))`, `nil`, `cons(0,nil)`,
`cons(suc(suc(0)),cons(0,nil))`

Some *invalid* data elements are:

`suc(nil)`	`nil` is not a valid construction for `suc`'s argument
`cons(nil,nil)`	`nil` is not a valid construction for `cons`'s first argument
`cons(0,0)`	`0` is not a valid construction for `cons`'s second argument
`suc`	No argument to `suc`, so usage is wrong.

Additional important elements of a constructive definition involve primitives called *selectors*. A selector function extracts a given argument from a construction. The constructor involved, of course, must have an arity greater than zero. To illustrate:

`constructor` `0`;
`constructor` `suc`(`pred:0` ∪ `suc`);
`constructor` `nil`;
`constructor` `cons`(`hd:0` ∪ `suc`, `tail:nil` ∪ `cons`)

A selector selects a given argument for its associated constructor. The effect of selectors works like:

$$\begin{aligned} \texttt{pred}(\texttt{suc(suc(0))}) &= \texttt{suc(0)} \\ \texttt{hd}(\texttt{cons(suc(0),cons(0,nil))}) &= \texttt{suc(0)} \\ \texttt{tl}(\texttt{cons(suc(0),cons(0,nil))}) &= \texttt{cons(0,nil))} \end{aligned}$$

Note that all constructors have a fixed arity.

2.1.2 Higher order data

One of the distinctive features of a functional language is the treatment of functions as first class objects. That is, functions may be passed as arguments to other functions, and may be returned as the output value of a function. In this fashion, functions are treated just like first order data objects.

Functions differ from first order data objects by the absence of a "constructor". Functions are constructed by the lambda abstraction operation of the programming language defined in section 2.2.

The presence of functional data introduces one minor difficulty. Suppose a programmer designs a data constructor, one of whose arguments should be a function. What should be done ? The constructive mechanisms introduced so far do not accommodate functions as data. The solution adopted in this research introduces the $\rightarrow$ symbol as a *pseudo*-constructor. No selectors correspond to $\rightarrow$, but the symbol may be used in the same way as any other constructor symbol in the first order data specification.

Example 2.2 *(Constructive definition, including functions)*

```
constructor   0;
constructor   suc(pred:0 ∪ suc);
constructor   nil;
constructor   cons(hd:0 ∪ suc ∪ cons ∪ nil∪ →, tail:nil ∪ cons)
```

Example 2.2 indicates that `cons` may take any kind of value whatsoever as its first argument.

2.2 The Programming Language Exp

The prototypical programming language family is related Exp family introduced by Milner[17, 34]. Our version of Exp lacks the `if` construct. Branching is accomplished by special constants. Note that the language family is parameterized by the set of allowed constants.

Definition 2.1 *(The programming language)* Let x range over a set of variables Vars, and c range over a set of constants K

$$
\begin{array}{rcl}
L & ::= & x \mid \\
 & & c \mid \\
 & & \lambda x.L \mid
\end{array}
$$

$$(L\ L) \mid$$
$$\texttt{let}\ x = L\ \texttt{in}\ L$$

The set of constants for a given member of the Exp family depends on the first order specification. Let S be a specification for a collection of first order data. That is, suppose S to be given as a set of `constructor` statements as illustrated in examples 2.1,2.2. Then, S contains a set of constructor names M and selector names Q. Let $P = M \cup Q$. Then $P \subset K$.

The set K must also contain a suite of `case` functions. The set of `case` functions also depends on the first order specification in the following way. Each `case` function is subscripted by a non-empty tuple of unique elements of M. For example, using $S = \{\texttt{0}, \texttt{cons}, \texttt{nil}, \texttt{suc}\}$, as in example 2.1, then $\texttt{case}_{\texttt{0},\texttt{suc}}$ and $\texttt{case}_{\texttt{cons},\texttt{nil},\texttt{0}}$ are valid `case` functions. These `case` functions are the branching mechanisms for the Exp family. Note that `if` can be expressed as $\texttt{case}_{\texttt{true},\texttt{false}}$, whenever the language has booleans.

Throughout this work, `case` expressions are syntactically sugared as follows:

Example 2.3 *(`case` syntactic sugar)* The expression

$$\begin{array}{l} \texttt{case}\ x\ \texttt{of} \\ \quad c_1 : e_1 \\ \quad c_2 : e_2 \\ \quad\quad \ldots \end{array}$$

is syntactic sugar for:

$$((\texttt{case}_{c_1,c_2}(\lambda\ x.e_1)(\lambda\ x.e_2)\ldots)\ x)$$

In fact, the user does not normally program with the `case` functions, but rather resorts to the syntactic sugar version instead.

The set K may include some additional constants such as `if` or `fix` as long as the semantics are specified.

To summarize, a programming language is completely specified by definition 2.1 and a set K. The set K is determined by a first order specification S, and an (optional) set of auxiliary constants.

2.3 Semantics of Exp

To give meaning to the programming language requires a semantics. The semantics for Exp employs the well understood denotational model illustrated in [45, 41, 42] and many others.

The first requirement for any denotational semantics is a data domain, as described in section 2.3.1. The remaining sections comprise the denotational definition of Exp. Section 2.3.2 defines the semantics for the language constructs, and section 2.3.3 gives the semantics of the set of constants described in section 2.2.

2.3.1 The Data Domain

To define the relevant domain D, let Triv be a one element domain $\{*, \perp\}$. Let $\mathbf{W}$, $\mathbf{F}$, and D^0 abbreviate Triv. Let the set of data constructors be M. Let $\times$ (infix) denote the n-ary *smash product*[4] for domains. Let D^k abbreviate $D \times D^{(k-1)}$, and let D_{C_i} abbreviate $D^{\mathrm{Arity}(\mathsf{C}_i}$. Let the $\oplus$ symbol be infix notation for the n-ary *coalesced sum*[5]operation for domains. Let $\rightarrow$ (in this context) be the continuous function domain constructor. Define D as the least fixed point solution of the following equation:

$$D \cong (D \rightarrow D) \oplus D_{\mathbf{c_1}} \oplus D_{\mathbf{c_2}} \oplus \ldots D_{\mathbf{cm}} \oplus \mathbf{W} \oplus \mathbf{F} \tag{2.1}$$

The domain $\mathbf{W}$ (a copy of Triv) appearing at the end of the summation is not associated with any constructor (or function). The purpose of this summand is to fatal errors[6]. The value **wrong** abbreviates $\mathrm{In}_{\mathbf{W}}(*)$. The domain $\mathbf{F}$ models run-time errors that are caught by the run-time system. The value **fault** abbreviates $\mathrm{In}_{\mathbf{F}}(*)$. The value **fault** is distinguished from **wrong** in that **fault** is non-fatal. The detailed treatment of the **fault** value appears in chapter 4.

---

[4]Smash product is defined by

$$D_1 \times D_2 = \{\langle d_1, d_2\rangle \mid d_1 \neq \perp \in D_1, d_2 \neq \perp \in D_2\} \cup \{\perp\}$$

See [41]

[5]The coalesced sum is defined as

$$D_1 \oplus \ldots D_n = \{\langle d_1, 1\rangle \mid d_1 \neq \perp \in D_1\} \cup \ldots \{\langle d_n, n\rangle \mid d_n \neq \perp \in D_n\} \cup \{\perp\}$$

[6]Some of our colleagues have suggested "core dump" as the name for this value

All program expressions take values in D.

There are two auxiliary domains:a syntactic domain Id of *identifiers* and an environment domain, **Env** : $\mathrm{Id} \to D$.

2.3.2 Semantics of the Programming Language Constructs

The semantics for Exp, excluding the constants, is conceptually straightforward, but can be notationally awkward. Consequently, we introduce the following conventions and definitions to overcome some of the clumsiness.

First, functions depending on different conditions appearing as

$$f(x) = \begin{cases} \mathrm{val}_1 & \mathrm{condition}_1 \\ & \vdots \\ \mathrm{val}_k & \mathrm{condition}_k \end{cases}$$

describe a function that tests the conditions in sequential order. That is, $\mathrm{condition}_1$ is tested. If it is true then val_1 is the value of the function. Only if $\mathrm{condition}_1$ is false is $\mathrm{condition}_2$ considered. In general val_k can only be the value of f if $\mathrm{condition}_k$ is true, and $\mathrm{condition}_i, 1 \leq i < k$ is false.

Semantic definitions depend on environments, so to facilitate the notation for environments derived from previous environments we introduce the following definition.

Definition 2.2

Let $\eta : \mathrm{Id} \to D$ be an environment function, that is, an assignment of values to free variables. Then

$$\eta[x = v](y) = \begin{cases} v & \text{if } y = x \\ \eta(y) & \text{otherwise} \end{cases}$$

Some aspects of the semantics depend on the various components (summands) indicated in defining equation 2.1. Intuitively, the domain D consists of various kinds of "tagged" elements. The "tagging" is represented formally as the n-ary coalesced sum operation. Semantic definitions require both "tagging" "untagging" auxiliary functions. The formal treatment of "tagging" uses injection and projection operations.

Definition 2.3 *(Injections)* Let

$$D = D_1 \oplus \cdots D_k \cdots D_n$$

and let $d_k \in D_k$. Then define

$$\mathrm{In}_{D_k}(d_k) = \begin{cases} \bot & d_k = \bot \\ \langle k, d_k \rangle & \text{otherwise} \end{cases}$$

The Out operation is a partial inverse for In When the inverse is not defined, the value **wrong** results.

Definition 2.4 *(Projections)* Let

$$D = D_1 \oplus \cdots D_k \cdots D_n$$

Then for $d \in D$

$$\mathrm{Out}_{D_k}(d) = \begin{cases} \bot & d = \bot \\ v' & d = \mathrm{In}_{D_k}(v') \\ \textbf{wrong} & \text{otherwise} \end{cases}$$

With notation set, the semantics for Exp (except constants) goes as follows:

Definition 2.5 *(Semantics for Exp)* . Let **Env** be the domain of environments Vars $\to D$, that is, assignments of free variables. Let Exp refer to the (syntactic) domain of programming language expressions. The semantic function $[\![\cdot]\!]$: Exp $\to$ **Env** $\to D$ is defined by the semantic equations

$$\begin{array}{rcl}
[\![x]\!]\eta & = & \eta(x) \\
[\![\lambda x.e]\!]\eta & = & \mathrm{In}_{D\to D}(f) \\
 & & \quad \text{such that } f(v) = [\![e]\!]\eta[x = v] \\
[\![(e_1\ e_2)]\!]\eta & = & \begin{cases} v_1 & v_1 \in \{\textbf{wrong}, \textbf{fault}\} \\ v_1 & v_2 \in \{\textbf{wrong}, \textbf{fault}\} \\ v_1(v_2) & \text{otherwise} \end{cases} \\
 & & \text{where } v_1 = \mathrm{Out}_{D\to D}([\![e_1]\!]\eta), v_2 = [\![e_2]\!]\eta \\
[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!]\eta & = & [\![e_2]\!]\eta[x = v_1],\ \text{where } v_1 = [\![e_1]\!]\eta
\end{array}$$

2.3.3 Semantics for The Constants

Constants for Exp, called K in section 2.2, must include the constructors and selectors specified in the first order specification, as well as the set of `case` functions. In addition, a language designer may choose to include some other primitives beyond the required ones. This section describes the semantics for requisite elements of K, any additional functions must have separately defined semantics.

The semantics for all requisite constants includes a test for "appropriate" arguments. If the argument is not "appropriate", then the value is automatically **wrong**. The set of "appropriate" arguments, however, may include values in more than one summand of D. For the remainder of this chapter, a set of "constructors" includes the function pseudo-constructor $\to$.

Definition 2.6 *(Confirming)* Let $R = \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}$ be a set of constructors. The auxiliary function $\mathcal{C}_R$ verifies that its argument belongs to one of the summands specified by R. It is defined by the equation:

$$\mathcal{C}_R(d) = \begin{cases} d & d = \mathrm{In}_{D_{\mathtt{r}_1}}(d') \\ \vdots & \\ d & d = \mathrm{In}_{D_{\mathtt{r}_n}}(d') \\ \mathbf{wrong} & \text{otherwise} \end{cases}$$

where $D_{\mathtt{r}_i}$ is the summand of D associated with constructor $\mathtt{r}_i$.

We generalize the confirming notion to tuples of legal constructions. For $1 \leq i \leq k$, let each R_i be a set of constructors. Then we define:

$$\mathcal{C}_{R_1,\ldots,R_k}(\langle d_1, \ldots, d_k \rangle) = \\ \begin{cases} \langle d_1, \ldots, d_k \rangle & \text{for all } i, 1 \leq i \leq k \\ & \mathcal{C}_{R_i}(d_i) \neq \mathbf{wrong} \\ \mathbf{wrong} & \text{otherwise} \end{cases}$$

For each constructor c, the first order specification indicates both the intended arity of the construction, and the valid constructions for each argument position. The meaning of a constructor c, where $\mathrm{Arity}(c) > 0$, however, is a *curried* function. A constructor function waits for all of its arguments before yielding the constructed value.

Definition 2.7 *(Constructor semantics)* Let $n = \mathrm{Arity}(\mathtt{c})$ and let R_i be the set of valid constructions for the i-th argument of $\mathtt{c}$. Then

$$\mathcal{E}[\![\mathtt{c}]\!]\eta = \begin{cases} \mathrm{In}_{D_{\mathtt{c}}}(*) & n = 0 \\ \mathrm{In}_{D \to D}(K) & n \neq 0 \end{cases}$$

where

$$K = \lambda d_1 \ldots d_n.\mathrm{In}_{D_{\mathsf{C}}}(\mathcal{C}_{R_1,\ldots,R_n}(\langle d_1, \ldots, d_n \rangle))$$

The semantics for selectors are defined similarly.

Definition 2.8 *(Selector semantics)* Denote the selector for the i-th position of constructor c by s_c^i. Let $p_i(\langle w_1, \ldots, w_i, \ldots, w_n \rangle) = w_i$, that is, p_i is the standard cartesian projection for tuples, and let $C = \{C\}$. Then

$$\mathcal{E}[\![s_c^i]\!]\eta = \mathrm{In}_{D \to D}(S)$$

where

$$S(v) = \begin{cases} p_i(\mathrm{Out}_{D_c}(v)) & \mathcal{C}_C(v) \neq \mathbf{wrong} \\ \mathbf{wrong} & \text{otherwise} \end{cases}$$

Finally, the semantics of the `case` function family:

Definition 2.9 *(*`Case` *functions)*

$$\mathcal{E}[\![\mathtt{case}_{\langle \mathsf{c}_1, \ldots, \mathsf{c}_k \rangle}]\!]\eta = \mathrm{In}_{D \to D}(C)$$

where

$$C = \lambda d f_1 \ldots f_k. \begin{cases} f_1(d) & d = \mathrm{In}_{c_1}(d') \\ & \vdots \\ f_k(d) & d = \mathrm{In}_{c_k}(d') \\ \mathbf{wrong} & \text{otherwise} \end{cases}$$

2.4 A Type Language for Exp

As illustrated in the introductory chapter, dynamically typed programs often involve types that are easily expressed with the mechanisms of union and recursion, supported by parametric polymorphism. One natural approach might be to "throw in" a union operator and a recursion operator into a standard polymorphic type language. Syntactically, such an extension is easy. A type language, however, consists of more than just syntax. A type language must support some form of formal reasoning about both the relevant properties of types, and the connection between the programming language family and the associate type language. Standard type formalisms do *not* account for all the relevant properties of the union and recursion operators. Thus, the type language for the Exp language family must provide augmented reasoning methods as well as some new syntax.

Syntactically, the potential type language starts with a standard type language:

$$\begin{aligned} &\text{StdType} ::= \\ &\quad c_0 \\ &\quad x \\ &\quad c_n(\text{StdType}_1, \ldots \text{StdType}_n) \end{aligned}$$

where the c_i are type constructors of arity i. To this standard language, we add the union (+) operation and the `fix` operation, giving the syntax for the proposed type language as:

$$\begin{aligned} &\text{Type} ::= \\ &\quad c_0 \\ &\quad x \\ &\quad c_n(\text{Type}_1, \ldots \text{Type}_n) \\ &\quad \text{Type}_1 + \text{Type}_2 \\ &\quad \texttt{fix}\ x.\text{Type} \end{aligned}$$

One useful technique for reasoning about the properties of these types can be discovered by noting the resemblance between the syntax of type terms and the form of *regular trees*. The theory of regular tress is a generalization of the more well known theory of regular expressions. A regular expression represents a *set* of strings over some alphabet. There exist algorithms for deciding whether a regular expression represents the empty set, whether it represents a finite set, or, given two regular expressions, whether one is a subset of the other. In other words, all basic questions about regular sets are decidable. For a complete account of regular expressions, see [27].

As discussed above, the theory of regular expressions is based on sets of *strings*. The general regularity theory is based on sets of *terms* or *trees*. In fact, in the literature, this theory appears under various names: tree automata theory, regular tree expressions, and regular term languages to name a few. The most important feature of general regularity is that all of the decidability properties of the standard string theory are retained.

The usefulness of basing a type language on regular tree expressions derives from the presence of two operators: union and closure. A tree expression union operator corresponds to the same concept for types, and the closure operator corresponds to the least fixed point operation used in recursive type definitions. Furthermore, the decision procedure that answers the subset question for regular tree expressions constitutes a ready-made system for reasoning about sub types relations. A similar type system was developed by Mishra and Reddy[35]. A type system based on general regularity theory contains a rich set of types. For example, it is possible to distinguish even and odd integers, and to prove that even (or odd) is a subtype of regular integers.

Regrettably, the correspondence between types and regular trees is not perfect. The presence of functional values implies the necessity of functional types. Functional types do *not* have the same algebraic or order theoretic properties of first order types. Consequently, we cannot accommodate all of the desired type language properties within the framework of regular tree expressions.

To overcome this limitation, we introduce a formal inference system that subsumes the algebraic reasoning of regular tree mechanisms, but incorporates additional mechanisms for reasoning about functional types. The inference system is a composite of two methods. That is, the syntax of regular tree expressions is still acceptable for a language with functional types, but the nice algebraic algorithms for deciding the subset question do not apply if the function constructor appears in the type expression. Type expressions with arrows must be reasoned about in a less convenient manner.

To summarize, the type language design is based on the general theory of regularity. The syntax for types is uniform, but any syntactic reasoning about type expressions is dichotomous: If there are no functional types in the expressions, then standard algebraic regular tree expression methods apply. On the other hand, the presence of functional components in the type expressions require the less automatic inference methods.

To preface the type language design, section 2.4.1 reviews the important definitions imported from regular tree expression theory. The discussion in section 2.4.2 continues with the design of the type language. Section 2.4.3 further explains the anomaly for functional types.

2.4.1 A Review of Regular Tree Expressions

The primary source for this treatment of regular tree expressions, can be found in Géceg and Steinby[25]. Some further sources are Thatcher and Wright, Brainerd, Arbib, and Goguen [48, 5, 3, 47]. Some aspects of the theory may be derived from universal algebra, as in Burris[6] This section is expository. The proofs for these results can be found in the cited references.

A regular tree expression defines a set of terms, so one should clarify what is meant by a term.

> **Definition 2.10** *(Terms)* Let $F = \{f_1, \ldots, f_k\}$ be a set of function symbols. Every symbol $f_i \in F$ has a fixed arity, denoted $\text{Arity}(f_i)$. Let X be a set of variables disjoint from F. The set of terms $\Sigma(F, X)$ is the smallest set satisfying
>
> 1. $x \in \Sigma(F, X)$, for all $x \in X$
> 2. if $c \in F, \text{Arity}(c) = 0$, then $c \in \Sigma(F, X)$
> 3. if $f \in F, \text{Arity}(f) = n$, and $t_1, \ldots, t_n \in \Sigma(F, X)$, then $f(t_1, \ldots, t_n) \in \Sigma(F, X)$
>
> In universal algebra, $\Sigma(F, X)$ is the free term algebra over F.

Note that an element of $\Sigma(F, X)$ can be expressed as a tree with nodes labeled by F in the obvious way: The 0-arity elements of F are leaves, and some term $f_i(t_1, \ldots, t_n)$ is a tree whose root label is f_i and whose subtrees are the trees constructed from $t_1, \ldots, t_n$[7].

Throughout this section, "terms" and "trees" will be used interchangeably.

The *regular tree expressions* indicate sets of terms.

> **Definition 2.11** *(Regular Tree Expressions)* As before, let F be a set of function symbols, with given arities. Let X, again, be a set of variable names disjoint from F. Then the set of regular tree expressions $R(F, X)$ is as the smallest set satisfying

[7] In fact, some alternative presentations of this material focus more on the generation of trees. These presentations use regular tree *grammars* instead of regular tree expressions as the primary formalism. Both points of view are equivalent. Again, see Géceg and Steinby[25]

1. The special symbol $\emptyset$ is a regular tree expression
2. If $x \in X$, $x \in R(F,X)$. Similarly, if $f \in F, \text{Arity}(f) = 0$, then $f \in R(F,X)$.
3. If $f \in F, \text{Arity}(f) = n, E_1 \ldots E_n \in R(F,X)$, then $f(E_1, \ldots, E_n) \in R(F,X)$.
4. $E_1, E_2 \in R(F,X)$ then $(E_1 + E_2) \in R(F,X)$. The parentheses are often dropped.
5. $E \in R(F,X), x \in X$, then $(E)^{*x} \in R(F,X)$. This operation is called the (generalized) Kleene closure.

To define the set of terms indicated by a regular tree expression requires a few auxiliary concepts.

First, we define a set of terms with the same outermost function symbol. This set of terms is also called a set of grafts.

Definition 2.12 *(Set of grafts)* Let $f \in F$, with $\text{Arity}(f) = n$. Let $T_1, \ldots, T_n$ be sets of terms. Then define the set of terms $f(T_1, \ldots, T_n)$ as follows

$$f(T_1, \ldots, T_n) = \begin{cases} f & \text{Arity}(f) = 0 \\ \emptyset & \text{Arity}(f) > 0, \text{any of the } T_i = \emptyset \\ \{f(t_1, \ldots, t_n) \,|\, t_i \in T_i\} & \text{otherwise} \end{cases}$$

We generalize the graft notion to all regular expressions:

Definition 2.13 *(Substitution from a set)* Let $t \in \Sigma(F,X)$ be a term. Let T be a set of terms. Then define the set of terms $t[x \leftarrow T]$ recursively as:

1. $x[x \leftarrow T] = T$
2. $y[x \leftarrow T] = \{y\}$, for $y \neq x$
3. $c[x \leftarrow T] = \{c\}$, where $c \in F, \text{Arity}(c) = 0$
4. $f(t_1, \ldots, t_n)[x \leftarrow T] = f(t_1[x \leftarrow T], \ldots, t_n[x \leftarrow T)$, using the set of grafts notation from definition 2.12

The notion is extended to a set of terms in the obvious way:

Let A, T be a set of terms. Then

$$A[x \leftarrow T] = \{t \mid t \in t'[x \leftarrow T], \text{ for some } t' \in A\}$$

Definition 2.14 *(Kleene closure definition)* Let A be a set of terms. Then define the set of terms A^{*x} by using the following sequence of sets

a. $A_0 = A[x \leftarrow \emptyset]$

b. $A_{i+1} = A[x \leftarrow A_i]$

Define

$$A^{*x} = \sup A_i$$

Definition 2.15 *(The terms of a regular tree expression)* To each regular tree expression E, the associated set of terms $\Sigma(E)$ is defined as:

1. $\Sigma(\emptyset) = \emptyset$
2. $\Sigma(x) = \{x\}$ for $x \in X$,Similarly, if $f \in F$, Arity$(f) = 0$, then $\Sigma(f) = \{f\}$
3. If $f \in F$, Arity$(f) = n, E_1 \dots E_n \in R(F, X)$, then $\Sigma(f(E_1, \dots, E_n)) = f(\Sigma(E_1), \dots, \Sigma(E_n))$
4. $\Sigma(E_1 + E_2) = \Sigma(E_1) \cup \Sigma(E_2)$.
5. $E \in R(F, X), x \in X$, then $\Sigma((E)^{*x}) = \Sigma(E)^{*x}$

The following theorem, proved in [25], justifies the interest in regular tree expressions

Theorem 2.1 *(Decidability of Emptyness,Finiteness, and Subset)* Let E be a regular type expression. There is an algorithm to determine if $\Sigma(E)$ is empty, or if it is finite. Furthermore, if E' is another regular tree expression, then there is an (algebraic) algorithm to determine if $\Sigma(E) \subseteq \Sigma(E')$

2.4.2 Regular Types and Polyregular Types

To base a type language on regular tree expressions requires the identification of a set of function symbols to serve for term formation. For this purpose, the first order specification again manifests a central role. As before, let M be the set of constructor names associated with the first order specification. For each member $m \in M$, introduce a *type constructor* name c. Call the set of type constructors C.

Typographically, a type constructor appears as $\mathsf{a} - \mathsf{constructor}$ to distinguish it from `a-constructor` $\in M$.

The choice for the *arity* of the type constructors is not so straightforward. The arity of a type constructor reflects the degree of polymorphism permitted. For reasons of simplicity, a designer may choose a smaller arity for the type constructor than for the data constructor. An example should clarify the options. Please note that the type language used in the example is, for the moment, informal.

Example 2.4 *(Arity for type constructors)*

Using the first order specification from example 2.2 (repeated here for convenience):

```
constructor   0;
constructor   suc(pred:0 ∪ suc);
constructor   nil;
constructor   cons(hd:0 ∪ suc ∪ cons ∪ nil∪ →, tail:nil ∪ cons)
```

The above definition induces $C = \{\mathsf{0}, \mathsf{suc}, \mathsf{nil}, \mathsf{cons}\}$. As for arity, $\mathsf{0}$ and nil must have 0-arity, as the data constructors are 0-ary.

Consider first the effect of a unary suc constructor. Then one may define such intuitive types as "whole numbers ≥ 2" or "the even whole numbers" via

$\mathsf{suc}(\mathsf{suc}(\mathsf{suc}(B) + \mathsf{0})$ where $B = \mathsf{suc}(B) + \mathsf{0}$	whole numbers ≥ 2
$E = \mathsf{suc}(\mathsf{suc}(E)) + \mathsf{nil}$	The even whole numbers

The price for this flexibility, however, is additional complexity for simpler functions. For example, the suc constructor has the type $\mathsf{0} + \mathsf{suc}(I) \rightarrow \mathsf{suc}(I)$, where $I = \mathsf{nil} + \mathsf{suc}(I)$.

In contrast, one might choose to have the type constructor suc be 0-ary, even though the data constructor suc is unary. In this case, the sophisticated types such as "even numbers" are not definable, but the constructor suc has type $\mathsf{0}+\mathsf{suc} \rightarrow \mathsf{suc}$. A language specifier may decide that the extra complexity of type expressions does not warrant the additional flexibility.

Now consider the cons, nil pair of types. The data constructor cons is 2-ary, so the type constructor can be designed as 0-ary, 1-ary, or 2-ary. The first argument to cons can be any construction (including $\rightarrow$). This complete lack of restriction suggests that cons should be at least 1-ary. If a designer chooses 1-ary for cons then the constructor cons's type can be written without recursion:

$$\mathtt{cons} : \alpha \rightarrow \mathsf{nil} + \mathsf{cons}(\alpha) \rightarrow \mathsf{cons}(\alpha)$$

As suggested by example 2.4, the choice of arities for the type constructors requires some thought by the designer.

Once the type arity decision has been made, however, the type language for the first order specification consists of $\Sigma(C, X)$, the regular tree expressions over C, X where X is a set of *type variables.* As an expression alternative, we often indicate A^{*x} as `fix` $x.A$, or even "equationally" as $x = A(x)$. For the sake of completeness, we repeat definition 2.11 here, using the `fix` notation. We coin the phrase *regular types* for this type language.

Repeating for convenience, the syntax of regular types is defined as:

Definition 2.16 *(Regular Types – Regular Tree Expressions Revisited)* Let C be a set of type constructors, and let X be a set of type variables. Let c_0 range over all 0-ary type constructors, and x stand for any variable of X. Then the following grammar generates the regular types

$$\begin{array}{ll} \text{RegType} & ::= \\ & c_0 \\ & x \end{array}$$

$$c_n(\text{RegType}_1, \ldots \text{RegType}_n)$$
$$\text{RegType}_1 + \text{RegType}_2$$
$$\texttt{fix}\ x.\text{RegType}$$

.

For semantic reasons, the `fix` operation must be slightly restricted. The restriction is mild. The sorts of expressions that are excluded are of the form:

$$\texttt{fix}\ x.x$$

or even

$$\texttt{fix}\ x.x + c(x) + k$$

The syntactic restriction for `fix` expressions is termed *formal contractiveness*[8]. The definition is as follows:

Definition 2.17 *(Formal Contractiveness)* A regular tree expression E is formally contractive in x iff:

1. $E = c_0$, for some 0-ary constructor
2. $E = x'$ for some variable $x' \neq x$
3. $E = c_n(E_1, \ldots, E_n)$, with c_n an n-ary constructor
4. $E = E_1 + E_2$, where E_1 and E_2 are both formally contractive in x
5. $E = \texttt{fix}\ x'.E_1$, and E_1 is contractive in x and x'.

A regular type is considered *ill-formed* if it has the form $\texttt{fix}\ x.E$, and E is not formally contractive in x. From now on, "regular type" means "well-formed regular type".

Standard type literature distinguishes the type expressions with no type variables as the *monotypes*. The analogous concept for regular types is slightly different, due to the presence of the `fix` operator.

Definition 2.18 *(Free type variables)* Let R be a regular type expression. Let $V(R)$ be the set of all type variables occurring in R. Then the set of free variables of R, notated as $F(R)$ is

$$F(R) = \begin{cases} V(R) - \{x\} & R = \texttt{fix}\ x.R' \\ V(R) & \text{otherwise} \end{cases}$$

[8]This corresponds to a *reduced grammar* in the tree grammar treatment

Then R is a *regular monotype* iff $F(R) = \emptyset$

The standard literature also distinguishes quantified types as type schemes. This same quantification distinguishes regular types from regular type schemes.[9] Regular type schemes mimic the standard situation exactly: If R is a regular type, possibly containing free variables, then $\forall\ x.R$ is a regular type scheme.

$$\text{RegTypeScheme} ::= \text{Regtype}|\forall\text{x.RegTypeScheme}$$

Regular type schemes are considered equivalent up to renaming of bound (by the quantifier) variables. Note that both a regular type and a regular type scheme may contain free variables.

Regular type schemes are sometimes called Polyregular types.

2.4.3 Regular Types with Functions

We can extend the syntax of regular types to type expressions including the function space constructor by simple inclusion of the function (arrow) constructor[10]. Syntactically speaking, the function constructor $\rightarrow$ behaves like any other 2-ary constructor. From an algebraic/order-theoretic point of view, however, the function constructor is very different from the first order constructors. The difference is that arrow is *antimonotonic* in its first argument. Example 2.5 illustrates the antimonotonicity of functions.

> **Example 2.5** *(Antimonotonicity of function types)* Let f be a function of type $\mathsf{0} + \mathsf{suc} \rightarrow \mathsf{0} + \mathsf{suc}$. Now if $f' : \mathsf{0} + \mathsf{suc} \rightarrow \mathsf{suc}$ is certainly such a function – the input types are identical, but the output type must be a suc object. But any suc object is trivially a $\mathsf{0} + \mathsf{suc}$ object. Therefore, the $\rightarrow$ type constructor is monotonic in its *second* argument.
>
> On the other hand, $f : \mathsf{0} + \mathsf{suc} \rightarrow \mathsf{0} + \mathsf{suc}$ works on input values that are either $\mathsf{0}$ or suc. So, in particular, $f : \mathsf{suc} \rightarrow \mathsf{0} + \mathsf{suc}$. This suggests that $(\mathsf{0}+\mathsf{suc} \rightarrow \mathsf{0}+\mathsf{suc}) \subseteq (\mathsf{suc} \rightarrow \mathsf{0}+\mathsf{suc})$, even though $\mathsf{suc} \subseteq (\mathsf{0}+\mathsf{suc})$ Stated

[9]The distinction is blurred somewhat in implementation. The actual quantifiers are not explicit in the output of type information by the language processor. Free variables are implicitly quantified. Nevertheless, at the theoretical level, it is easier to make the quantification explicit.

[10]For purposes of formal contractiveness, $\rightarrow$ is treated as any other 2-ary constructor.

in words: The more restrictive the class of inputs to a function, the more types it belongs to. This property is named *antimonotonicity*

The $\rightarrow$ type constructor is antimonotonic in its first argument, but monotonic in its second argument.

The antimonotonicty of arrow does not fit in the regular tree expression framework established for non-functional types. To reason about functional types requires an inference system:

Definition 2.19 *(Inference Rules for Functional Types)*

$$
\begin{array}{rc}
\textsc{Ord1:} & A \vdash T_1 \subseteq T_2 \quad T_1, T_2 \text{ are first order}, \Sigma(T_1) \subseteq \Sigma(T_2) \\
\textsc{Axiom:} & A, \tau_1 \subseteq \tau_2 \vdash \tau_1 \subseteq \tau_2 \\
\textsc{Refl:} & A \vdash \tau \subseteq \tau \\
\textsc{Union:} & A \vdash \tau_1 \subseteq \tau_1 + \tau_2 \\
\textsc{Trans:} & \dfrac{A \vdash \tau_1 \subseteq \tau_2 \qquad\qquad A \vdash \tau_2 \subseteq \tau_3}{A \vdash \tau_1 \subseteq \tau_3} \\
\textsc{Con:} & \dfrac{A \vdash \tau_1 \subseteq \tau_1' \ldots A \vdash \tau_n \subseteq \tau_n'}{A \vdash c(\tau_1, \ldots, \tau_n) \subseteq c(\tau_1', \ldots, \tau_n')} \\
\textsc{Fun:} & \dfrac{A \vdash \tau_1' \subseteq \tau_1 \qquad\qquad A \vdash \tau_2 \subseteq \tau_2'}{A \vdash \tau_1 \rightarrow \tau_2 \subseteq \tau_1' \rightarrow \tau_2'} \\
\textsc{Fix1:} & A \vdash \texttt{fix}\ x.\tau = \tau[x \leftarrow \texttt{fix}\ x.\tau] \\
\textsc{Fix2:} & \dfrac{A, x \subseteq y \vdash \tau_1 \subseteq \tau_2}{A \vdash \texttt{fix}\ x.\tau_1 \subseteq \texttt{fix}\ y.\tau_2}
\end{array}
$$

The associativity, commutativity, and idempotence of the $+$ operator are assumed in the above inference rules. Amadio and Cardelli[2] derive a similar inference system to address the problems of subtyping and recursive types.

2.5 The Semantics of Regular Types

In order to make meaningful statements about type expressions, and their relation to program expressions, we need a semantics for type expressions. The semantic formalism for types in this exposition is a metric space of *ideals* of D. The ideal structure is closed under union, cross products, direct sums, and function construction. Furthermore, the metric structure assures solutions to certain fixed point equations.[11]

A summary of the important features of the ideal model for types appears in section 2.5.1. Next, the semantics for the regular types of section 2.4.2 are defined in section 2.5.2. Finally, section 2.5.3 contains a proof of soundness for the first order (algebraic) subtype inference scheme, and well as a proof of soundness for the inference scheme in definition 2.19.

2.5.1 A Summary of the Ideal Model for Types

The informal notion underlying the concept of type is that a type is a "set" of values. When the value space has structure, however, the set of values comprising the type should have some structure as well – that is, given two types, one would like for the cartesian product to be a type, the continuous functions to be a type, etc. If a type can be structure free, then there may be no good way to define new types from old.

When using domains (countably based, algebraic, complete, partial orders), one successful semantics for types introduced by MacQueen et al[31] uses the *ideals* of the domain to interpret type expressions.

The definition of an ideal is:

Definition 2.20 *(Ideals of D)* Let D be a domain with order relation $\sqsubseteq$, and let $I \subseteq D$. Then I is an ideal iff:

1. I is *downward closed.* That is, for all $x \in I$, if $y \sqsubseteq x$, then $y \in I$.
2. I is *consistent closed.* That is, if $X \subseteq I$ is a consistent (that is, $\exists\, b \in D$, s.t. $\forall\, x \in X, x \sqsubseteq b$), then $\sqcup X \in I$. That is, . Note that $X = \emptyset$ gives the result that $\perp \in I$

The set of types consists of the ideals not containing **wrong**:

Definition 2.21 *(The Set of Types)*

$$\mathcal{I} = \{ I \,|\, I \text{ is an ideal of } D, \text{ and } \mathbf{wrong} \notin I \}$$

[11]The ideal model is not the only possible choice for denotational type models. See Cartwright[11] for a discussion of other possible choices

Theorem 2.2 *($\mathcal{I}$ is a lattice)* $\mathcal{I}$ is a complete lattice under $\subseteq$ (set inclusion) or $\supseteq$ (set containment). The lattice $(\mathcal{I}, \supseteq)$ is sometimes notated as $\mathcal{I}^o$

The type building operations forming products, direct sums, functions, and unions are continuous functions over $\mathcal{I}$ (or $\mathcal{I}^o$). In particular, the continuous function constructor is the most interesting:

Definition 2.22 *(Function Constructor)* For $A, B \in \mathcal{I}$

$$A \to B = \{f \in D \to D \mid f(A) \subseteq B\}$$

To treat recursion in the context of types-as-ideals, we need to impose some extra structure on $\mathcal{I}$. The extra structure is a metric, turning $\mathcal{I}$ into a complete metric space. Given a complete metric space, the Banach fixed point theorem from classical analysis assures the existence of unique fixed points for "contractive" maps. The type expressions of the type language fortunately, are all contractive, as shown in section 2.5.2.

The MacQueen-Plotkin-Sethi(MPS) metric is defined via a *rank* function. A rank function assigns to each finite element of D, a natural number.

Definition 2.23 *(The MPS rank function)* Recall that D is constructed by a limiting process using a chain of domains D_n, starting from $D_0 = \{\perp\}$. The next domain is given as:

$$D_{n+1} = D_n \to D_n \oplus (D_n)_{c_1} \oplus (D_n)_{c_l} \oplus \mathbf{W}$$

The rank of a basis element b is taken to be the least n such that $b \in D_n$ but $b \notin D_{n-1}$.

Given a rank of elements, define a metric on ideals:

Definition 2.24 *(MPS metric on ideals)* Let $I, J \in \mathcal{I}$. Then $I \neq J$ implies there is an element distinguishing the two ideals. Furthermore, by closure properties of ideals, there is a finite element separating the two ideals. Let b be the basis (finite) element *of least rank* separating the two ideals. Then the *closeness* of I, J,$c(I, J) = \text{rank}(b)$. If I, J are equal, the closeness is defined to be ∞. The distance between I, J is defined to be

$$d(I, J) = 2^{-c(I,J)}$$

Theorem 2.3 $(\mathcal{I}, d)$, with d as above is a complete metric space.

Furthermore, d satisfies a strong version of the triangle inequality:

$$d(I,J) \leq \max(d(I,K), d(K,J))$$

In other words, d is ultrametric.

Contractive mappings are the mappings of interest for fixed point theory.

Definition 2.25 *(Contractive,Non-Expansive Maps)* Let $f : X \to Y$ be a map of metric spaces. Then f is uniformly contractive if there is a real number $0 \leq r < 1$ such that for all $x, x' \in X$

$$d(f(x), f(x')) \leq r d(x, x')$$

If we relax the restriction on r so that $0 \leq r \leq 1$ then the map f is termed *non-expansive*

The Banach fixed point theorem [20] summarizes the importance of contractive maps.

Theorem 2.4 *(Banach fixed point theorem)* Let X be a complete metric space. Let $f : X \to X$ be contractive. Then f has a unique fixed point given by the limit of the Cauchy sequence $f^n(x_0)$ where x_0 is any point in X.

MacQueen, Plotkin and Sethi show;

Theorem 2.5 *(Properties of standard type functions)*

1. The union function $\cup$ is non-expansive in its arguments
2. The product (tuple construction), direct sum, and function type operators are all contractive.
3. The projection functions are all non-expansive
4. The composition of a contractive map with a non-expansive map (and vice-versa) is contractive.
5. Since d is an ultra metric, a multi-argument function is contractive iff it is contractive in all arguments separately.
6. If $f(u, v)$ is contractive, then $\mu\ f$ is contractive.

This subject is discussed in more detail by MacQueen et al [31, 32].

2.5.2 Semantics of Regular Types

The definition of semantics for a type language is quite similar to the definition of semantics for a programming language. For a type language, however, the meanings are ideals of D, rather than elements of D.[12]

The definition for the semantics of regular types appears below:

Definition 2.26 *(Type semantic function $\mathcal{T}[\![\cdot]\!]$)* Let $\nu : V \to \mathcal{I}$ denotes a valuation for free type variables; In_S denotes the injection function for summand S mapped over ideals of S (interpreted as sets); μ stands for the fixed point operator on contractive functions mapping $\mathcal{I}$ into $\mathcal{I}$: $\mathtt{c}$ denotes any type constructor other than $\to$: and $\Longrightarrow: \mathcal{I} \to \mathcal{I} \to \mathcal{I}$ denotes the ideal function space constructor defined by the equation:

$$A \Longrightarrow B = \{f \in D \to D \mid f(A) \subseteq B\}.$$

Let $\mathcal{T}[\![\cdot]\!]$, the semantic function for types be defined recursively by these equations:

$$\begin{array}{rcl}
\mathcal{T}[\![\emptyset]\!]\nu & = & \{\bot\} \\
\mathcal{T}[\![c]\!]\nu & = & \mathcal{I}(D_{\mathbf{c}}) \text{ for 0-ary functions (constants)} \\
\mathcal{T}[\![c(t_1, \ldots, t_n)]\!]\nu & = & \mathcal{I}(D_{\mathbf{c}})(\langle \mathcal{T}[\![t_1]\!]\nu, \ldots, \mathcal{T}[\![t_n]\!]\nu \rangle) \text{ for } c \neq \to \\
\mathcal{T}[\![t_1 \to t_2]\!]\nu & = & \mathcal{I}(D \to D)(\mathcal{T}[\![t_1]\!]\nu \Longrightarrow \mathcal{T}[\![t_2]\!]\nu) \\
\mathcal{T}[\![t_1 + t_2]\!]\nu & = & \mathcal{T}[\![t_1]\!]\nu \cup \mathcal{T}[\![t_2]\!]\nu \\
\mathcal{T}[\![\mathtt{fix}\ \alpha.t]\!]\nu & = & \mu\ i.f(i) \\
 & & \text{where } f(i) = \mathcal{T}[\![t]\!]\nu[\alpha = i]
\end{array}$$

In order to establish that the semantic function in definition 2.26 is indeed well-defined requires that the argument of the μ operator is contractive. Recall that the syntactic argument to the `fix` operator must be *formally* contractive as stated in definition 2.17. Consequently, it suffices to show that formally contractive expressions are semantically interpreted as contractive maps on $\mathcal{I}$. We begin by showing that the types induced by elements of M are contractive.

We introduce the notion of an induced function to facilitate reasoning about the fixed point values.

[12]As a note on terminology, a function from type variables to ideals is often called a *valuation*.

Definition 2.27 *(Induced Function)* As notation, define the induced function F_t as

$$F_{t(\alpha)}\nu(i) \equiv \mathcal{T}[\![t]\!]\nu[\alpha = i]$$

Lemma 2.1 *(Constructor terms are contractive)* Let $c \in M$ be any constructor such that $\text{Arity}(c) > 1$. Then F_c is contractive.

Proof Let I, J be two ideals, distinguished a witness $v \in D$ of rank r. Then $F_c(I) = \mathcal{I}(D_c)(T_1 \times \ldots I \ldots \times T_n)$, and $F_c(J) = \mathcal{I}(D_c)(T_1 \times \ldots J \ldots \times T_n)$ are distinguished by $a_1 \times \ldots v \times \ldots a_n$, where $a_i \in T_i$. The min rank of this element is at least $v+1$. Consequently, $d(F_c(I), F_c(J) \leq \frac{1}{2}d(I, J)$, and so F_f is contractive in any argument. Further, since MPS is an ultrametric, contractiveness in each argument separately implies contractiveness in all arguments. □

Lemma 2.1 and the results of theorem 2.5 key the proof that F_t for any formally contractive t is contractive.

Theorem 2.6 *(Formally Contractive implies Semantically Contractive)* Let t be any expression formally contractive in x. Then F_t is contractive with respect to the MPS metric.

Proof Proceed by case analysis on the the structure of t (according to definition 2.17). Details for this proof appear in section 2.8. □

Theorem 2.6 assures that the semantics for `fix`-types given in definition 2.26 are defined for all formally contractive regular type expressions. By convention, only formally contractive expressions are allowed for `fix` expressions. Hence, the semantics for regular types is well-defined.

2.5.3 Soundness of Subtype Inference

The type language itself admits a reasoning process for determining when one type is a subtype of another. The semantics should preserve the subtype properties derivable by either regular tree (algebraic) decision procedure, or the conventional inference system given in section 2.4.3. Consequently, we will show the soundness of the composite system in two parts: First, we will establish that the decision procedure for first order regular types is sound with respect to the ideal semantics. Then, we establish that the rest of the inference system of definition 2.19 is sound with respect to the ideal semantics.

The first order decision procedure takes two (first order) regular type *expressions*, and algebraically determines whether one is contained in the other. We would like to show the soundness of the first order decision procedure, however, without reference to any specific procedure. This is possible by observing that one can prove $t \in E$ by running the containment decision procedure to determine if $\{t\} \subseteq E$. Note that any *term* $t \in \Sigma(C, X)$ is also a regular type. Thus, for *any* regular type E, $\Sigma(E)$ is a union of singleton sets, each singleton a regular type. Consequently, showing soundness of subtype inference requires showing the equivalence union of singleton term ideals with the ideal for the original regular type. The equivalence is shown in theorem 2.7. The proof of the theorem is deferred to section 2.8.

As noted earlier, $\Sigma(T)$ is a set of special regular types called regular terms. The algebraic reasoning system using regular tree algorithms (or their equivalent) makes statements about containment of various $\Sigma(T)$'s. The ideal based semantics, however, works directly with the T expression. The connection between $\Sigma(T)$ and the ideal semantics, fortunately, is very strong. The connection is formulated in the following soundness theorem.

Theorem 2.7 *(Soundness of First Order Inference)* For any first order regular type expression T, $\mathcal{T}[\![\Sigma(T)]\!]\nu = \mathcal{T}[\![T]\!]\nu$.

Proof We use structural induction. Details are in section 2.8 $\square$

The of theorem 2.7 is to assure that *any* correct decision procedure for deciding the subset relationship of two regular type *expressions* is sound with respect to the ideal semantics for regular types.

The complete inference system of definition 2.19, includes all the first order subset relations, as well as some others. The soundness of the extended inference system is the subject of theorem 2.8.

Theorem 2.8 *(Extended Subtype Soundness Theorem)* The inference rules in definition 2.19 are sound.

Proof By structural induction on the shape of the proof. Details in section 2.8 □

2.6 Type Inference for Exp Using Regular Types

The previous sections of this chapter introduced both a programming language and a type language, and defined their respective semantics. The usefulness of a type discipline, however, hinges on its ability to prove statements of the form "expression e has type t". The intuitive notion "has type" is formalized by a set of inference rules.

The components of a type inference statement consist of a type *assumption* A, an expression of Exp e, and a type expression t. A type assumption is an assignment of all free variables to type expressions. Type statements take the form

$$A \vdash e : t$$

The intuitive meaning of a type statement is "under assumption A, the expression e has type t". The type inference system provides a system for constructing type statements. One crucial property of such a system is soundness. The type statments that are deduced within the inference system framework should certainly be valid, otherwise the system has no practical value.

The inference system for the regular types is based on the inference system for ML as defined by Milner. These original rules serve to illustrate the treatment of parametric polymorphism. To understand parametric polymorphism in this context requires an understanding of *instantiation.*

The parametric polymorphism in the ML system, and the regular type system both depend on the distinction between types and type schemes. A type scheme, recall, is quantified at the outermost level. Thus

Definition 2.28 *(Generic Instances)* Let σ be a type scheme:

$$\sigma = \forall \alpha_1, \ldots, \alpha_n . \tau$$

Let τ' be a type. Then τ' is a generic instance of σ, written $\sigma > \tau'$ iff there exists substitution S over $\{\alpha_1, \ldots, \alpha_n\}$ such that

$$S(\tau) = \tau'$$

The definition is extended to type schemes by asserting that $\sigma_1 \geq \sigma_2$ if for all types τ, $\sigma_2 > \tau$ implies $\sigma_1 > \tau$

Intuitively, a function described by a type scheme has all of the types that are generic instances of the type scheme. For example, the `length` function can be said to have type $\forall t.\mathsf{list}(t) \to \mathsf{int}$. That is, `length` has any generic instance of its type scheme, such as: $\mathsf{list}(\mathsf{int}) \to \mathsf{int}$, or $\mathsf{list}(\mathsf{int}) \to \mathsf{int}$, or even $\forall t.\mathsf{list}(\mathsf{list}(t)) \to \mathsf{int}$.

The Milner rules appear below in definition 2.29. Conventionally, any σ expressions ranges over type schemes or types, but any τ term must be a type (unquantified type expression).

Definition 2.29 *(The Milner rules)*

$$
\begin{array}{lcl}
\textsc{Taut}: & A \vdash x : \sigma & A(x) = \sigma \\
\textsc{Inst}: & \dfrac{A \vdash x : \sigma}{A \vdash x : \sigma'} & \sigma' < \sigma \\
\textsc{Gen}: & \dfrac{A \vdash e : \sigma}{A \vdash e : \forall \alpha \sigma} & \alpha \text{ not free in } A \\
\textsc{Abst}: & \dfrac{A \cup \{x : \tau'\} \vdash e_1 : \tau}{A \vdash \lambda x.e_1 : \tau' \to \tau)} & \\
\textsc{App}: & \dfrac{A \vdash f : \tau_1 \to \tau_2 \quad A \vdash e : \tau_1}{A \vdash (f\ e) : \tau_2} & \\
\textsc{Let}: & \dfrac{A \vdash e_1 : \sigma \quad A \cup \{x : \sigma\} \vdash e_2 : \tau}{A \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau} &
\end{array}
$$

The Milner type system presumes that all monotypes are disjoint. In contrast, the regular types, and polyregular types have more structure. The type inference rules for polyregular types must accommodate this extra structure. The regular types admit a relation $\subseteq$ giving an order structure to the types. Consequently, the inference systems for polyregular type inference contains one extra rule:

$$
\textsc{Sub}: \quad \frac{A \vdash e : \tau}{A \vdash e : \tau'} \quad \tau \subseteq \tau'
$$

For convenience, we restate the Milner rules, plus the new rule in definition 2.30 Please note that both $\leq$ (for generic instance) and $\subseteq$ (the ordering relation) appear in the following definition. The convention of σ's for either type schemes or types, and τ for types only is still in force for the definition.

Definition 2.30 *(The modified rules)*

TAUT:	$A \vdash x : \sigma$	$A(x) = \sigma$
INST:	$\dfrac{A \vdash x : \sigma}{A \vdash x : \sigma'}$	$\sigma' < \sigma$
GEN:	$\dfrac{A \vdash e : \sigma}{A \vdash e : \forall\alpha\sigma}$	α not free in A
ABST:	$\dfrac{A \cup \{x : \tau'\} \vdash e_1 : \tau}{A \vdash \lambda x.e_1 : \tau' \rightarrow \tau)}$	
APP:	$\dfrac{A \vdash f : \tau_1 \rightarrow \tau_2 \quad A \vdash e : \tau_1}{A \vdash (f\ e) : \tau_2}$	
LET:	$\dfrac{A \vdash e_1 : \sigma \quad A \cup \{x : \sigma\} \vdash e_2 : \tau}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$	
SUB:	$\dfrac{A \vdash e : \tau}{A \vdash e : \tau'}$	$\tau \subseteq \tau'$

The rules in definition 2.30 indicate when a *syntactic* expression e has a *syntactic* type τ. Semantically, a value $v \in D$ has type $T \in \mathcal{I}$, written as $v : T$ whenever $v \in T$. Analogously, an expression e in environment ρ has type τ in valuation ν iff $\mathcal{E}[\![e]\!]\rho \in \mathcal{T}[\![\tau]\!]\nu$. Extending this idea to polyregular types employs the interpretation of polyregular types as "standing for" all their generic instances. Definition 2.31 captures this notion:

Definition 2.31 *(Polyregular type semantics)* Let $\sigma = \forall\alpha_1 \ldots \alpha_n.t$ be a polyregular type. Let $\tau_1, \ldots, \tau_n$ be any regular monotypes. Let I_i be the ideal $\mathcal{T}[\![\tau_i]\!]\nu$. Then a value $v : \mathcal{T}[\![\sigma]\!]$ in valuation ν if for *every* choice of monotypes τ_i:

$$v \in \mathcal{T}[\![t]\!]\nu[\alpha_1 \leftarrow I_1, \alpha_n \leftarrow I_n]$$

Damas[18] proves a useful lemma. This lemma presumes that type schemes are equivalent up to renaming of quantified variables.

Lemma 2.2 *(Structure of instance relation)* Let $\sigma_1 = \forall\alpha.\tau_1, \sigma_2 = \forall\alpha.\tau_2$ If $\sigma_1 \geq \sigma_2$, then

$$\mathcal{T}[\![\tau_1]\!]\nu = \mathcal{T}[\![\tau_2]\!]\nu$$

for any ν.

Finally, soundness requires that the type assumption A and the environment ρ be semantically "matched" That is,

Definition 2.32 *(Environment matches assumption)* An environment ρ *respects* a type assumption A iff for all $x \in \mathrm{Dom}(A)$, $A(x) : \sigma$, $\rho(x) : \mathcal{T}[\![\sigma]\!]\nu$ for all valuations ν.

Given an environment respecting the type assumptions, soundness is the property that syntactic type inference is modeled by semantic type membership.

Theorem 2.9 *(Soundness of Regular Type Inference)*

Let $A \vdash e : \sigma$ be any inference from definition 2.30. Let ρ be an environment respecting A, and let ν be any valuation. Then $\mathcal{E}[\![e]\!]\rho : \mathcal{T}[\![\sigma]\!]\nu$

Proof By structural induction on the last step of the inference. Details in section 2.8. □

As perspective, all the cases in theorem 2.9 are virtually identical to the original proof in [34, 18]. The only difference is the rules of definition 2.30 have one additional element, namely rule Sub. The semantics of rule Sub are very intuitive — any type can be replaced by a supertype. The effect of such a simple change is surprising. In section 2.7 some example deductions serve to illustrate these effects.

2.7 Some Example Deductions

The usefulness and flexibility of the polyregular type system can be best seen with some examples. In this section, we will examine the two examples in the introduction, as well as two other more intricate examples.

The first examples are taken from chapter 1 Throughout these examples, the `if` construct is syntactic sugar for:

$$\texttt{if} \equiv \texttt{case}_{\mathbf{true},\mathbf{false}}$$

By assumption, $\texttt{if} : \forall\ \alpha.\mathsf{true} + \mathsf{false} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$.

The examples use the following first order specification:

$$\begin{array}{lc}
\texttt{constructor} & \texttt{0;} \\
\texttt{constructor} & \texttt{suc}(\texttt{pred} : (\texttt{0} + \texttt{suc})); \\
\texttt{constructor} & \texttt{nil;} \\
\texttt{constructor} & \texttt{cons}(\texttt{hd} : (\texttt{0} + \texttt{suc}), \texttt{tl} : (\texttt{nil} + \texttt{cons})) \\
\texttt{constructor} & \texttt{true} \\
\texttt{constructor} & \texttt{false}
\end{array}$$

The types for the constructor functions are

$$\begin{array}{rcl}
\mathbf{suc} & : & \mathsf{0} + \mathsf{suc} \to \mathsf{suc} \\
\mathbf{cons} & : & \forall\ \alpha.\alpha \to (\mathsf{nil} + \mathsf{cons}(\alpha)) \to \mathsf{cons}(\alpha)
\end{array}$$

The first example we consider is one of the motivating examples from chapter 1. Recall that example 1.3 could not be assigned a type using standard methods, because there were no union types. The following example indicates that the union type facility of the regular types are adequate to assign a type in this case.

Example 2.6 Consider the function

$\lambda x.$ `if` x `then` 1 `else` `nil`

The symbol 1 abbreviates the term $\mathbf{suc(0)})$. Abbreviate the initial type assumption as T_0. Abbreviate $T_0 \cup \{x : \mathsf{true} + \mathsf{false}\}$ as T_1.

Consider the following polyregular deduction:

$$\begin{array}{rll}
1 & T_1 \vdash \texttt{if} : \forall\alpha.\mathsf{true} + \mathsf{false} \to \alpha \to \alpha & \text{Ax} \\
2 & T_1 \vdash \texttt{if} : \mathsf{true} + \mathsf{false} \to (\mathsf{nil} + \mathsf{suc}) \to (\mathsf{nil} + \mathsf{suc}) & 1, \text{Inst} \\
3 & T_1 \vdash x : \mathsf{true} + \mathsf{false} & \text{Ax} \\
4 & T_1 \vdash \texttt{if}(x) : (\mathsf{nil} + \mathsf{suc}) \to (\mathsf{nil} + \mathsf{suc}) \to (\mathsf{nil} + \mathsf{suc}) & 2, 3, \text{App} \\
5 & T_1 \vdash 1 : \mathsf{suc} & \text{Ax} \\
6 & T_1 \vdash 1 : \mathsf{suc} + \mathsf{nil} & 5, \text{Sub}, \text{alg} \\
7 & T_1 \vdash \texttt{if}(x)(1) : (\mathsf{nil} + \mathsf{suc}) \to (\mathsf{nil} + \mathsf{suc}) & 6, 4, \text{App} \\
8 & T_1 \vdash \texttt{nil} : \mathsf{nil} & \text{Ax} \\
9 & T_1 \vdash \texttt{nil} : (\mathsf{nil} + \mathsf{suc}) & 8, \text{Sub}, \text{alg} \\
10 & T_1 \vdash \texttt{if}(x)(1)(\texttt{nil}) : (\mathsf{nil} + \mathsf{suc}) & 7, 9, \text{App} \\
11 & T_0 \vdash \lambda x.\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else nil} : & \\
 & \quad \mathsf{true} + \mathsf{false} \to (\mathsf{nil} + \mathsf{suc}) \to (\mathsf{nil} + \mathsf{suc}) \to (\mathsf{nil} + \mathsf{suc}) & 10, \text{Abs}
\end{array}$$

The next example indicates that regular types also remove the difficulty of example 1.4 from chapter 1.

Example 2.7 Consider the function

$(\lambda x.\texttt{cons}\,(1, x))\,[\texttt{true}, \texttt{false}]$

The expression $[\texttt{true}, \texttt{false}]$ stands for

$$\texttt{cons}(\texttt{true}, \texttt{cons}(\texttt{false}, \texttt{nil}))$$

And so, has type $\mathsf{cons}(\mathsf{true} + \mathsf{false})$.

As in example 2.6, let T_0 be the initial type environment, and let T_1 be $T_0 \cup \{x : \mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false})\}$. With these conventions in mind, consider the following deduction:

1	$T_1 \vdash \texttt{cons} : \forall\alpha.\alpha \to (\mathsf{nil} + \mathsf{cons}(\alpha)) \to \mathsf{cons}(\alpha)$	AX
2	$T_1 \vdash \texttt{cons} :$ $\mathsf{suc} + \mathsf{true} + \mathsf{false} \to$ $\mathsf{nil} + \mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false}) \to$ $\mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false})$	INST
3	$T_1 \vdash 1 : \mathsf{suc}$	AX
4	$T_1 \vdash 1 : \mathsf{suc} + \mathsf{true} + \mathsf{false}$	SUB, algebra
5	$T_1 \vdash \texttt{cons}(1) :$ $\mathsf{nil} + \mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false}) \to$ $\mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false})$	$2, 4$, APP
6	$T_1 \vdash \texttt{cons}(1) :$ $\mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false}) \to$ $\mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false})$	5, SUB, inference
7	$T_1 \vdash x : \mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false})$	AX
8	$T_1 \vdash \texttt{cons}(1)(x) : \mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false})$	$6, 7$, APP
9	$T_0 \vdash \lambda x.\texttt{cons}(1)(x) :$ $\mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false}) \to$ $\mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false})$	8, ABS
10	$T_0 \vdash [\texttt{true}, \texttt{false}] : \mathsf{cons}(\mathsf{true} + \mathsf{false})$	AX
11	$T_0 \vdash [\texttt{true}, \texttt{false}] : \mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false})$	10, SUB, algebra
12	$T_0 \vdash (\lambda x.\texttt{cons}(1)(x))[\texttt{true}, \texttt{false}] :$ $\mathsf{cons}(\mathsf{suc} + \mathsf{true} + \mathsf{false})$	$9, 11$, APP

Example 2.8 *(The function* **deep** *)* The following example is taken from an introductory Scheme course taught at Rice. The function **deep** is supposed to take a non-negative integer n as input, and return a list that has a 0 nested $n+1$ deep. Using Exp, with the first order specification as above, one possible solution is:

$$
\begin{array}{l}
\textbf{deep} = \\
\lambda n.\mathtt{case}\ n\ \mathtt{of} \\
\quad\quad \mathtt{0\ :\ \ cons(0,nil)} \\
\quad\quad \mathtt{suc\ :\ \ cons(deep(pred(n)),nil)}
\end{array}
$$

Note the use of syntactic sugaring for `case`. In fact, the recursion is syntactic sugaring also. To do the type deduction, we will use the desugared version:

$$
\begin{array}{l}
\textbf{deep} = \\
\mathtt{rec}\ (\lambda D.\lambda n. \\
\quad\quad \mathtt{case}_{\mathsf{0,suc}} \\
\quad\quad\quad\quad (\lambda d.\mathtt{cons(0)(nil)}) \\
\quad\quad\quad\quad (\lambda d.\mathtt{cons}(D(\mathtt{pred}(d)))(\mathtt{nil})) \\
\quad\quad\quad\quad (n)
\end{array}
$$

where `rec` is the least fixed point function.

For this example, the initial type assumption T_0 includes:

$$
\begin{array}{rcl}
\mathtt{case}_{\mathsf{0,suc}} & : & \forall\alpha.(\mathsf{0} \rightarrow \alpha) \rightarrow (\mathsf{suc} \rightarrow \alpha) \rightarrow (\mathsf{0} + \mathsf{suc} \rightarrow \alpha) \\
\mathtt{rec} & : & \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha
\end{array}
$$

as well as the standard types for the constructors and selectors.

Again, some abbreviation simplifies presentation. Abbreviate the type $\mathtt{fix}\ t.\mathsf{0} + \mathsf{cons}(t))$ by Z. Use the following names for various type environments:

$$
\begin{array}{rcl}
T_1 & = & T_0 \cup \{D : \mathsf{0} + \mathsf{suc} \rightarrow \mathsf{cons}(\mathsf{Z})\} \\
T_2 & = & T_1 \cup \{n : \mathsf{0} + \mathsf{suc}\} \\
T_3 & = & T_2 \cup \{d : \mathsf{0}\} \\
T_3' & = & T_2 \cup \{d : \mathsf{suc}\}
\end{array}
$$

Beginning with the inner function:

1	$T_3 \vdash \mathtt{cons} : \forall\ \alpha.\alpha \to (\mathsf{nil} + \mathsf{cons}(\alpha)) \to \mathsf{cons}(\alpha)$	Ax
2	$T_3 \vdash \mathtt{cons} : \mathsf{0} \to (\mathsf{nil} + \mathsf{cons}(\mathsf{0})) \to \mathsf{cons}(\mathsf{0})$	Inst
3	$T_3 \vdash \mathtt{0} : \mathsf{0}$	Ax
4	$T_3 \vdash \mathtt{cons}(\mathtt{0}) : (\mathsf{nil} + \mathsf{cons}(\mathsf{0})) \to \mathsf{cons}(\mathsf{0})$	2, 3, App
5	$T_3 \vdash \mathtt{nil} : \mathsf{nil}$	Ax
6	$T_3 \vdash \mathtt{nil} : \mathsf{nil} + \mathsf{cons}(\mathsf{0})$	Sub, alg.
7	$T_3 \vdash \mathtt{cons}(\mathtt{0})(\mathtt{nil}) : \mathsf{cons}(\mathsf{0})$	4, 6, App
8	$T_2 \vdash \lambda d.\mathtt{cons}(\mathtt{0})(\mathtt{nil}) : \mathsf{0} \to \mathsf{cons}(\mathsf{0})$	Abs

Continuing with the next inner function

9	$T_3' \vdash \mathtt{pred} : \mathsf{suc} \to \mathsf{0} + \mathsf{suc}$	Ax
10	$T_3' \vdash d : \mathsf{suc}$	Ax
11	$T_3' \vdash \mathtt{pred}(d) : \mathsf{0} + \mathsf{suc}$	9, 10, App
12	$T_3' \vdash D : \mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z})$	Ax
13	$T_3' \vdash D(\mathtt{pred}(d)) : \mathsf{cons}(\mathsf{Z})$	11, 12, App
14	$T_3' \vdash \mathtt{cons} : \forall\ \alpha.\alpha \to (\mathsf{nil} + \mathsf{cons}(\alpha)) \to \mathsf{cons}(\alpha)$	Ax
15	$T_3' \vdash \mathtt{cons} : \mathsf{Z} \to \mathsf{nil} + \mathsf{cons}(\mathsf{Z}) \to \mathsf{cons}(\mathsf{Z})$	14, Inst
16	$T_3' \vdash \mathtt{cons}(D(\mathtt{pred}(d))) : (\mathsf{nil} + \mathsf{cons}(\mathsf{Z})) \to$ $\mathsf{cons}(\mathsf{Z})$	13, 15, App
17	$T_3' \vdash \mathtt{nil} : \mathsf{nil}$	Ax
18	$T_3' \vdash \mathtt{nil} : \mathsf{nil} + \mathsf{cons}(\mathsf{Z})$	17, Sub, Alg.
19	$T_3' \vdash \mathtt{cons}(D(\mathtt{pred}(d)))(\mathtt{nil}) : \mathsf{cons}(\mathsf{Z})$	16, 18, App
20	$T_2 \vdash \lambda d.\mathtt{cons}(D(\mathtt{pred}(d)))(\mathtt{nil}) : \mathsf{suc} \to \mathsf{cons}(\mathsf{Z})$	19, Abs

To continue the derivation, we abbreviate

$$\lambda d.\mathtt{cons}(\mathtt{0})(\mathtt{nil})$$

by $\mathbf{C_1}$. Additionally, we abbreviate

$$\lambda d.\mathtt{cons}(D(\mathtt{pred}(d)))(\mathtt{nil})$$

by $\mathbf{C_2}$. Thus, the relevant lines in the proof can be rewritten as:

8	$T_2 \vdash \mathbf{C_1} : \mathsf{0} \to \mathsf{cons}(\mathsf{z})$	Abs
20	$T_2 \vdash \mathbf{C_2} : \mathsf{suc} \to \mathsf{cons}(\mathsf{Z})$	19, Abs

Using these notations, we consider the `case` subexpression:

$$
\begin{array}{lll}
21 & T_2 \vdash \mathtt{case}_{\mathsf{0},\mathsf{suc}} : \forall\alpha.(\mathsf{0} \to \alpha) \to (\mathsf{suc} \to \alpha) \to & \\
 & \quad (\mathsf{0} + \mathsf{suc} \to \alpha) & \textsc{Ax} \\
22 & T_2 \vdash \mathtt{case}_{\mathsf{0},\mathsf{suc}} : (\mathsf{0} \to \mathsf{cons}(\mathsf{Z})) \to & \\
 & \quad (\mathsf{suc} \to \mathsf{cons}(\mathsf{Z})) \to & \\
 & \quad (\mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z})) & 21, \textsc{Inst} \\
23 & T_2 \vdash \mathbf{C_1} : \mathsf{0} \to \mathsf{cons}(\mathsf{Z}) & 8, \textsc{Sub}, \mathrm{Alg} \\
24 & T_2 \vdash \mathtt{case}_{\mathsf{0},\mathsf{suc}}(\mathbf{C_1}) : (\mathsf{suc} \to \mathsf{cons}(\mathsf{Z})) \to & \\
 & \quad (\mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z})) & 22, 23, \textsc{App} \\
25 & T_2 \vdash \mathtt{case}_{\mathsf{0},\mathsf{suc}}(\mathbf{C_1})(\mathbf{C_2}) : (\mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z})) & 20, 24, \textsc{App} \\
26 & T_2 \vdash n : \mathsf{0} + \mathsf{suc} & \textsc{Ax} \\
27 & T_2 \vdash \mathtt{case}_{\mathsf{0},\mathsf{suc}}(\mathbf{C_1})(\mathbf{C_2})(n) : \mathsf{cons}(\mathsf{Z}) & 25, 26, \textsc{App}
\end{array}
$$

Again, in the interest of simpler notation, abbreviate

$$\mathtt{case}_{\mathsf{0},\mathsf{suc}}(\mathbf{C_1})(\mathbf{C_2})(n)$$

as **S**. Then we write

$$27 \quad T_2 \vdash \mathbf{S} : \mathsf{cons}(\mathsf{Z}) \quad 25, 26, \textsc{App}$$

So we conclude

$$
\begin{array}{lll}
28 & T_1 \vdash \lambda n.\mathbf{S} : \mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z}) & 27, \textsc{Abs} \\
29 & T_0 \vdash \lambda D.\lambda n.\mathbf{S} : (\mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z})) \to & \\
 & \quad \mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z}) & 28, \textsc{Abs} \\
30 & T_0 \vdash \mathtt{rec} : \forall\alpha.(\alpha \to \alpha) \to \alpha & \textsc{Ax} \\
31 & T_0 \vdash \mathtt{rec} : ((\mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z})) \to & \\
 & \quad (\mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z}))) \to & \\
 & \quad (\mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z})) & 30, \textsc{Inst} \\
32 & T_0 \vdash \mathtt{rec}(\lambda D.\lambda n.\mathbf{S}) : \mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathsf{Z}) & 29, 31, \textsc{App}
\end{array}
$$

Line 32 above states

$$T_0 \vdash \mathbf{deep} : \mathsf{0} + \mathsf{suc} \to \mathsf{cons}(\mathtt{fix}\ t.\mathsf{z} + \mathsf{cons}(t))$$

rewriting the abbreviation for Z.

Example 2.9 *(A tautology checking function)* This example was presented to us by Jim Hook and Neal Nelson[38][13]. The purpose is very simple. Given a curried function f that takes some arbitrary number of boolean arguments, decide if f is a tautology or not. That is, is $f(x_1, x_2, \ldots, x_n)$ always **true**.

Recall that the first order specification includes the 0-ary constructors `true,false`. It is convenient to abbreviate `true` and `false` as `T` and `F` respectively. Furthermore, we will use `fun` to indicate the $\rightarrow$ constructor on the `case` statement.

With that in mind, consider the following (syntactically sugared) function **taut**.

$$
\begin{array}{l}
\textbf{taut} = \\
\lambda f.\texttt{case}\ f\ \texttt{of} \\
\quad \texttt{T : T} \\
\quad \texttt{F : F} \\
\quad \texttt{fun} : \texttt{and}(\textbf{taut}(f(\texttt{T})))(\textbf{taut}(f(\texttt{F})))
\end{array}
$$

The type reasoning is more transparent using a desugared version

$$
\begin{array}{l}
\textbf{taut} = \\
\texttt{rec}(\lambda U.\lambda f. \\
\quad \texttt{case}_{\texttt{T,F,fun}} \\
\qquad \lambda d.\texttt{T} \\
\qquad \lambda d.\texttt{F} \\
\qquad \lambda d.\texttt{and}(U(d(\texttt{T})))(U(d(\texttt{F}))) \\
\qquad (f) \\
\quad)
\end{array}
$$

Again, the initial type assumption T_0 contains all the constructor and selector information, as well as the constant `and`

$$
\begin{array}{rcl}
\texttt{case}_{\texttt{T,F,fun}} & : & \forall \alpha\beta\gamma.(\mathsf{T} \rightarrow \alpha) \rightarrow \\
& & (\mathsf{F} \rightarrow \alpha) \rightarrow ((\beta \rightarrow \gamma) \rightarrow \alpha) \rightarrow \\
& & (\mathsf{T} + \mathsf{F} + (\beta \rightarrow \gamma)) \rightarrow \alpha \\
\texttt{and} & : & (\mathsf{T} + \mathsf{F}) \rightarrow (\mathsf{T} + \mathsf{F}) \rightarrow (\mathsf{T} + \mathsf{F})
\end{array}
$$

[13]They claim it came from an old SASL manual

Some of the type expressions become unwieldy without abbreviation. To prevent this obscurity, the following abbreviations are active for the remainder of this example

$$\begin{aligned} \mathsf{B} &\equiv \mathsf{T} + \mathsf{F} \\ \mathsf{TA} &\equiv \mathtt{fix}\, t.\mathsf{T} + \mathsf{F} + (\mathsf{T} + \mathsf{F} \to t) \end{aligned}$$

Next, we introduce some notation for some type assumptions used in intermediate steps:

$$\begin{aligned} T_1 &= T_0 \cup \{U : \mathsf{TA} \to \mathsf{B}\} \\ T_2 &= T_1 \cup \{f : \mathsf{B} + (\mathsf{B} \to \mathsf{TA})\} \\ T_3 &= T_2 \cup \{d : \mathsf{T}\} \\ T_3' &= T_2 \cup \{d : \mathsf{F}\} \\ T_3'' &= T_2 \cup \{d : (\mathsf{B}) \to \mathsf{TA}\} \end{aligned}$$

The innermost functions (the arguments to `case`) receive treatment first:

$$\begin{array}{lll} 1 & T_3 \vdash \mathtt{T} : \mathsf{T} & \textsc{Ax} \\ 2 & T_3 \vdash \mathtt{T} : \mathsf{B} & 1, \textsc{Sub}, \text{alg.} \\ 3 & T_2 \vdash \lambda d.\mathtt{T} : \mathsf{T} \to \mathsf{B} & 2, \textsc{Abs} \end{array}$$

Similarly,

$$\begin{array}{lll} 4 & T_3' \vdash \mathtt{F} : \mathsf{F} & \textsc{Ax} \\ 5 & T_3' \vdash \mathtt{F} : \mathsf{B} & 5, \textsc{Sub}, \text{alg.} \\ 6 & T_2 \vdash \lambda d.\mathtt{T} : \mathsf{T} \to \mathsf{B} & 5, \textsc{Abs} \end{array}$$

And finally,

7	$T_3'' \vdash \mathtt{T} : \mathsf{T}$	AX
8	$T_3'' \vdash \mathtt{T} : \mathsf{B}$	7, SUB, Alg
9	$T_3'' \vdash d : (\mathsf{B}) \to \mathsf{TA}$	AX
10	$T_3'' \vdash d(\mathtt{T}) : \mathsf{TA}$	8, 9, APP
11	$T_3'' \vdash U : \mathsf{TA} \to (\mathsf{B})$	AX
12	$T_3'' \vdash U(d(\mathtt{T})) : (\mathsf{B})$	10, 11, APP
13	$T_3'' \vdash \mathtt{and} : (\mathsf{B}) \to (\mathsf{B}) \to (\mathsf{B})$	AX
14	$T_3'' \vdash \mathtt{and}(U(d(\mathtt{T}))) : (\mathsf{B}) \to (\mathsf{B})$	12, 13, APP
15	$T_3'' \vdash \mathtt{F} : \mathsf{F}$	AX
16	$T_3'' \vdash \mathtt{F} : \mathsf{B}$	15, SUB, Alg
18	$T_3'' \vdash d(\mathtt{F}) : \mathsf{TA}$	9, 16, APP
19	$T_3'' \vdash U(d(\mathtt{F})) : \mathsf{B}$	11, 18, APP
20	$T_3'' \vdash \mathtt{and}(U(d(\mathtt{T})))(U(d(\mathtt{F}))) : \mathsf{B}$	14, 19, APP
21	$T_2 \vdash \lambda d.\mathtt{and}(U(d(\mathtt{T})))(U(d(\mathtt{F}))) : (\mathsf{B} \to \mathsf{TA}) \to \mathsf{B}$	20, ABS

As in example 2.8, we give the above functions names so that the expressions are simpler. For the next piece of the derivation,

$$\begin{aligned} \mathbf{C_1} &\equiv \lambda d.\mathtt{T} \\ \mathbf{C_2} &\equiv \lambda d.\mathtt{T} \\ \mathbf{C_3} &\equiv \lambda d.\mathtt{and}(U(d(\mathtt{T})))(U(d(\mathtt{F}))) \end{aligned}$$

Considering the $\texttt{case}$ statement:

$$
\begin{array}{lll}
22 & T_2 \vdash \texttt{case}_{\texttt{T},\texttt{F},\texttt{fun}} : \forall\alpha\beta\gamma.(\mathsf{T} \rightarrow \alpha) \rightarrow (\mathsf{F} \rightarrow \alpha) \rightarrow & \\
 & \quad ((\beta \rightarrow \gamma) \rightarrow \alpha) \rightarrow & \\
 & \quad (\mathsf{B} + (\beta \rightarrow \gamma)) \rightarrow \alpha & \textsc{Ax} \\
23 & T_2 \vdash \texttt{case}_{\texttt{T},\texttt{F},\texttt{fun}} : (\mathsf{T} \rightarrow \mathsf{B}) \rightarrow (\mathsf{F} \rightarrow \mathsf{B}) \rightarrow & \\
 & \quad ((\mathsf{B} \rightarrow \mathsf{TA}) \rightarrow \mathsf{B}) \rightarrow & \\
 & \quad (\mathsf{B} + (\mathsf{B} \rightarrow \mathsf{TA})) \rightarrow \mathsf{B} & 22, \textsc{Inst} \\
24 & T_2 \vdash \texttt{case}_{\texttt{T},\texttt{F},\texttt{fun}}(\mathbf{C_1}) : (\mathsf{F} \rightarrow \mathsf{B}) \rightarrow & \\
 & \quad ((\mathsf{B} \rightarrow \mathsf{TA}) \rightarrow \mathsf{B}) \rightarrow & \\
 & \quad (\mathsf{B} + (\mathsf{B} \rightarrow \mathsf{TA})) \rightarrow \mathsf{B} & 3, 23, \textsc{App} \\
25 & T_2 \vdash \texttt{case}_{\texttt{T},\texttt{F},\texttt{fun}}(\mathbf{C_1})(\mathbf{C_2}) : (\mathsf{B} \rightarrow \mathsf{TA}) \rightarrow \mathsf{B}) \rightarrow & \\
 & \quad (\mathsf{B} + (\mathsf{B} \rightarrow \mathsf{TA})) \rightarrow \mathsf{B} & 6, 24, \textsc{App} \\
26 & T_2 \vdash \texttt{case}_{\texttt{T},\texttt{F},\texttt{fun}}(\mathbf{C_1})(\mathbf{C_2})(\mathbf{C_3}) : & \\
 & \quad (\mathsf{B} + (\mathsf{B} \rightarrow \mathsf{TA})) \rightarrow \mathsf{B} & 21, 25, \textsc{App} \\
27 & T_2 \vdash f : \mathsf{B} + (\mathsf{B} \rightarrow \mathsf{TA}) & \textsc{Ax} \\
28 & T_2 \vdash \texttt{case}_{\texttt{T},\texttt{F},\texttt{fun}}(\mathbf{C_1})(\mathbf{C_2})(\mathbf{C_3})(f) : \mathsf{B} & 26, 27, \textsc{App}
\end{array}
$$

Abbreviating $\texttt{case}_{\texttt{T},\texttt{F},\texttt{fun}}(\mathbf{C_1})(\mathbf{C_2})(\mathbf{C_3})(f)$ as $\mathbf{S}$ the derivation concludes:

$$
\begin{array}{lll}
29 & T_1 \vdash \lambda f.\mathbf{S} : & \\
 & \quad \mathsf{B} + (\mathsf{B} \rightarrow \mathsf{TA}) \rightarrow \mathsf{B} & 28, \textsc{Abs} \\
30 & T_0 \vdash \lambda U.\lambda f.\mathbf{S} : & \\
 & \quad (\mathsf{TA} \rightarrow \mathsf{B}) \rightarrow & \\
 & \quad (\mathsf{B} + (\mathsf{B} \rightarrow \mathsf{TA})) \rightarrow \mathsf{B} & 29, \textsc{Abs} \\
31 & T_0 \vdash \lambda U.\lambda f.\mathbf{S} : & \\
 & \quad (\mathsf{TA} \rightarrow \mathsf{B}) \rightarrow \mathsf{TA} \rightarrow \mathsf{B} & 30, \textsc{Sub}, \text{inf } \textsc{Fix1} \\
32 & T_0 \vdash \texttt{rec} : \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha & \textsc{Ax} \\
33 & T_0 \vdash \texttt{rec} : & \\
 & \quad (((\mathsf{TA} \rightarrow \mathsf{B}) \rightarrow \mathsf{TA} \rightarrow \mathsf{B}) \rightarrow & \\
 & \quad ((\mathsf{TA} \rightarrow \mathsf{B}) \rightarrow \mathsf{TA} \rightarrow \mathsf{B})) & \\
 & \qquad\quad \rightarrow & \\
 & \quad ((\mathsf{TA} \rightarrow \mathsf{B}) \rightarrow \mathsf{TA} \rightarrow \mathsf{B}) & 32, \textsc{Inst} \\
34 & T_0 \vdash \texttt{rec}(\lambda U.\lambda f.\mathbf{S}) : & \\
 & \quad (\mathsf{TA} \rightarrow \mathsf{B}) \rightarrow \mathsf{TA} \rightarrow \mathsf{B} &
\end{array}
$$

Hence, the inference system for **taut** can be read off line 34 of the deduction.

$$\mathbf{taut} : (\mathsf{TA} \rightarrow \mathsf{B}) \rightarrow \mathsf{TA} \rightarrow \mathsf{B}$$

2.8 Proofs for Chapter 2

2.8.1 Theorem 2.6 *(Formally Contractive implies Semantically Contractive)*

Let t be any expression formally contractive in x. Then F_t is contractive with respect to the MPS metric.

Proof We proceed by case analysis on the the structure of t (according to definition 2.17)

Case $\mathbf{t = c_0}$. In this case $F_t(I) = \mathcal{I}(D_{c_0})$, for any $I \in \mathcal{I}$. For any metric whatsoever, constants are contractive.

Case $\mathbf{t = x', x' \neq x}$. Again, this is a constant function, hence contractive.

Case $\mathbf{t = c_n(t_1, \ldots, t_n)}$. By lemma 2.1, these expressions yield contractive maps.

Case $\mathbf{t_1 + t_2}$, with $\mathbf{t_1, t_2}$ formally contractive . By induction, each of F_{t_1}, F_{t_2} is contractive. By theorem 2.5, $\cup(x, y)$ is non-expansive in both arguments. Furthermore, the composition of a non-expansive function with a contractive one is contractive.

Case $\mathbf{t = \texttt{fix}\ x'.t_1}$, with $\mathbf{t_1}$ contractive in both $\mathbf{x, x'}$. Since contractive in both arguments separately, MPS is an ultra-metric, so F_{t_1} is contractive. By theorem 2.5, $\mu\ F_{t_1}$ is contractive. But $\mu\ F_{t_1}$ is precisely $F_{\texttt{fix}\ x'.t_1}$. So the theorem holds for this case.

□

2.8.2 Theorem 2.7 *(Soundness of First Order Inference)*

A *regular term* is a synonym for a regular type expression corresponding to a term in $\Sigma(C, X)$.

To simplify notation, the following convenience prefaces theorem 2.7.

Definition 2.33 *(Extended $\mathcal{T}[\![\cdot]\!]$)* Let $\Lambda = \{T \mid T \text{ is a termexpression}\}$ be a set of regular term expressions. Then define the obvious:

$$\mathcal{T}[\![\Lambda]\!]\nu = \bigcup_{T\in\Lambda} \mathcal{T}[\![T]\!]\nu$$

By convention,

$$\mathcal{T}[\![\emptyset]\!]\nu = \{\perp\} = B.$$

The proof of theorem 2.7 depends on a the semantic function possessing a substitution property. Hence, we establish that the semantic function we use possesses this crucial property. Throughout, the symbol B denotes the trivial ideal $\{\perp\}$.

Lemma 2.3 *(Term Substitution Lemma)* Let A be a set of terms. Then

$$\mathcal{T}[\![A[x \leftarrow S]]\!]\nu = \mathcal{T}[\![A]\!]\nu[x \leftarrow \mathcal{T}[\![S]\!]\nu]$$

Proof If $A = \emptyset$, then $A[x \leftarrow S] = \emptyset$, and the lemma is trivially true. So, suppose A is non-empty. Let $t \in A$ be a term. We now show by structural induction on the shape of terms that

$$\mathcal{T}[\![t[x \leftarrow S]]\!]\nu = \mathcal{T}[\![t]\!]\nu[x \leftarrow \mathcal{T}[\![S]\!]\nu]$$

Since t is arbitrary, the lemma is shown.

Case $\mathbf{t = x}$. In this case $t[x \leftarrow S] = S$,so

$$\mathcal{T}[\![x[x \leftarrow S]]\!]\nu = \mathcal{T}[\![S]\!]\nu = \mathcal{T}[\![x]\!]\nu[x \leftarrow \mathcal{T}[\![S]\!]\nu]$$

Case $\mathbf{t = y, y \neq x}$. Here $t[x \leftarrow S] = t$, so the lemma is trivially true

Case $\mathbf{t = c, c \in C}$, $\text{Arity}(\mathbf{c}) = \mathbf{0}$. Same as above

Case $\mathbf{t = c_n(t_1, \ldots, t_n)}$. By the inductive hypothesis,

$$\mathcal{T}[\![t_i[x \leftarrow S]]\!]\nu = \\ \mathcal{T}[\![t_i]\!]\nu[x \leftarrow \mathcal{T}[\![S]\!]\nu]$$

So

$$\langle \mathcal{T}[\![t_1[x \leftarrow S]]\!]\nu, \ldots \mathcal{T}[\![t_n[x \leftarrow S]]\!]\nu\rangle = \\ \langle \mathcal{T}[\![t_1]\!]\nu[x \leftarrow \mathcal{T}[\![S]\!]\nu], \ldots, \mathcal{T}[\![t_n]\!]\nu[x \leftarrow \mathcal{T}[\![S]\!]\nu]\rangle$$

Consequently

$$\mathcal{I}(D_c)(\langle \mathcal{T}[\![t_1[x \leftarrow S]]\!]\nu, \ldots \mathcal{T}[\![t_n[x \leftarrow S]]\!]\nu\rangle) =$$
$$\mathcal{I}(D_c)(\langle \mathcal{T}[\![t_1]\!]\nu[x \leftarrow \mathcal{T}[\![S]\!]\nu], \ldots, \mathcal{T}[\![t_n]\!]\nu[x \leftarrow \mathcal{T}[\![S]\!]\nu]\rangle)$$

Rewriting the equality using definition 2.13, and definition 2.26

$$\mathcal{T}[\![c(t_1, \ldots, t_n)[x \leftarrow S]]\!]\nu = \mathcal{T}[\![c(t_1, \ldots, t_n)]\!]\nu[x \leftarrow S]$$

completing the proof for this case.

□

The MPS metric has the property that some sequences of non-telescoping sets may still converge in the metric sense. Lemma 2.4 indicates that a telescoping series that is also cauchy converges to the same limit.

Lemma 2.4 *($\sup I_k = \lim I_k$)* Let I_k be a cauchy sequence of ideals such that $I_n \subseteq I_{n+1}$. Then $I_n \subseteq \lim I_k$ for any n.

Proof Suppose not. Let I_m be an ideal in the sequence such that some element $v \in I_m, v \notin \lim I_k$. Furthermore, assume v is the minimum rank such element, of rank r. Since I_k is a cauchy sequence, select I_n such that $n > m$ and $d(I_n, \lim I_k) < 2^{-r}$. Since $n > m$, $I_m \subseteq I_n$, so $v \in I_n, v$ / $in \lim I_k$. The rank of v is r, so the rank of the distinguishing element of $I_m, \lim I_k$ must be less than r. This means that $d(I_n, \lim I_k) \geq 2^{-r}$. A contradiction. Consequently, no such I_m exists, and the lemma is proved. □

Theorem 2.7 *(Soundness of First Order Inference)* For any first order regular type expression T, $\mathcal{T}[\![\Sigma(T)]\!]\nu = \mathcal{T}[\![T]\!]\nu$.

Proof We use structural induction.

Case $\mathbf{T} = \mathbf{x}, \mathbf{x} \in \mathrm{TyVars}$.

Here $\Sigma(x) = \{x\}$, so $\mathcal{T}[\![\{x\}]\!]\nu = \mathcal{T}[\![x]\!]\nu$ by definition 2.33.

Case $\mathbf{T} = \mathbf{c_0}, \mathbf{c_0} \in \mathbf{M}, \mathrm{Arity}(\mathbf{c}) = \mathbf{0}$.

As for variables, $\mathcal{T}[\![\{c_0\}]\!]\nu = \mathcal{T}[\![c_0]\!]\nu$ by definition.

Case $\mathbf{T} = \mathbf{c_n}(\mathbf{T_1}, \ldots, \mathbf{T_n})$.

By the first order hypothesis, $c_n \neq \rightarrow$. Thus, it is sufficient to consider only tuples of ideals.

$$\begin{aligned} \mathcal{T}[\![\Sigma(T_i)]\!]\nu &= \mathcal{T}[\![T_i]\!]\nu, \text{ by ind. hyp.} \\ \langle \mathcal{T}[\![T_1]\!]\nu \ldots \mathcal{T}[\![T_n]\!]\nu \rangle &= \langle \mathcal{T}[\![\Sigma(T_1)]\!]\nu \ldots \mathcal{T}[\![\Sigma(T_n)]\!]\nu \rangle \\ \mathcal{I}(D_{\mathbf{c_n}})(\langle \mathcal{T}[\![T_1]\!]\nu \ldots \mathcal{T}[\![T_n]\!]\nu \rangle) &= \mathcal{I}(D_{\mathbf{c_n}})\langle \mathcal{T}[\![\Sigma(T_1)]\!]\nu \ldots \mathcal{T}[\![\Sigma(T_n)]\!]\nu \rangle \end{aligned}$$

Hence,

$$\mathcal{T}[\![\Sigma(c_n(T_1, \ldots, T_n))\nu]\!] = \mathcal{T}[\![c_n(T_1, \ldots, T_n]\!]\nu$$

Case $\mathbf{T} = \mathbf{T_1} + \mathbf{T_2}$.

By induction, $\mathcal{T}[\![T_i]\!]\nu = \mathcal{T}[\![\Sigma(T_i)]\!]\nu$, so $\mathcal{T}[\![T_1]\!]\nu \cup \mathcal{T}[\![T_2]\!]\nu = \mathcal{T}[\![\Sigma(T_1)]\!]\nu \cup \mathcal{T}[\![\Sigma(T_2)]\!]\nu$. And thus, $\mathcal{T}[\![T_1 + T_2]\!] = \mathcal{T}[\![\Sigma(T_1 + T_2)]\!]$

Case $\mathbf{T} = \texttt{fix}\ \mathbf{x}..\mathbf{T}'$.

By definition 2.15, $\Sigma(T) = \Sigma(T')^{*x}$. By definition 2.14, $\Sigma(T')^{*x} = \sup T'_i$, where T'_i is the Kleene sequence

$$\begin{aligned} T'_0 &= T'[x \leftarrow \emptyset] \\ T'_{i+1} &= T'[x \leftarrow T'_i] \end{aligned}$$

By lemma 2.3

$$\begin{aligned} \mathcal{T}[\![T_0]\!]\nu &= \mathcal{T}[\![T']\!]\nu[x \leftarrow \{\perp\}] \\ \mathcal{T}[\![T'_{i+1}]\!]\nu &= \mathcal{T}[\![T']\!]\nu[x \leftarrow \mathcal{T}[\![T_i]\!]\nu] \end{aligned}$$

Note that

$$\mathcal{T}[\![T_0]\!]\nu = F_{T'}(\{\perp\})$$

By simple induction, (using above as base case) we note

$$\mathcal{T}[\![T'_i]\!]\nu = F^{i+1}_{T'}(\{\perp\})$$

The right hand side of the above is a banach sequence for $\mu\ x.F_{T'}\nu$ beginning at $\{\perp\}$. Consequently,

$$\lim \mathcal{T}[\![T'_i]\!]\nu = \mu x.F_{T'}\nu = \mathcal{T}[\![\texttt{fix}\ x.T']\!]\nu$$

The sequence T_i' is known to be telescoping, so $\mathcal{T}[\![T_i']\!]\nu$ is a telescoping series of ideals. By previous analysis, it is a cauchy sequence, so

$$\sup \mathcal{T}[\![T_i']\!]\nu = \mathcal{T}[\![\mathtt{fix}\ x.T']\!]\nu$$

Definition 2.33, and elementary properties of sets gives

$$\sup \mathcal{T}[\![T_i']\!]\nu = \mathcal{T}[\![\sup T_i']\!]\nu$$

So

$$\mathcal{T}[\![\sup T_i']\!]\nu = \mathcal{T}[\![\mathtt{fix}\ x.T']\!]\nu$$

This is, by definition 2.14

$$\mathcal{T}[\![\Sigma(T'^{*x})]\!]\nu = \mathcal{T}[\![\Sigma(\mathtt{fix}\ x.T')]\!]\nu = \mathcal{T}[\![\mathtt{fix}\ x.T']\!]\nu$$

□

2.8.3 Theorem 2.8 *(Extended Subtype Soundness Theorem)*

The inference rules in definition 2.19 are sound.

Proof By structural induction on the shape of the proof.

Case $\mathbf{T_1 \subseteq T_2, T_1, T_2}$ first order This is theorem 2.7

Case $\mathbf{T \subseteq T}$ $\mathcal{T}[\![T]\!]\nu \subseteq \mathcal{T}[\![T]\!]\nu$.

Case $\mathbf{T_1 \subseteq T_1 + T_2}$ $\mathcal{T}[\![T_1]\!]\nu \subseteq \mathcal{T}[\![T_1]\!]\nu \cup \mathcal{T}[\![T_2]\!]\nu$

Case $\mathbf{T_1 \subseteq T_2, T_2 \subseteq T_3 \vdash T_1 \subseteq T_3}$ Again, by elementary properties of sets: If $\mathcal{T}[\![T_1]\!]\nu \subseteq \mathcal{T}[\![T_2]\!]\nu, \mathcal{T}[\![T_2]\!]\nu \subseteq \mathcal{T}[\![T_3]\!]\nu$, then $\mathcal{T}[\![T_1]\!]\nu \subseteq \mathcal{T}[\![T_3]\!]\nu$.

Case $\mathbf{T_3 \subseteq T_1, T_2 \subseteq T_4 \vdash (T_1 \to T_2) \subseteq T_3 \to T_4}$ By hypothesis for this case, and inductive hypothesis,

$$\mathcal{T}[\![T_3]\!]\nu \subseteq \mathcal{T}[\![T_1]\!]\nu, \mathcal{T}[\![T_2]\!]\nu \subseteq \mathcal{T}[\![T_4]\!]\nu$$

By definition 2.22,

$$\mathcal{T}[\![T_1 \to T_2]\!]\nu = \{f \mid f(\mathcal{T}[\![T_1]\!]\nu) \subseteq \mathcal{T}[\![T_2]\!]\nu\}$$

But since $\mathcal{T}[\![T_3]\!]\nu \subseteq \mathcal{T}[\![T_1]\!]\nu$, any f with $f(\mathcal{T}[\![T_1]\!]\nu) \subseteq \mathcal{T}[\![T_2]\!]\nu$ is also an f

$$f(\mathcal{T}[\![T_3]\!]\nu) \subseteq \mathcal{T}[\![T_2]\!]\nu$$

Likewise, since $\mathcal{T}[\![T_2]\!]\nu \subseteq \mathcal{T}[\![T_4]\!]\nu$, any f such that $f(\mathcal{T}[\![T_3]\!]\nu) \subseteq \mathcal{T}[\![T_2]\!]\nu$ is also an f

$$f(\mathcal{T}[\![T_3]\!]\nu) \subseteq \mathcal{T}[\![T_4]\!]\nu$$

Consequently,

$$f \in \{g \,|\, g(\mathcal{T}[\![T_3]\!]\nu) \subseteq \mathcal{T}[\![T_4]\!]\nu\}$$

So

$$\mathcal{T}[\![(T_1 \to T_2)]\!]\nu \subseteq \mathcal{T}[\![(T_3 \to T_4)]\!]\nu$$

Case $\texttt{fix}\ \mathbf{x}.\mathbf{T} = \mathbf{T}[\mathbf{x} \leftarrow \texttt{fix}\ \mathbf{x}.\mathbf{T}]$ By definition 2.26,

$$\mathcal{T}[\![\texttt{fix}\ x.T]\!]\nu = \mu F_T \nu$$

Similarly,

$$\begin{aligned} \mathcal{T}[\![T[x \leftarrow \texttt{fix}\ x.T]]\!]\nu &= \mathcal{T}[\![T]\!]\nu[x \leftarrow \mathcal{T}[\![\texttt{fix}\ x.T]\!]] \\ &= F_T\nu(\mu F_T \nu) \\ &= \mu F_T \nu \text{ by definition of fixed point} \end{aligned}$$

Case FIX2 The hypotheses for this rule is

$$t_1 \subseteq t_2 \vdash T_1[x \leftarrow t_1] \subseteq T_2[x \leftarrow t_2]$$

So, let $I_1 \subseteq I_2$ Since $T_1 \subseteq T_2$, the induction hypothesis yields

$$\mathcal{T}[\![T_1]\!]\nu[x \leftarrow I_1] \subseteq \mathcal{T}[\![T_2]\!]\nu[x \leftarrow I_2]$$

By definition 2.27

$$I_1 \subseteq I_2\ rmimplies\ F_{T_1}(I_1) \subseteq F_{T_2}(I_2)$$

This implies

$$F^n_{T_1}(I_1) \subseteq F^n_{T_2}(I_2)$$

Consequently,

$$\mu F_{T_1} \subseteq \mu F_{T_2}$$

Or,

$$\mathcal{T}[\![\texttt{fix}\ x.T_1]\!]\nu \subseteq \mathcal{T}[\![\texttt{fix}\ x.T_2]\!]$$

$\square$

2.8.4 Theorem 2.9 *(Soundness of Regular Type Inference)*

Let $A \vdash e : \sigma$ be any inference from definition 2.30. Let ρ be an environment respecting A, and let ν be any valuation. Then $\mathcal{E}[\![e]\!]\rho : \mathcal{T}[\![\sigma]\!]\nu$

Proof By structural induction on the last step of the inference.

Case TAUT By hypothesis, ρ respects A, so

$$\mathcal{E}[\![x]\!]\rho : \mathcal{T}[\![A(x)]\!]\nu$$

Case INST By rule requirements, $A \vdash e : \sigma$. So $\mathcal{E}[\![e]\!]\rho : \mathcal{T}[\![\sigma]\!]\nu$ by inductive hypothesis. Write $\sigma = \forall\vec{\alpha}.t, \sigma' = \forall\vec{\alpha}.t'$. This is possible by renaming bound variables. Then lemma 2.2 establishes that

$$\mathcal{T}[\![t]\!]\nu = \mathcal{T}[\![t']\!]\nu$$

Consequently, $v : \mathcal{T}[\![\sigma]\!]\nu$ implies $v : \mathcal{T}[\![\sigma']\!]\nu$.

Case GEN By inductive hypothesis,

$$\mathcal{E}[\![e]\!]\rho : \mathcal{T}[\![\sigma]\!]\nu$$

for all valuations ν. Since α is not free in A, any valuation $\nu[\alpha \leftarrow I]$ also maintains the theorem. In particular, let t be a regular monotype, and I_t the ideal $\mathcal{T}[\![t]\!]\nu$. Then

$$\mathcal{E}[\![e]\!]\rho : \mathcal{T}[\![\sigma]\!]\nu[\alpha \leftarrow I_t]$$

But t was arbitrary, so

$$\mathcal{E}[\![e]\!]\rho : \mathcal{T}[\![\forall\alpha\sigma]\!]\nu$$

Case SUB By inductive hypothesis

$$\mathcal{E}[\![e]\!]\rho : \mathcal{T}[\![\tau]\!]\nu$$

By theorem 2.8, if $\tau \subseteq \tau'$

$$\mathcal{T}[\![\tau]\!]\nu \subseteq \mathcal{T}[\![\tau']\!]\nu$$

Thus,

$$\mathcal{E}[\![e]\!]\rho : \mathcal{T}[\![\tau']\!]\nu$$

by transitivity of set inclusion.

Case Abst By inductive hypothesis,

$$\mathcal{E}[\![e_1]\!]\rho[x \leftarrow v : \mathcal{T}[\![\tau']\!]\nu] \in \mathcal{T}[\![\tau]\!]\nu$$

By definition 2.5

$$\mathcal{E}[\![\lambda\ x.e_1]\!]\rho(v) = \mathcal{E}[\![e_1]\!]\rho[x \leftarrow v]$$

Consequently

$$\mathcal{E}[\![\lambda\ x.e_1]\!]\rho(v) \in \mathcal{T}[\![\tau]\!]\nu$$

whenever $v \in \mathcal{T}[\![\tau']\!]\nu$. By definition 2.22, this means

$$\mathcal{E}[\![\lambda\ x.e_1]\!]\rho \in \mathcal{T}[\![\tau' \to \tau]\!]\nu$$

Case App By inductive hypothesis,

$$\begin{aligned} \mathcal{E}[\![f]\!]\rho &\in \mathcal{T}[\![\tau_1]\!]\nu \to \mathcal{T}[\![\tau_2]\!]\nu \\ \mathcal{E}[\![e]\!]\rho &\in \mathcal{T}[\![\tau_1]\!] \end{aligned}$$

By definition 2.22, any element $F \in \mathcal{T}[\![\tau_1]\!]\nu \to \mathcal{T}[\![tau_2]\!]\nu$ has the property

$$F(\mathcal{T}[\![\tau_1]\!]\nu) \subseteq \mathcal{T}[\![\tau_2]\!]\nu$$

Since $\mathcal{E}[\![f]\!]\rho$ is one such element

$$\mathcal{E}[\![f]\!]\rho(\mathcal{E}[\![e]\!]) \in \mathcal{T}[\![\tau_2]\!]$$

Case Let By the inductive hypothesis,

$$\mathcal{E}[\![e_1]\!]\rho : \mathcal{T}[\![\sigma]\!]\nu$$

From definition 2.5

$$\mathcal{E}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] = \mathcal{E}[\![e_2]\!]\rho[x \leftarrow \mathcal{E}[\![e_1]\!]\rho]$$

So $\rho[x \leftarrow \mathcal{E}[\![e_1]\!]\rho]$ respects the assumption

$$A \cup \{x : \sigma\}$$

Consequently, the inductive hypothesis applies to the second condition for this inference rule, and we have

$$\mathcal{E}[\![e_2]\!]\rho[x \leftarrow \mathcal{E}[\![e_1]\!]\rho] : \tau$$

Putting this together yields

$$\mathcal{E}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] : \tau$$

$\square$

Chapter 3

Automated Type Assignment for Regular Types

A major obstacle to programmer acceptance of statically typed languages is the requirement that the types of identifiers be declared. To remove this particular obstacle, some implementations provide an automatic method of assigning types to program expressions. Such a method is essential in a soft typing system. In fact, the importance is codified in the Minimal-Text Principle of chapter 1. Consequently, this chapter presents an automatic type assignment method for RegType, the type language of chapter 2.

The automatic type assignment algorithm presented in this chapter is based on Milner's type assignment algorithm [34] The Milner type assignment method can be decomposed into two separate tasks. First, all expressions are assigned a unique type variable, and the inference system generates a set of equality constraints necessary for the type inference to be valid. This task resembles generating verification conditions for programs. After task one is complete, task two finds the most general solution to the constraint equations, or reports failure if no solution exists.

The type language used by Milner, however, is not as expressive as RegType. Furthermore, the regular types do *not* fit the framework of equality constraints. The Sub rule (see definition 2.30) generates *in*equality constraints instead. The task of inequality constraint generation is not any more difficult than generating equality constraints, but the task of *solving* the inequalities is much more difficult than the corresponding task for equalities. Consequently, the type assignment task becomes one of solving inequalities.

One standard mathematical tool for handling inequalities is the introduction of *slack variables*. By adding slack variables, we can convert inequalities to equalities For example,

$$x < 5$$

can be changed into

$$x + S = 5$$

where S is the slack variable. Analogously, for the type assignment context, one converts

$$\tau \subseteq \mu$$

to

$$\tau \cup S = \mu$$

Using slack variables, we may convert the inequalities we generate for regular type assignment to equalities over type constructors, and the union operator. We say the resulting set of equalities is the *slack form* of the problem.

Regrettably, using slack variables still leaves the problem of finding solutions for the resulting slack form equalities. Our regret is that a general solution method for the slack form equalities is an open problem. To overcome the difficulty, we restrict the solution space by assuming the solutions have a certain special form. This restriction of the solution space, of course, excludes certain solutions. The restriction we impose, however, is mild, and retains most types of practical interest.

Our solution method adapts a a technique developed by D. Rémy for typing variant records in an object oriented language. By restricting the set of type expressions appropriately, Rémy's method can be applied to the slack form equalities discovered in the constraint generation phase.

All of these issues are detailed in the five sections of this chapter. Section 3.1 reviews the Milner type assignment algorithm. We include constraint generation in Section 3.2 continues the analysis by discussing inequality constraints, and slack variables, and difficulties with unrestricted slack variables. Next, section 3.3 indicates an appropriate restriction of the slack variable form. The solution of these restricted slack variable equalities uses Rémy's technique. To show the efficacy of the method, section 3.4 demonstrates that inferences produced using the restricted slack variable method can be translated to inferences using the inference rules for regular types (definition 2.30). The final section, section 3.6, illustrates the method with a few concrete examples.

3.1 The Milner Type System and algorithm W

One of the key properties of the Milner inference system is the decidability of deduction in that system. An algorithm exists to determine whether an expression can be proved to have some type. Furthermore, there is a best type for any expression having a type, and, moreover, there is an algorithm constructs computes the best type for a given expression (or fails when no type can be deduced for the expression).

Naturally, then, we would like to generalize the Milner algorithm to our type inference problem. Generalizing the algorithm to regular types, however, is non-trivial because the Milner system addresses a less expressive type language than the regular types. The Milner type language has no notion of union or recursive type.

Type inference for Exp under the Milner type language uses the rules from definition 2.29 (repeated here for convenience)

Recalling that A is a set of type assumptions about the types of free type variables, the inference rules are:

$$
\begin{array}{lcl}
\textsc{Taut:} & A \vdash x : \sigma & A(x) = \sigma \\
\textsc{Inst:} & \dfrac{A \vdash x : \sigma}{A \vdash x : \sigma'} & \sigma' < \sigma \\
\textsc{Gen:} & \dfrac{A \vdash e : \sigma}{A \vdash e : \forall \alpha \sigma} & \alpha \text{ not free in } A \\
\textsc{Abst:} & \dfrac{A \cup \{x : \tau'\} \vdash e_1 : \tau}{A \vdash \lambda x.e_1 : \tau' \rightarrow \tau)} & \\
\textsc{App:} & \dfrac{A \vdash f : \tau_1 \rightarrow \tau_2 \quad A \vdash e : \tau_1}{A \vdash (f\ e) : \tau_2} & \\
\textsc{Let:} & \dfrac{A \vdash e_1 : \sigma \quad A \cup \{x : \sigma\} \vdash e_2 : \tau}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} &
\end{array}
$$

Milner defines a denotational semantics for his language, and shows that his rules are sound with respect to the semantics.

Associated with the inference system is the type assignment algorithm, often called "algorithm W" in the literature. The version given by Damas and Milner[17] uses the unification algorithm of Robinson[40]. The symbol U denotes this unification algorithm. For further notational convenience $\overline{A}(t) = \forall \alpha_1 \ldots \alpha_n.t$ where the α_i are free in t, but not in A.

Definition 3.1 *(Algorithm W)* $W(A, e) = (S, \tau)$ where

i). if e is x, a variable or a constant, and $A(x) = \forall \alpha_1 \ldots \alpha_n \tau'$, then $S = \mathrm{Id}$ and $\tau = \tau'[\alpha_i \leftarrow \beta_i]$, where the β_i's are new.

ii). if e is $e_1 e_2$ then let $W(A, e_1) = (S_1, \tau_1)$ and $W(S_1 A, e_2) = (S_2, \tau_2)$, and $U(S_2 \tau_1, \tau_2 \rightarrow \beta) = V$, where β is new; Then $S = V S_2 S_1$ and $\tau = V\beta$.

iii). if e is $\lambda x.e_1$, let β be a new type variable. Let $W(A[x : \beta], e_1) = (S_1, \tau_1)$; Then $S = S_1$, and $\tau = S_1\beta \rightarrow t_1$.

iv). if $e is$`let` $x = e_1$ `in` e_2, then let $W(A, e_1) = (S_1, \tau_2)$ and $W(S_1A[x : \overline{S_1A}(\tau_1), e_2]) = (S_2, \tau_2)$; then $S = S_2S_1$ and $\tau = \tau_2$.

The algorithm fails whenever any of the above conditions are not met.

Damas[18] proves two theorems concerning algorithm W.

Theorem 3.1 *(Syntactic Soundness)* If $W(A, e)$ succeeds with (S, τ), then there is a derivation using the Milner rules of $SA \vdash e : \tau$.

Theorem 3.2 *(Syntactic Completeness)* Given A, e, let A' be an instance of A, and σ a type scheme such that

$$A' \vdash e : \sigma$$

Then

1. $W(A, e)$ succeeds
2. If $W(A, e) = (S, \tau)$ then, for some substitution R,

$$A' = RSA \text{ and } \sigma < R\overline{SA}(\tau)$$

where $<$ is the generic instance relation of definition 2.28

These two theorems demonstrate that algorithm W is exceptionally well suited to the Milner inference system. The syntactic soundness theorem indicates that any proof produced by the algorithm corresponds to a proof in the inference system. Syntactic completeness, on the other hand, states that any proof in the inference system can be generated automatically, up to generic instance. That is, given any proof in the inference system, the concomitant automated proof derives a type that has the original type as a generic instance. In other words, algorithm W generates the most general, or *principal* type.

We gain some additional insight into algorithm W derives from separating the type assignment process into two phases: a constraint phase, and a solution phase. Intuitively, type assignment can be viewed as giving all terms an unknown type, and using the inference system to derive constraints that must hold for a derivation to be valid. When the constraint phase is finished, one solves the resulting constraint equations. In algorithm W, the solution method is term unification.

To clarify the mechanics of this view of type assignment, consider example 3.1. This example indicates the constraint generation process in an informal way. For a more systemic treatment, see Wand[54].

Example 3.1 *(Constraint generation)* Consider the S combinator.

$$S = \lambda x.\lambda y.\lambda z.xz(yz)$$

Initially, assign type t_0 to S.

Considering the outermost lambda, if we assume $x : t_1$, and that term $\lambda y.\lambda z.xz(yz)$ has type t_2, then the ABST rule requires $t_0 = t_1 \rightarrow t_2$. Similarly, let $y : t_3$, and $\lambda z.xz(yz) : t_4$, then ABST

$$t_2 = t_3 \rightarrow t_4$$

And again, the if $z : t_5$ and $xz(yz) : t_6$, then

$$t_4 = t_5 \rightarrow t_6$$

Examining (xz), under the assumption that $(yz) : t_7$ gives

$$xz : t_7 \rightarrow t_6$$

The types of x and z are already assigned, so

$$t_1 = t_5 \rightarrow (t_7 \rightarrow t_6)$$

Also $yz : t_7$, using assigned types gives

$$t_3 = t_5 \rightarrow t_7$$

Gathering all these constraints together yields the set

$$\begin{aligned}
t_0 &= t_1 \rightarrow t_2 \\
t_2 &= t_3 \rightarrow t_4 \\
t_4 &= t_5 \rightarrow t_6 \\
t_1 &= t_5 \rightarrow (t_7 \rightarrow t_6) \\
t_3 &= t_5 \rightarrow t_7
\end{aligned}$$

Unification gives the solution

$$t_0 = (t_5 \rightarrow t_7 \rightarrow t_6) \rightarrow (t_5 \rightarrow t_7) \rightarrow (t_5 \rightarrow t_6)$$

Both example 3.1 and Damas-Milner use Robinson's unification algorithm as the solution mechanism for the system of equalities. Unification exactly characterizes solutions of equations over the free algebra $\Sigma(C, V)$. This algebra is also called the algebra of finite trees over (C, V). Consequently, the solutions to the systems of equations must, perforce, be finite trees. Some simple equations, however, cannot be solved when the solution space is restricted to finite trees. For example, $x = f(x)$ has no finite tree solution. To solve a larger class of equations, we must enlarge the space of possible solutions In particular, other algebras besides the finite trees may be candidate solution spaces.

One useful alternative solution space to consider is the *rational* trees. Rational trees include all the finite trees, and some infinite ones. They are characterized by

> **Definition 3.2** *(Rational Trees)* Let T be a tree, and let
>
> $$T_S = \{T_1, \ldots, T_i, \ldots\}$$
>
> be the set of non-isomorphic subtrees of T. Then T is *rational* if T_S is finite.

Obviously, any finite tree is certainly a rational tree, but some infinite trees are also rational. For example,

> **Example 3.2** *(An infinite rational tree)* Let $c \in C$ be an arity-2 constructor, and let $a \in C$ be a constant. Let T be the tree that has its root labeled with c, and its right son is a, and the left son is isomorphic to T. The set of non isomorphic subtrees of $T = \{a, T\}$, so T is rational. In fact, T is often written as a recursive equation: $T = c(T, a)$. Pictorially, T is shown in figure 3.1.

Since the rational trees contain the finite trees, any set of equations that has solutions over the algebra of finite trees will have solutions over the algebra of rational trees. Furthermore, some equations that are unsolvable over finite trees algebra are solvable over rational trees. Example 3.2 illustrates this property. The equation

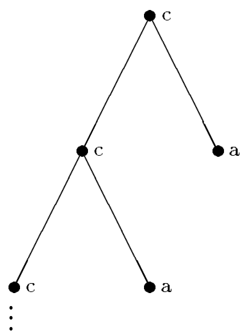

Figure 3.1 An example of a rational tree

$X = c(X, a)$ has no finite tree solution, but the indicated rational tree solves the equation.

Rational trees share an important property of finite trees. This property is the existence of an algorithm to solve systems of equations. For finite trees, Robinson's unification algorithm computes the most general solution. For rational trees, there is a variant of ordinary unification, called "circular" unification[14, 12, 33] that computes most general solutions. This algorithm makes rational trees viable for solving type constraint equations.

One must take care, however, to examine carefully the consequences of using a different algebra as a solution space. Changing solution spaces requires a change in the type language. In the case under discussion, the change is from finite trees over (C, X) to rational trees over (C, X). The natural question to ask, then, is do syntactic soundness and completeness still hold for Algorithm W with the new type language ? In the case of rational trees, the answer is yes. Rémy[39], building on results of Huet[28], proves that both syntactic soundness and syntactic completeness are valid for algorithm W, even when the type language is the rational trees over (C, X) and standard unification is replaced by circular unification.

To summarize, the Milner type assignment method can be viewed as two distinct processes: the generation of a set of constraint equations, and the solution of the equations. Consequently, there are two separate degrees of freedom for generalizing the Milner algorithm. This section focused on the generalizing the solution process. In particular, examination of the standard Robinson unification method suggested the use of rational trees and circular unification as a possible alternative. For this particular generalization, the syntactic soundness and completeness properties still hold.

Even though the extension of the Milner method to rational types is important, to handle regular types in any meaningful way requires an examination of the constraint *generation* component as well. That is the subject of the next section.

3.2 Equalities, Inequalities and Type Inference

We observed in an earlier section that type assignment in the context of subtyping involves the generation of *inequality* constraints. Intuitively, the use of the SUB introduces the inequalities. As a very simple illustration of the phenomenon, consider the typing for $\lambda x.x$ in example 3.3

Example 3.3 *(Inequality Constraints)* Proceeding as in example 3.1, we assign unknown types to all subterms of $\lambda x.x$, and carry on with a deduction. The assignment looks like

$$(\lambda x^{t_1}.x^{t_2})^{t_3}$$

Following the inference system in definition 2.30

$$\begin{array}{ll} \{x : t_1\} \vdash x : t_1 & \text{AX} \\ \{x : t_1\} \vdash x : t_2 & \text{SUB provided } t_1 \subseteq t_2 \end{array}$$

Consequently,

$$\begin{array}{lcl} \vdash \lambda x.x : & t_1 \rightarrow t_2 & \text{ABS} \\ \vdash \lambda x.x : & t_3 & \text{SUB provided } t_1 \subseteq t_2 \end{array}$$

Thus, $\lambda x.x : t_3$ as long as

$$\begin{array}{rcl} t_1 \rightarrow t_2 & \subseteq & t_3 \\ t_1 & \subseteq & t_2 \end{array}$$

There are many different approaches to handling the generated inequalities. Almost all of these strategies fall into one of the following two strategies:

Retained inequalities This strategy incorporates actual inequalities as part of the type expression

Convert to Equalities "Somehow" convert the inequalities to equalities, and use some appropriate method for solving the concomitant equalities.

The primary advocates of the retained inequalities method are Mitchell[36], Fuh and Mishra[21, 22, 23], and Curtis[15]. In this approach, some inequality constraints are retained as part of the type expression. For example, the identity function $\lambda x.x$ has a type described as

$$t, \{\alpha \subseteq \beta, (\alpha \rightarrow \beta) \subseteq t\}$$

The type expression informally reads as

> The function $\lambda x.x$ has type t, where t must contain a type $\alpha \rightarrow \beta$, and α must be contained in β

This method has several interesting properties, but it also has some features that make it unsuitable for soft typing:

1. The set of inequalities is potentially quite large, even for simple program expressions. This makes the type expression almost unreadable in some cases. Fuh and Mishra have developed techniques for reducing the set of inequalities in the type, but some program expressions simply do not have comprehensible type expressions.

2. In any event, *some* solution to the inequalities must be found to establish that the set of inequalities are consistent. For these systems, an inconsistent set of inequalities signals a type error

3. Few of these systems support parametric polymorphism.

These disadvantages indicate that the "convert to equalities" strategy is the right choice for soft typing systems. The conversion of inequalities to equalities suggests the venerable mathematical technique of "slack variables"[14]. The slack variable approach introduces auxiliary variables to "take up the slack" in the inequalities. To

[14]The idea of slack variables appears to be very old. The interested reader, however, may wish to examine Tapia's account of this technique in a different context from typechecking[46, 37]

see specifically how this applies to type inequalities, suppose a generated constraint requires

$$X \subseteq Y$$

Using the slack variable technique, the constraint may be expressed

$$Y = X \cup S$$

for S some new variable. S is the slack variable. The slack variable equations generated by regular type constraints may be alternatively notated as $Y = X + S$.

Equations generated in this fashion, however, cannot be solved using Robinson's unification method. Recall that Robinson's algorithm solves equations over a *free* term algebra. The algebra of regular type constraints, is not free, as it uses the + operator. The + operator is associative, commutative, and idempotent. Consequently, a solution method for equations over an associative, commutative, idempotent algebra is required.

The solution of equations over various algebraic theories is the subject of general unification theory[43]. For an associative, commutative, idempotent theory, general unification theory provides ACI unification[8, 30]. In principle, ACI unification could be used to generate solutions to regular type equations. This method, however, also has drawbacks:

1. There are multiple unifiers, as opposed to standard unification's single unifier. Hence, there are no principle types.

2. It is not clear how to modify the ACI unification algorithm to include "circularity". The circularity property is necessary to infer recursive types.

3. ACI theories do not have enough structure to capture the behavior of regular types. Some distributivity is required. For example $t(a) + t(b) = t(a + b)$ is valid in the theory of regular types, but this equality is not captured in the ACI theory. Furthermore, distributive, associative theories do *not* have a unification method associated with them.

The disadvantages associated with ACI unification force us to consider some alternative methods for solving regular type slack equations. The method that we propose places a mild restriction on the solution space. The restricted solution set has additional properties we can exploit to produce viable solutions to the slack equations. The restriction, and its associated solution method is the subject of the next section.

3.3 Slack Variables and the Rémy Encoding

As seen in section 3.2, solving inequalities using general slack variables is not completely acceptable. In cases where general methods are inconvenient, a standard mathematical strategy is to introduce a bit more "structure" into the solution space, and exploit the additional structure for solutions. For the problem at hand, we add our additional structure to the set of acceptable type expressions.

The structure we impose on the type expressions depends on one key observation: the number of type constructors is *finite*. Consequently, union types that have at most one outermost occurrence of any given constructor can be put into standard form by sorting the subterms. For example

> **Example 3.4** *(Standard form for union types)* Let a, b, c be type constructors, and let a be polymorphic, that is, $\text{Arity}(a) = 1$. Furthermore, let the sort order be exactly $a \leq b \leq c$. Then

Type Expression	Standard Form
$a(x) + c$	$a(x) + c$
$c + b$	$b + c$
$c + a(c + b)$	$a(b + c) + c$

We intend to limit slack variable equations to be of the form in example 3.4.

While formal arguments will appear shortly, the intuition inspiring this particular specialization of the slack variables is important. The idea is fairly simple: having some standard ordering obviates the need for associative and commutative complications, and the limitation to single occurrences of type constructors eliminates the idempotence consideration. Thus, finding solutions for slack variable equations in this form may require only standard unification.

To formalize the argument requires a firm definition of the type expressions proposed as the special forms. In particular, the notion of "occurrences" of a constructor is central. Hence,

> **Definition 3.3** *(Occurrences of a constructor)* Let C be a set of constructors, and $\Sigma(C, X)$ the set of terms over C with X as a set of variables. Let $+$ be 2-ary function symbol, $+ \notin C$. Let $t \in \Sigma(C \cup \{+\}, X)$. Then

define the syntactic function Occur(c,t) by

$$\text{Occur}(c,t) = \begin{cases} 1 & t = c(\ldots) \\ \text{Occur}(c,t_1) + \text{Occur}(c,t_2) & t = +(t_1,t_2) \\ 0 & \text{Otherwise} \end{cases}$$

The terms of interest have at most one occurrence of any constructor. Furthermore, each subterm has the same property. These are the *discriminative* terms.[15]

Definition 3.4 *(Discriminative terms)* Let C be a set of constructors, and let $+$ be a disjoint 2-ary function symbol. Then $t \in \Sigma(C \cup \{+\}, X)$ is discriminative iff for any subterm t' of t, and any $c \in C$, Occur$(c,t') \leq 1$.

To illustrating the definition, consider

Example 3.5 *(Discriminative, and non-discriminative types)* Let a, b, c be type constructors, with a having arity 1. Then the following terms are discriminative

$$\begin{gathered} a(x) + c \\ b + c \end{gathered}$$

On the other hand, all of the following terms are not discriminative

$$\begin{array}{cc} b + b & b \text{ used twice} \\ a(b) + a(c) & a \text{ used twice} \\ a(b+b) & b + b \text{ subterm not discriminative} \end{array}$$

With the definition of the discriminative terms in mind, there are two different situations in which they can be used to indicate solutions to inequalities, depending on whether constructors appear on the right side, or the left side of inequalities. Example 3.6 (really several examples in one) indicates some of the various possibilities.

Example 3.6 *(Using discriminative slack variables)* In this example, a is a 1-ary type constructor, b, and c are 0-ary type constructors, and x, y are variables. Then the inequality

$$x \subseteq a(b) + c$$

[15]This terminology is derived from a similar notion by Mishra and Reddy[35], which is discussed in chapter 5.

has the following three solutions

$$
\begin{aligned}
x &= a(b) \\
x &= c \\
x &= a(b) + c
\end{aligned}
$$

These solutions are obtained by simply eliminating components from the upper bound $a(b) + c$, and solving the corresponding equation.

On the other hand, solving the inequality

$$a(b) \subseteq x$$

is more complicated. In this case, we want to *add* components instead of take components away. For example, adding a b component yields one solution:

$$a(b) \subseteq x \Longrightarrow a(b) + b = x$$

Adding a c instead of a b yields another:

$$a(b) \subseteq x \Longrightarrow a(b) + c = x$$

In fact, any of the following equations are possible solutions to the original inequality

$$
\begin{aligned}
a(b) &= x \\
a(b) + b &= x \\
a(b) + c &= x \\
a(b) + b + c &= x
\end{aligned}
$$

The requirement of discriminativity is crucial for inequalities of this type. Without this requirement, then there are arbitrarily many solutions: $a(b) + a(c) = x$, $a(b) + a(a(b)) = x$, etc.

As example 3.6 suggests, restricting solutions to discriminative type expressions allows a systematic generation of of solutions not present in the unrestricted case. Even with the restriction, we need to develop succinct representations for arbitrary sets of incomparable types. fortunately, Didier Rémy has invented an ingenious encoding of multiple types as a single record type. His encoding allows us to try the various combinations "all at one time".

To get a glimpse of the intuition behind Rémy's technique, consider the following set of discriminative types:

$$a(b)$$
$$a(b) + b$$
$$a(b) + c$$
$$a(b) + b + c$$

Since the number of constructors is finite, the set of types might be expressed as a "table", indicating the presence or absence of a component:

	Status $a(b)$	Status b	Status c
$a(b)$	present	absent	absent
$a(b) + b$	present	present	absent
$a(b) + c$	present	absent	present
$a(b) + b + c$	present	present	present

The table enumerates all the discriminative supertypes of $a(b)$. The table could be simplified to one line:

Status $a(b)$	Status b	Status c
present	present or absent	present or absent

This representation can easily be extended to incorporate constructors with arguments. That is, for the same set of constructors, suppose we want the discriminative supertypes of b, then $a(b) + b$ is one, $a(c) + b$ is another, $a(a(c)) + b$ is yet another. So is $a(a(a \ldots)$. Clearly, there are an infinite number of discriminative supertypes. The trick to encoding this infinite number is to use a variable to cover all the structurally similar ones. In general, $a(x) + b$ is a supertype, for any x. In extended table form:

	Status a	Arg a	Status b	Status c
b	absent	x	present	absent
$a(x) + b$	present	x	present	absent
$b + c$	absent	x	present	present
$a(x) + b + c$	present	x	present	present

Reducing to a single line gives:

Status a	Arg a	Status b	Status c
present or absent	x	present	present or absent

The table representation works for discriminative subtypes as well. Consider the type $b + c$, then the discriminative subtypes are:

$$b$$
$$c$$
$$b + c$$

The full table representation is:

	Status a	Arg a	Status b	Status c
b	absent	x	present	absent
c	absent	x	absent	present
$b + c$	absent	x	present	present

Which condenses to the single line:

Status a	Arg a	Status b	Status c
absent	x	present or absent	present or absent

Observe that in supertype considerations, the positive, or "present" information is crucial, whereas for subtype considerations, negative, or "absent" information is crucial

The final technique in Rémy's representation is to encode single line tables by using additional *variables* to express the "present or absent" condition. Using this idea, the previous tables can be written as:

Supertypes of $a(b)$

Status a	Arg a	Status b	Status c
present	b	P_1	P_2

Supertypes of b

Status a	Arg a	Status b	Status c
P_3	x	present	P_4

Subtypes of $b + c$

Status a	Arg a	Status b	Status c
absent	x	P_5	P_6

To express this representation in algebraic form, we can formulate tables as terms over a single 4-ary constructor $\mathcal{R}$. To simplify notation, use the symbol ! for "present" and $-$ for "absent". Then write

Supertypes of $a(b)$

$$\mathcal{R}(!, b, P_1, P_2)$$

Supertypes of b

$$\mathcal{R}(P_3, x, !, P_4)$$

Subtypes of $b + c$

$$\mathcal{R}(-, x, P_5, P_6)$$

The beauty of Rémy's representation is that is reduces subtypes and supertypes to substitution instances of $\mathcal{R}$-terms. It eliminates the need for associative, commutative, and idempotent operators. The discriminative types are encoded in a free term algebra over $\{\mathcal{R}, !, -\}$. Being a free term algebra, regular unification is a viable solution method for equations over $\{\mathcal{R}, !, -\}$. As an added bonus, the Milner algorithm constitutes a method of both generating the constraints, and solving them. A rigorous treatment of this process appears in the next section.

3.4 Automating Type Assignment for Regular Types

Based on intuitions described in section 3.3, we prove the main result of this chapter:

> Type assignment for discriminative regular types can be accomplished using Algorithm W.

As discussed in section 3.3, discriminative regular types may be encoded using a special $\mathcal{R}$ constructor. Therefore, the general plan for type assignment consists of three parts:

1. Translate discriminative regular types for constants and constructors into the $\mathcal{R}$-terms
2. Use Algorithm W to do type assignment for types described by $\mathcal{R}$-terms.
3. Translate the $\mathcal{R}$-terms back to regular types

Each step is discussed in the sections that follow

3.4.1 Encoding discriminative regular types

To encode discriminative regular types as $\mathcal{R}$-terms we must rigorously define what constitutes an $\mathcal{R}$-term. The arity of the $\mathcal{R}$ function symbol on the specification of the programming language.

For the remainder of this treatment, let C be the set of type constructors derived from a language specification. Fix an ordering $C = \{c_1, \ldots, c_n\}$. The following definition formalizes the $\mathcal{R}$ function symbol

Definition 3.5 *(The $\mathcal{R}$ function symbol)* Let

$$N = \sum_i (\text{Arity}(c_i) + 1)$$

Then $\mathcal{R}$ is an N-ary function symbol

For each $c_i \in C$, there is a set of argument positions in an $\mathcal{R}$-term associated with c_i. Those positions consist of a *flag* position, and $\text{Arity}(c_i)$ argument positions. By convention, the flag position occurs before any of the argument positions. Thus,

Definition 3.6 *(Argument positions for a constructor)* Let the argument positions of the $\mathcal{R}$ function symbol be numbered from 1 to N. For any constructor $c_i \in C$, define the *flag* position f_i in $\mathcal{R}$ as:

$$f_i = \begin{cases} 1 & i = 1 \\ \sum_{j<i}(1 + \text{Arity}(c_j)) & \text{otherwise} \end{cases}$$

The argument positions for c_i extend from $f_i + 1$ to $f_i + \text{Arity}(c_i)$.

As a notational convenience, we define selector functions p_j by

$$p_j(\mathcal{R}(a_1, \ldots, a_j, a_{j+1}, \ldots, a_N) = a_j$$

Let $\{!, -\}$ be two constants. We distinguish F, where

$$F \subset \Sigma(\{\mathcal{R}, !, -\}, X)$$

called the *definite* $\mathcal{R}$-terms.

Definition 3.7 *(The definite $\mathcal{R}$-terms)* Let $r \in \Sigma(\{\mathcal{R}, !, -\}, X)$ be a term such that $p_{f_i}(r) =!$ or $p_{f_i}(r) = -$ for any flag position f_i. Then r is a *definite* $\mathcal{R}$ term. The set of all definite $\mathcal{R}$ terms is named F

As a notational convenience, let $\mathcal{R}_{k_1,\ldots,k_m}(\ldots) = (e_1, \ldots, e_m)$ be an identical $\mathcal{R}$-term, except that argument position k_i is e_i. That is,

$$p_j(\mathcal{R}_{k_1,\ldots,k_m}(\ldots) = (e_1, \ldots, e_m)) = \begin{cases} e_i & j = k_i \\ p_j(\mathcal{R}(\ldots)) & \text{otherwise} \end{cases}$$

Let $E \subset \Sigma(C, X)$ be the discriminative subset of the regular types. To make the definition of the translations somewhat easier, we note the following lemma:

Lemma 3.1 *(Leading constructor form for +)* Any discriminative regular term of the form $+(t_1, t_2)$ can be written in the form:

$$+(c_i(\ldots), t'_2)$$

Proof The proof proceeds by induction on the number of + symbols. The key observation is that + is associative.

Case n = 1 . The term must be $+(c_i(\ldots), c_j(\ldots))$. So the lemma is trivially true

Case n > 1 . Assume the lemma is true for all terms with less than n + symbols. Consider $+(t_1, t_2)$. Clearly, both t_1 has less than n + symbols. If t_1 has no + symbols, we are done. If t_1 has 1 or more, then the induction hypothesis gives $t_1 = +(c_i(\ldots), t'_1)$. So

$$\begin{aligned} +(t_1, t_2) &= +(+(c_i(\ldots), t'_1), t_2) \text{ (induction)} \\ &= +(c_i(\ldots), +(t'_1, t_2)) \text{ associativity} \end{aligned}$$

□

Corollary 3.1 *(Standard form for +)* Every regular term can be written so that any + subterm can be written

$$+(c_i(\ldots), t_2')$$

Proof Structural induction on term shape

Case $\mathbf{t = c_i(t_1, \ldots, t_m)}$. There is no outermost +, so, by inductive hypothesis, there are terms t_i' of the appropriate form, so $c(t_1', \ldots, t_m')$ satisfies the corollary.

Case $\mathbf{t = +(t_1, t_2)}$. By lemma, $t = +(c_i(u_1, \ldots, u_m), t_2')$. By induction, there are u_i',t_2'' satisfying the corollary. Hence, $+(c_i(u_1', \ldots, u_m'), t_2'')$ satisfies the corollary.

□

We define a map $R : E \rightarrow F$

Definition 3.8 *(Translating E to F)* Let $t \in E$, and let $X' \subseteq X$ be a set of variables that do not occur in t. Then define R, R_i by

$$R_i(t) = \begin{cases} x & t = x, x \in X \\ \langle !, R(t_1), R(t_2), \ldots, R(t_m)\rangle & \mathrm{Occur}(c_i(t_1, \ldots, t_m), t) = 1 \\ \langle -, x_1', \ldots, x_m'\rangle & \mathrm{Occur}(c_i, t) = 0 \end{cases}$$

$$R(t) = \mathcal{R}(R_1(t)||R_2(t)\ldots||R_n(t))$$

where the notation $A||B$ is a syntactic concatenation operator: that is

$$\langle a_1, \ldots, a_k\rangle||\langle b_1, \ldots, b_l\rangle \equiv \langle a_1, \ldots, a_k, b_1, \ldots, b_l\rangle$$

Note that R_i is well-defined. By definition 3.4,

$$\begin{array}{rcl} \mathrm{Occur}(c, t) & = & 1 \\ & \mathrm{OR} & \\ \mathrm{Occur}(c, t) = 0 & & \end{array}$$

Furthermore, since the depth of a term is finite, the mutually recursive definition is consistent.

An example will help clarify the definition of R.

Example 3.7 *(The use of R)* Let a, b, c be type constructors, with Arity$(a) = 1$, and the ordering as given. Then

$$R(a(b) + c) = \mathcal{R}(!, R(-, x_1', !, -), -, !)$$

Similarly,

$$R(b + c) = \mathcal{R}(-, x_2', !, !)$$

Finally,

$$R(a(a(c)) + b) = \mathcal{R}(!, \mathcal{R}(!, \mathcal{R}(-, x_3', -, !), -, -), !, -)$$

In a similar fashion, we define $C : F \to E$

Definition 3.9 *(Translating F to E)* Define a set of mutually recursive syntactic functions C, C_i Let $t = \mathcal{R}(\ldots)$ be an $\mathcal{R}$-term such that there exists an i for which $p_{f_i}(t) =!$

$$C_i(t) = c_i(C(p_{f_{i+1}}(t), \ldots, C(p_{f_{i+m}}(t))$$

$$C(t) = \sum_{p_{f_i}(t)=!} C_i(t)$$

where $\sum$ in this case is the analog summation for the regular expression $+$ symbol. Also set

$$C_i(x) = C(x) = x$$

We note that R and C are pseudo-inverses in the following precise sense:

Theorem 3.3 *(R and C are pseudo-inverse pairs)* Let $t \in E$. Then $C(R(t)) = t$.

Proof By induction on M, the total number of constructors c_i occurring in t.

Case M = 0 . If no constructors appear in t, t must be a variable. In this case, $C(R(x)) = x$ trivially

Case Inductive Part . Assuming the theorem is true for all terms having M or less constructors, Consider a term t with $M+1$ constructors. Using definition 3.8,

$$R(t) = \mathcal{R}(R_1(t)||\ldots R_n(t))$$

So suppose $\mathrm{Occur}(c_i(t_1,\ldots,t_n),t) = 1$. Then

$$R_i(t) = \langle !, R(t_1),\ldots,R(t_n)\rangle$$

This means that $p_{f_i}(R(t)) =!$. Also, by definition 3.9,

$$C_i(R(t)) = c_i(C(R(t_1),\ldots,C(R(t_n))))$$

But each term t_j has M or fewer total constructors, so by the inductive hypothesis

$$C_i(R(t)) = c_i(t_1,\ldots,t_n)$$

Thus, if

$$\mathrm{Occur}(c_i(t_1,\ldots,t_n),t) = 1$$

then

$$\mathrm{Occur}(c_i(t_1,\ldots,t_n),C(R(t)) = 1$$

Now suppose $\mathrm{Occur}(c_i(t_1,\ldots,t_n),t) = 0$. Then

$$R_i(t) = \langle -, x'_1,\ldots,x'_n\rangle$$

So $p_{f_i}(R(t)) = -$. Consequently,

$$\mathrm{Occur}(c_i(t_1,\ldots,t_n),C(R(t)) = 0$$

By discriminativity, $\mathrm{Occur}(c_i,t)$ can only be 0 or 1. Hence, the theorem is true for the inductive case as well.

$\square$

The definite $\mathcal{R}$-terms, however, fail to capture the notion of subtyping or supertyping. In section 3.3, we discovered (intuitively) that variables provide the key element for encoding the discriminative subtype / supertype behavior. To enhance our understanding of this process, we introduce two more variants of the R mapping – R_p and R_n. Both mappings have codomain $\Sigma(\{\mathcal{R},!,-\},X)$, rather than just the definite $\mathcal{R}$-terms. In all of the following definitions, the κ_i variables are type variables distinct from any other type variables labeled by x_i

Definition 3.10 *(The mapping R_p)* Let $t \in E$, and let $X' \subseteq X$ be a set of variables that do not occur in t. Furthermore, let f_j be a set of variables intended to have only $\{!, -\}$ substituted for them. Then define $R_p, R_{p,i}$

$$R_{p,i}(t) = \begin{cases} x & t = x, x \in X \\ \langle !, R_p(t_1), R_p(t_2), \ldots, R_p(t_m) \rangle & \text{Occur}(c_i(t_1, \ldots, t_m), t) = 1 \\ \langle \kappa_j, x'_1, \ldots, x'_m \rangle & \text{Occur}(c_i, t) = 0 \end{cases}$$

Each occurrence of κ_j in the above formula is distinct.

$$R_p(t) = \mathcal{R}(R_{p,1}(t)||R_{p,2}(t) \ldots ||R_{p,n}t))$$

where the notation $A||B$ is a syntactic concatenation operator

Definition 3.11 *(The mapping R_n)* Let $t \in E$, X' and P as in definition 3.10 Then define $R_n, R_{n,i}$

$$R_{n,i}(t) = \begin{cases} x & t = x, x \in X \\ \langle \kappa_j, R_n(t_1), R_n(t_2), \ldots, R_n(t_m) \rangle & \text{Occur}(c_i(t_1, \ldots, t_m), t) = 1 \\ \langle -, x'_1, \ldots, x'_m \rangle & \text{Occur}(c_i, t) = 0 \end{cases}$$

Each occurrence of κ_j in the above formula is distinct.

$$R_n(t) = \mathcal{R}(R_{n,1}(t)||R_{n,2}(t) \ldots ||R_{n,}t))$$

These two transformations transform subtyping and supertyping into substitution instances of the appropriate term. To prove the necessary result requires two stages: First, we establish the appropriate properties of these transformations for*monotonic* constructors. A monotonic constructor, recall, has the property that $u_1 \subseteq u_2$ implies $c_i(u_1) \subseteq c_i(u_2)$. Second, we extend the result to the function constructor $\rightarrow$. The function constructor is *not* monotonic, but an appropriate transformation can still be defined.

For the moment, the only relevant substitutions involve only the flag variables. The relevant form of substitution is defined by

Definition 3.12 *(Flag-defining substitution)* Let T be an $\mathcal{R}$-term, with flag variables $\{\kappa_j\}$. Then a *flag defining substitution* is a finite function $s : \{\kappa_j\} \rightarrow \{!, -\}$. Note that s assigns a value to all flags of T.

With this definition in mind, we proceed to analyze R_p and R_n with respect to monotonic constructors.

Theorem 3.4 *(R_p gives supertypes)* Let $t \in E$, $T = R_p(t)$. Assume every $c_i \in C$ is monotonic. Then for any flag defining substitution S for T, $t \subseteq C(S(T))$. Furthermore, if S' is any flag substitution, then $C(S(T)) \subseteq C(S(S'(T)))$

Proof We note that

$$S(R_p(t)) = \mathcal{R}(S(R_{p,1}(t))||S(R_{p,2}(t))\dots||S(R_{p,n}(t)))$$

so that

$$C(S(R_p(t))) = \sum_{f_i=!} C_i(S(R_p(t)))$$

By induction on M, the total number of constructors in t

Case $\mathbf{M = 0}$ In this case, $t = x$, for $x \in X$. So the theorem is trivially true.

Case $\mathbf{M = k + 1}$. Assume the theorem is true for any term having k or fewer constructors. Suppose Occur$(c_i(t_1, \dots, t_m), t) = 1$. Then

$$R_{p,i}(t) = \langle !, R_p(t_1), R_p(t_2), \dots, R_p(t_m)\rangle$$

by definition 3.10. Furthermore, note that the flag position is already set, so, $f_i =!$, consequently, no substitution alters it:

$$\begin{aligned} S(R_{p,i}(t)) &= \langle !, S(R_p(t_1)), S(R_p(t_2)), \dots, S(R_p(t_m))\rangle \\ S(S'(R_{p,i}(t))) &= \langle !, S(R_p(t_1)), S(R_p(t_2)), \dots, S(R_p(t_m))\rangle \end{aligned}$$

And so

$$\begin{aligned} C_i(S(R_{p,i}(t))) &= c_i(C(S(R_p(t_1))), \dots, C(S(R_p(t_m))) \\ C_i(S(S'(R_{p,i}(t)))) &= c_i(C(S(S'(R_p(t_1)))), \dots, C(S(S'(R_p(t_m)))) \end{aligned}$$

Each term t_1 has k or fewer constructors, so by the induction hypothesis,

$$t_j \subseteq C(S(R_p(t_j)) \text{ for any } j$$

By monotonicity of c_i,

$$c_i(t_1, \ldots, t_m) \subseteq C(S(R_p(c_i(t_1, \ldots, t_m))))$$

Thus, in this case, the theorem is true.

Now suppose $\mathrm{Occur}(c_i, t) = 0$. Then

$$R_{p,i}(t) = \langle \kappa_j, x'_1, \ldots, x'_m \rangle$$

If $S(\kappa_j) = -$, then $p_{f_i}(S(R_p(t))) = -$, so this term does not alter $C(S(R_p))$,so the theorem holds trivially for this case

Equally, if $S(\kappa_j) =!$, then $p_{f_i}(S(R_p(t))) =!$, so the term

$$c_i(C(S(R(x'_1))), \ldots, C(S(R(x'_m)))) = c_i(x'_1, \ldots, x'_m)$$

occurs in $C(S(R_p(t)))$. Consequently, the theorem holds for this case as well.

□

Virtually the same proof works for the R_n translation in relation to subtyping phenomenon.

Theorem 3.5 *(R_n gives subtypes)* Let $t \in E$, $T = R_n(t)$. Assume every $c_i \in C$ is monotonic. Then for any flag defining substitution S for T, $C(S(T)) \subseteq t$. Furthermore, $C(S(S'(T))) \subseteq C(S(T))$ for any flag substitution S'

Proof [sketch] By induction on M, the total number of constructors in t

Case $\mathbf{M = 0}$ t is a variable, so theorem is trivial

Case $\mathbf{M = k + 1}$. Assume the theorem is true for any term having k or fewer constructors. Suppose $\mathrm{Occur}(c_i(t_1, \ldots, t_m), t) = 1$. Then

$$R_{n,i}(t) = \langle \kappa_j, R_n(t_1), R_n(t_2), \ldots, R_n(t_m) \rangle$$

by definition 3.10. If $S(\kappa_j) = -$, then $p_{f_i}(S(R_n(t))) = -$, so this term does not appear in $C(S(R_n(t)))$, even though $\text{Occur}(c_i, t) = 1$. So the theorem holds trivially for this case. Now suppose $S(kappa_j) =!$

$$S(R_{n,i}(t)) = \langle !, S(R_n(t_1)), S(R_n(t_2)), \ldots, S(R_n(t_m)) \rangle$$

And so

$$C_i(S(R_{n,i}(t))) = c_i(C(S(R_n(t_1))), \ldots, C(S(R_n(t_m)))$$

Each term t_1 has k or fewer constructors, so by the induction hypothesis,

$$C(S(R_n(t_j)) \subseteq t_j \text{ for any } j$$

By monotonicity of c_i,

$$C(S(R_n(c_i(t_1, \ldots, t_m)))) \subseteq c_i(t_1, \ldots, t_m)$$

Thus, in this case, the theorem is true.

Now suppose $\text{Occur}(c_i, t) = 0$. Then

$$R_{n,i}(t) = \langle -, x'_1, \ldots, x'_m \rangle$$

Then $S(R_{n,i}(t)) = R_{n,i}(t)$. Furthermore, the $-$ flag assures that $\text{Occur}, (c_i, C(S(R_n(t)))$. So the theorem holds in this case as well.

□

Now that the appropriate properties are established for monotonic constructors, the extension to the function constructor $\rightarrow$ can be established. Since $\rightarrow$ is a two place function constructor, with the first argument being *anti*monotonic. Consequently, to get supertyping behavior for the entire term requires subtype behavior in the first argument, and supertype behavior in the second argument. Formalizing this intuition gives

Definition 3.13 *(Extended R_p and R_n)* Define $R_{p,i}, R_{n,i}, R_p, R_n$ as follows:

$$R_{p,i}(t) \quad = \quad \begin{cases} x & t = x, x \in X \\ \langle !, R_p(t_1), \ldots, R_p(t_m) \rangle & \text{Occur}(c_i(t_1, \ldots, t_m), t) = 1, \\ & c_i \neq \rightarrow \\ \langle !, R_n(t_1), \ldots, R_p(t_m) \rangle & \text{Occur}(t_1 \rightarrow t_2, t) = 1 \\ \langle \kappa_j, x'_1, \ldots, x'_m \rangle & \text{Occur}(c_i, t) = 0 \end{cases}$$

$$
\begin{aligned}
R_{n,i}(t) &= \begin{cases} x & t = x, x \in X \\ \langle \kappa_j, R_n(t_1), \ldots, R_n(t_m)\rangle & \text{Occur}(c_i(t_1, \ldots, t_m), t) = 1 \\ & c_i \neq \rightarrow \\ \langle \kappa_j, R_p(t_1), R_n(t_2)\rangle & \text{Occur}(t_1 \rightarrow t_m, t) = 1 \\ \langle -, x'_1, \ldots, x'_m\rangle & \text{Occur}(c_i, t) = 0 \end{cases} \\
R_p(t) &= \mathcal{R}(R_{p,1}(t) || R_{p,2}(t) \ldots || R_{p,n} t)) \\
R_n(t) &= \mathcal{R}(R_{n,1}(t) || R_{n,2}(t) \ldots || R_{n,} t))
\end{aligned}
$$

As before, each κ_j is unique.

The extended R_p, R_n retain the supertype/subtype properties of the original transformations.

Theorem 3.6 *(Extended Properties of R_p,R_n)*

Let $t \in E$, $T_p = R_p(t), T_n = R_n(t)$. Assume every $c_i \neq \rightarrow \in C$ is monotonic, $\rightarrow$ is antimonotonic in its first argument, but monotonic in its second. Then for any flag defining substitution S for T, $C(S(T_n)) \subseteq t$, and $t \subseteq C(S(T_p))$. Similarly, $C(S(S'(T_n))) \subseteq C(S(T_n))$, and $C(S(S'(T_p))) \subseteq C(S(T_p))$.

Proof It suffices to consider the case when $\text{Occur}(t_1 \rightarrow t_2, t)$, otherwise, previous arguments hold. Also, note that the induction proofs for theorem 3.4 and theorem 3.5 hold here as well. So, we consider the inductive case.

Assume the theorem holds for all t with k or fewer constructors, and consider a system with $k+1$ constructors. Note that for this proof, the inductive property holds for *both* R_p, and R_n.

Looking at R_p first,

$$R_{p,\rightarrow}(t) = \langle !, R_n(t_1), R_p(t_2)\rangle$$

so

$$S(R_{p,\rightarrow}(t)) = \langle !, S(R_n(t_1)), S(R_p(t_2))\rangle$$

So $C(S(R_n(t_1)) \rightarrow C(S(R_p(t_2))$ appears in $C(S(T_p))$. By inductive hypothesis,

$$C(S(R_n(t_1)) \subseteq t_1$$

and

$$t_2 \subseteq C(S(R_p(t_2))$$

Thus, by antimonotonicity, monotonic properties of $\rightarrow$, $(t_1 \rightarrow t_2) \subseteq C(S(R_n(t_1)) \rightarrow C(S(R_p(t_2))$, so $t \subseteq C(S(T_p)$.

Similarly,

$$R_{n,\rightarrow}(t) = \langle \kappa_j, R_p(t_1), R_n(t_2) \rangle$$

If $S(\kappa_j) = -$, then the theorem holds trivially. So suppose $S(\kappa_j) =!$. Then

$$S(R_{n,\rightarrow}(t)) = \langle !, S(R_p(t_1)), S(R_n(t_2)) \rangle$$

And $C(S(R_p(t_1)) \rightarrow C(S(R_n(t_2))$ appears in $C(S(T_p))$. By inductive hypothesis, $t_1 \subseteq C(S(R_p(t_1))$ and $C(S(R_n(t_2)) \subseteq t_2$. Again, by antimonotonicity, monotonicity properties of the function constructor $\rightarrow$,

$$C(S(R_p(t_1)) \rightarrow C(S(R_n(t_2)) \subseteq t_1 \rightarrow t_2$$

and the theorem holds. □

3.5 Accommodating Recursive Types

Theorems 3.3,3.4,3.5, and 3.6 form the foundation for doing regular type inference via translation to $\mathcal{R}$-terms. None of these theorems, however, addressed the `fix` types. To extend the results to `fix` types requires extending $\mathcal{R}$-terms to *rational* $\mathcal{R}$-terms. Moreover, only a certain kind of rational $\mathcal{R}$-terms is useful. These are the rational terms that do not have any flag position as a `fix` bound variable. That is,

Definition 3.14 *(The free-flag $\mathcal{R}$-terms)* Let $T = \mathtt{fix}\, t.\mathcal{R}(f_1, t_1, \ldots, t_n)$ be a rational $\mathcal{R}$-term. Then T is *bound-flag* $\mathcal{R}$-term iff $t \equiv f_i$ for some flag position i, where $\equiv$ means syntactically identical. If T is not a bound-flag term then T is a *free-flag* $\mathcal{R}$-term. By convention, any finite $\mathcal{R}$-term is flag-free.

We can extend R_p^*, R_n^* to accommodate discriminative `fix` regular types in an obvious fashion.

Definition 3.15 *(R^*_p, R^*_n for `fix` types)*

$$R^*_p(t) = \begin{cases} \texttt{fix}\, t.R^*_p t_1 & t = \texttt{fix}\, t.t_1 \\ R_p(t) & \text{otherwise} \end{cases}$$

$$R_n(t) = \begin{cases} \texttt{fix}\, t.R^*_n t_1 & t = \texttt{fix}\, t.t_1 \\ R_n(t) & \text{otherwise} \end{cases}$$

The use of `fix` for defining rational trees, as well as for recursive regular types is, hopefully, intuitive rather than confusing.

Likewise, the C transformation may be easily extended to free-flag $\mathcal{R}$-terms

Definition 3.16 *(C^*, translating flag free rational $\mathcal{R}$-terms)* Let T be a rational $\mathcal{R}$-term. Define

$$C^*(T) = \begin{cases} \texttt{fix}\, t.C^*(t_1) & T = \texttt{fix}\, t.t_1 \\ C(T) & \text{otherwise} \end{cases}$$

The extended $R^*_p, andR^*_n$ retain the supertype/subtype properties.

Theorem 3.7 *($C^*(R^*_p(t))$ is a supertype of t, R^*_n is a subtype)* Let t be a discriminative, rational term. Let S be a flag defining substitution. Let S' be any flag substitution. Then:

i. $t \subseteq C^*(S(R^*_p(t)))$

ii. $C^*(S(R^*_n(t))) \subseteq t$

Proof By induction on the number of `fix` operators

Case 0 operators In this case,

$$R^*_p(t) = R_p(t)$$
$$R^*_n(t) = R_n(t)$$

Consequently,

$$t \subseteq C(S(R^*_p(t)))$$
$$C(S(R^*_n(t))) \subseteq t$$

By definition 3.15, and the assumption of no `fix` operators in t, neither $R_p(t)$ nor $R_n(t)$ contain any `fix` operators. Consequently,

$$\begin{aligned} C(S(R_p^*(t))) &= C^*(S(R_p^*(t))) \\ C(S(R_n^*(t))) &= C^*(S(R_n^*(t))) \end{aligned}$$

So the theorem is proved for this case

Case $k+1$ `fix` operators Suppose $t = \mathtt{fix}\ x.t_1$. Then

$$\begin{aligned} R_p^*(t) &= \mathtt{fix}\ x.R_p^*(t_1) \\ R_n^*(t) &= \mathtt{fix}\ x.R_n^*(t_1) \end{aligned}$$

So

$$\begin{aligned} S(R_p^*(t)) &= \mathtt{fix}\ x.S(R_p^*(t_1)) \\ S(R_n^*(t)) &= \mathtt{fix}\ x.S(R_n^*(t_1)) \end{aligned}$$

And

$$\begin{aligned} C^*(S(R_p^*(t))) &= \mathtt{fix}\ x.C^*(S(R_p^*(t_1))) \\ C^*(S(R_n^*(t))) &= \mathtt{fix}\ x.C^*(S(R_n^*(t_1))) \end{aligned}$$

By induction,

$$\begin{aligned} t_1 &\subseteq C^*(S(R_p^*(t_1))) \\ C^*(S(R_n^*(t_1))) &\subseteq t_1 \end{aligned}$$

Consequently, by the Fix2 rule of definition 2.19,

$$\begin{aligned} \mathtt{fix}\ x.t_1 &\subseteq \mathtt{fix}\ x.C^*(S(R_p^*(t_1))) \\ \mathtt{fix}\ x.C^*(S(R_n^*(t_1))) &\subseteq \mathtt{fix}\ x.t_1 \end{aligned}$$

By definition 3.16,

$$\begin{aligned} \mathtt{fix}\ x.t_1 &\subseteq C^*(S(R_p^*(\mathtt{fix}\ x.t_1))) \\ C^*(S(R_n^*(\mathtt{fix}\ x.t_1))) &\subseteq \mathtt{fix}\ x.t_1 \end{aligned}$$

And the theorem is proved

$\square$

3.5.1 Using Algorithm W for $\mathcal{R}$-type assignment

The second step of the strategy for discriminative type assignment employs algorithm W to assign $\mathcal{R}$ types to program expressions. The function of Algorithm W is to generate proofs in the inference system of definition 2.29. Given such a proof over the $\mathcal{R}$ term algebra, we seek a method for transforming it into a proof about regular types, using the inference system from definition 2.30. The two inference systems are similar, differing only in the SUB rule for regular types.

To maintain clarity, we will subscript the $\vdash$ symbol to indicate the appropriate inference system:

$$\begin{array}{ll} A \vdash_M e : \mathcal{R}(\ldots) & \text{For inferences using defn 2.29} \\ A \vdash_R e : \sigma & \text{For inferences using defn 2.30} \end{array}$$

The main idea behind the translation is that a proof using $\mathcal{R}$ terms may correspond to many different regular type proofs. An $\mathcal{R}$ term encodes many different regular types, and different instantiations of $\mathcal{R}$ proofs may require different proof trees in the regular type inference system.

We extend R_p^*, C^* to type schemes in the obvious fashion

Definition 3.17 *(R_p^*, C^* for type schemes)* Let $\sigma = \forall v_1 \ldots v_l.\tau$ be a regular type scheme. Then define

$$R_p^*(\sigma) = \forall f_1 \ldots f_m v_1 \ldots v_l.R_p^*(\tau)$$

where each f_i is a flag variable in $R_p^*(\tau)$. Similarly, let

$$\nu = \forall f_1 \ldots f_m v_1 \ldots v_l.\mu_R$$

be a $\mathcal{R}$ type scheme. Let S_f be a flag defining substitution for μ_R. Then

$$C^*(S_f(\nu)) = \forall v_1 \ldots v_l.S_f(\mu_R)$$

Finally, if A is a set of regular type assumptions,

$$x : \forall f_i R_p^*(\sigma) \in R_p^*(A) \text{ iff } x : \sigma \in A$$

For purposes of simplifying the presentation of the inference translation method, we make some assumptions:

1. If $\forall f_1 \ldots f_l v_1 \ldots .\tau$, and $\forall g_1 \ldots g_q v_1 \ldots .\tau$ are both $\mathcal{R}$ type schemes, with f_i, g_j as flag variables, then $\{f_i\} \cap \{g_j\} = \emptyset$.

2. A flag defining substitution for $\forall f_1 \ldots f_l v_1 \ldots . \tau$ is a flag defining substitution for τ.

The main theorem states that any flag defining instance of a $\vdash_M$ inference can be converted to a $\vdash_R$ inference.

Theorem 3.8 *(Regular type inference via $\mathcal{R}$-term inference)* Suppose

$$R_p^*(A) \vdash_M e : \sigma$$

Then for any flag defining substitution S_f for σ

$$A \vdash_R e : C^*(S_f(\sigma))$$

Proof By structural induction on the last line of the proof tree in $\vdash_M$

Case TAUT In this case

$$R_p(A)(x) = \sigma = \forall f_i v_j R_p(\tau)$$

by definition 3.17. By theorem 3.7, any S_f for τ has the property

$$\tau \subseteq C^*(S_f(R_p(\tau)))$$

Hence, for any S_f, given $R_p^*(A) \vdash_M e : \sigma$ by TAUT, construct

$$\begin{array}{lll} L1 & A \vdash_R x : \forall v_j . \tau & \textsc{Taut} \\ L2 & A \vdash_R x : \forall v_j . C^*(S_f(R_p(\tau))) & L1, \textsc{Sub} \end{array}$$

Case INST In this case, the hypothesis is

$$R_p(A) \vdash_M e : \sigma, \quad \sigma' < \sigma$$

And the conclusion

$$R_p(A) \vdash e : \sigma'$$

Let $\sigma' = S(\sigma)$ for some substitution of bound variables. Then S may be decomposed into $S_v \circ S_f'$, where S_v applies to argument variables, and S'_f applies to flag variables. Such a decomposition is possible because flag and argument variables are disjoint. Furthermore, $S_f \circ S'_f$ is still a flag defining substitution for σ. Also note that S_v is an ordinary substitution for argument variables. By induction hypothesis, for any S_f for σ, there exists an inference such that

$$A \vdash_R e : C^*(S_f(\sigma)))$$

In particular, for any $S_f \circ S'_f$, there is a deduction

$$A \vdash_R e : C^*(S_f \circ S'_f(\sigma)))$$

By definition,

$$S_v(C^*(S_f \circ S'_f(\sigma)))) \leq C^*(S_f \circ S'_f(\sigma)))$$

Consequently, for any S_f, the deduction

$$\begin{array}{lll} L1 & A \vdash e : C^*(S_f \circ S'_f(\sigma)) & \text{induction hyp} \\ & S_v(C^*(S_f \circ S'_f(\sigma)))) \leq C^*(S_f \circ S'_f(\sigma))) & \text{defn} \\ L2 & A \vdash e : S_v(C^*(S_f \circ S'_f(\sigma)))) & L1, \textsc{Inst} \end{array}$$

But, since flag variables and argument variables are disjoint

$$\begin{aligned} S_v(C^*(S_f \circ S'_f(\sigma)))) &= C^*(S_f \circ (S_v \circ S'_f(\sigma)))) \\ &= C^*(S_f(S(\sigma))) \\ &= C^*(S_f(\sigma')) \end{aligned}$$

Thus, $L2$ can be written

$$L2 \quad A \vdash e : C^*(S_f(\sigma')) \quad L1, \textsc{Inst}$$

So the theorem is true for this case

Case GEN For this case we have the hypothesis

$$R_p(A) \vdash_M e : \sigma$$

and that α does not occur free in $R_p(A)$. The conclusion is

$$R_p(A) \vdash_M e : \forall \alpha . \sigma$$

There are two cases to distinguish. First suppose α is a flag variable. By induction hypothesis, for any S_f there is an inference

$$A \vdash_R e : C^*(S_f(\sigma))$$

By hypothesis, the flag variable α appears in σ, and for any assignment of α, there is an inference as above. Hence, by definition 3.17, any S_f for σ is also a S_f for $\forall\alpha.\sigma$. Thus, the inference constructed by hypothesis also works here. Similarly, if α is an argument variable, by definition of R_p, α free in $R_p(A)$ means α free in A. so the inference

$L1$	$A \vdash_R e : C^*(S_f(\sigma))$	Inductive hyp.
$L2$	$A \vdash_R e : C^*(S_f(\forall\alpha.\sigma))$	$L1$, Inst

Case Abst In this case, we have

$$\frac{R_p(A)[x : \tau] \vdash_M e : \tau'}{R_p(A) \vdash_M \lambda x.e : \tau \rightarrow \tau'}$$

By the substitution property for $\vdash_M$, for any F

$$\frac{(F(R_p(A)[x : \tau])) \vdash_M e : F(\tau')}{R_p(A) \vdash_M \lambda x.e : \tau \rightarrow \tau'}$$

But τ is a *type*, not a type scheme, so any free flag variables in τ are unquantified, and may be disjoint from from the flag variables in any type scheme of $R_p(A)$. Consequently,

$$\frac{((R_p(A)[x : F(\tau)])) \vdash_M e : F(\tau')}{R_p(A) \vdash_M \lambda x.e : \tau \rightarrow \tau'}$$

By inductive hypothesis for any S_f, there is an inference

$$A[x : C^*(S_f(\tau))] \vdash_R e : C^*(S_f(\tau'))$$

So, construct proof tree

$L1$	$A[x : C^*(S_f(\tau))] \vdash_R e : C^*(S_f(\tau'))$	Inductive hyp.
$L2$	$A \vdash_R \lambda x.e : C^*(S_f(\tau)) \rightarrow C^*(S_f(\tau'))$	Abst

But $L2$ can be written as

$$A \vdash_R \lambda x.e : C^*(S_f(\tau \rightarrow \tau'))$$

So the theorem holds in this case

Case App The deduction on the $\vdash_M$ side reads

$$\frac{R_p(A) \vdash_M f : \tau_1 \rightarrow \tau_2 \quad R_p(A) \vdash_M e : \tau_1}{R_p(A) \vdash_M (f\ e) : \tau_2}$$

By induction, for any S_f, there must be inferences

$$A \vdash_R f : C^*(S_f(\tau_1 \rightarrow \tau_2))$$

and

$$A \vdash_R e : C^*(S_f(\tau_1))$$

So we construct the inference

$L1$	$A \vdash_R f : C^*(S_f(\tau_1 \rightarrow \tau_2))$	Inductive hyp.
$L2$	$A \vdash_R e : C^*(S_f(\tau_1))$	Inductive hyp.
$L3$	$A \vdash_R (f\ e) : C^*(S_f(\tau_2))$	$L1, L2,$ App

proving the theorem for this case

Case Let The inference for $\vdash_M$ must be

$$\frac{R_p(A) \vdash_M e_1 : \sigma \quad R_p(A)[x : \sigma] \vdash_M e_2 : \tau}{R_p(A) \vdash_M \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

From this, construct the following $\vdash_R$ inference for any S_f

$L1$	$A \vdash_R e_1 : C^*(S_f(\sigma))$	Inductive hyp.
$L2$	$A[x : C^*(S_f(\sigma)] \vdash_M e_2 : C^*(S_f(\tau))$	Inductive hyp.
$L3$	$A \vdash_M \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau$	$L1, L2,$ Let

$\square$

Theorem 3.8 justifies the use of algorithm W for type assignment.

Corollary 3.2 *(Algorithm W yields regular type assignment)* Let A be a set of regular type assumptions, and suppose $W(R_p(A), e) = (\sigma, S)$. Then for any flag defining substitution S_f, there is a deduction

$$S(A) \vdash_R e : C^*(S_f(\sigma))$$

Proof By syntactic soundness for algorithm W, there is an inference

$$S(R_p(A)) \vdash_M e : \sigma$$

But $R_p(A)$ contains no free flag variables, so $S(R_p(A)) = R_p(S(A))$ By theorem 3.8, there is a deduction

$$S(A) \vdash_R e : C^*(S_f(\sigma))$$

$\square$

3.5.2 Interpreting $\mathcal{R}$-terms as sets of regular types

Ideally, we would like to translate any $\mathcal{R}$ type derived from algorithm W back into a discriminative regular type. Regrettably, this is not always possible. Some combinations of Inst and Sub do not yield single regular types as results.

Example 3.8 *(A non uniform deduction)* Consider a function $f_1 : (\alpha \to \alpha) \to \alpha \to \alpha$, and a function $f_2 : a + b \to a$. Consider the following deduction for $(f_1 f_2) : a \to a$.

1	$T_0 \vdash_R f_1 : (\alpha \to \alpha) \to \alpha \to \alpha$	Taut
2	$T_0 \vdash_R f_1 : (a \to a) \to a \to a$	$1,$ Inst
3	$T_0 \vdash_R f_2 : a + b \to a$	Taut
4	$T_0 \vdash_R f_2 : a \to a$	$3,$ Sub
5	$T_0 \vdash (f_1 f_2) : a \to a$	$2, 4,$ App$L$

Another deduction, however, yields

1	$T_0 \vdash_R f_1 : (\alpha \to \alpha) \to \alpha \to \alpha$	Taut
2	$T_0 \vdash_R f_1 : (a + b \to a + b) \to a + b \to a + b$	$1,$ Inst
3	$T_0 \vdash_R f_2 : a + b \to a$	Taut
4	$T_0 \vdash_R f_2 : a + b \to a + b$	$3,$ Sub
5	$T_0 \vdash (f_1 f_2) : a + b \to a + b$	$2, 4,$ App$L$

The types $a + b \to a + b$ and $a \to a$ are incomparable.

The inferences above indicates the futility of hoping for a "principal" discriminative regular type as the outcome of the translation process in all cases. Consequently, the best that can be said of type assignment process it that it always yields a *set* of regular types. If $R_p(A) \vdash_M e : \sigma$, then $e : \{\nu \mid \nu = C^*(S_f(\sigma))\}$ where S_f ranges over the entire range of flag defining substitutions.

The worst case, however, appears to be infrequent. In most examples, the type assignment yields a single regular type. Examples of both phenomena appear in section 3.6.

3.6 Some Examples

To ease the presentation, some notational conventions for $\mathcal{R}$ types are useful. Instead of positional notation, we will *label* the position. For example, the type table

Status a	Arg a	Status b	Status c
f_1	x	!	f_2

corresponding to

$$\mathcal{R}(f_1, x, !, f_2)$$

can be written using labels as

$$[f_1.a(x) \mid !.b \mid f_2.c]$$

The labels give us the freedom to reorder constructor names.

Also, since in most cases, the actual names of the flag variables are unimportant for understanding the type, Consequently, these will be given a ".". In rare cases where the actual flag variable cannot be anonymous, we will provide it. Modifying our example for this convention yields

$$[.a(x) \mid !.b \mid .c]$$

As a final simplification, we note that most flag variables are either ! or $-$, depending on whether R_p or R_n has been used to generate them. Notationally, we indicate "the rest are variable" with "...", and "the rest are $-$" by closing the bracket. For our running example, the convention yields

$$[!.b \mid \ldots]$$

For an example of the $-$ convention, consider the function f with regular type $b+c \to b$. Then the abbreviated notation for this function is (with `fn` being the $\to$ constructor label)

$$[!.\mathtt{fn}([.b \mid .c], [!.b \mid \ldots]) \mid \ldots]$$

To illustrate the workings of algorithm W, we indicate the depth of recursive calls by a level number. For application, there are two separate recursive calls, so we show these by level n and n'. We show the type output via a line "$=$ type". The substitution component is shown implicitly in subsequent type environments

Example 3.9 *(Non uniform* `if` *)* Consider the function from example 1.3

$$\lambda\ x.\mathtt{if}\ x\ \mathtt{then}\ 1\ \mathtt{else}\ \mathtt{nil}$$

To simplify this example, we will assume

$$\mathtt{if} : \mathsf{true} + \mathsf{false} \to \alpha \to \alpha \to \alpha$$

The bracket convention expresses this as

$$\mathtt{if} : [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}], [!\mathsf{fn}(\alpha, [!\mathsf{fn}(\alpha, \alpha) \mid] \ldots) \mid \ldots]) \mid \ldots]$$

The initial type assumption A_0 is

$$A_0 = \begin{array}{cl} \mathtt{if} & : \quad t_{\mathrm{if}} \text{ as above} \\ 1 & : \quad [!\mathsf{suc} \mid \ldots] \\ \mathtt{nil} & : [!\mathsf{nil} \mid \ldots] \end{array}$$

Stepping thru the algorithm, the initial recursive calls find the innermost function application

$$\begin{array}{lll} 0 & A_0 & \lambda\ x.\mathtt{if}\ x\ \mathtt{then}\ 1\ \mathtt{else}\ \mathtt{nil} \\ 1 & A_0[x:\tau] & (((\mathtt{if}\ x)\ 1)\ \mathtt{nil}) \\ 2 & A_0[x:\tau] & ((\mathtt{if}\ x)\ 1) \\ 3 & A_0[x:\tau] & (\mathtt{if}\ x) \\ 4 & A_0[x:\tau] & \mathtt{if} \\ 4 & = & \\ & \multicolumn{2}{l}{[!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}],} \\ & \multicolumn{2}{l}{[!\mathsf{fn}(\alpha, [!\mathsf{fn}(\alpha, \alpha) \mid] \ldots) \mid \ldots]) \mid \ldots]} \end{array}$$

Then, the argument for this application is checked:

$$
\begin{array}{rll}
3' & A_0[x:\tau] & x \\
3' & = & \tau \\
4,3' & \text{Unify} & \left\{ \begin{array}{l} [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}], [!\mathsf{fn}(\alpha, [!\mathsf{fn}(\alpha,\alpha) \mid \ldots]) \mid \ldots]) \mid \ldots] \\ [!\mathsf{fn}(\tau, \beta_1)] \end{array} \right. \\
4,3' & = & \\
 & \tau : & [.\mathsf{true} \mid .\mathsf{false}] \\
 & \beta_1 : & [!\mathsf{fn}(\alpha, [!\mathsf{fn}(\alpha,\alpha) \mid \ldots]) \mid \ldots] \\
3 & = & \beta_1
\end{array}
$$

The type assumptions are now:

$$
A_1 = A_0 \left[\begin{array}{c} x : [.\mathsf{true} \mid .\mathsf{false}] \\ \beta_1 : [!\mathsf{fn}(\alpha, [!\mathsf{fn}(\alpha,\alpha) \mid \ldots]) \mid \ldots] \end{array} \right]
$$

So,

$$
\begin{array}{rl}
2' & A_1 \quad 1 \\
2' & = \\
2' & [!\mathsf{suc} \mid \ldots] \\
3,2' & \text{Unify} \left\{ \begin{array}{c} [!\mathsf{fn}(\alpha, [!\mathsf{fn}(\alpha,\alpha) \mid \ldots]) \mid \ldots] \\ [!\mathsf{fn}([!\mathsf{suc} \mid \ldots], \beta_2)] \end{array} \right. \\
3,2' & = \\
3,2' & \alpha : [!\mathsf{suc} \mid \ldots] \\
 & \beta_2 : [!\mathsf{fn}(\alpha,\alpha) \mid \ldots]
\end{array}
$$

Letting

$$
A_2 = A_1 \left[\begin{array}{c} \alpha : [!\mathsf{suc} \mid \ldots] \\ \beta_2 : [!\mathsf{fn}(\alpha,\alpha) \mid \ldots] \end{array} \right]
$$

we continue with

$$
\begin{array}{rl}
2 & = \quad \beta_2 \\
1' & A_2 \;\; \texttt{nil} \\
1' & = \\
1' & [!\mathsf{nil} \mid \ldots] \\
2,1' & \text{Unify} \begin{cases} [!\mathsf{fn}(\alpha,\alpha) \mid \ldots] \\ [!\mathsf{fn}([!\mathsf{nil} \mid \ldots], \beta_3) \end{cases} \\
2,1' & = \\
2,1' & \text{Unify} \begin{cases} [!\mathsf{suc} \mid \ldots] \\ [!\mathsf{nil} \mid \ldots] \end{cases} \\
2,1' & = \\
2,1' & \alpha : [!\mathsf{suc} \mid !\mathsf{nil} \mid \ldots] \\
 & \beta_3 : [!\mathsf{suc} \mid !\mathsf{nil} \mid \ldots] \\
1 & = \\
 & \beta_3
\end{array}
$$

Now, using the λ rule gives

$$
\begin{array}{rl}
0 & = \\
0 & [!\mathsf{fn}(\tau, \beta_3) \mid \ldots]
\end{array}
$$

Using the built up substitutions, and re-interpreting the bracket notation, the algorithm deduces:

$$\lambda x.\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else}\ \texttt{nil} : \mathsf{true} + \mathsf{false} \rightarrow \mathsf{suc} + \mathsf{nil}$$

In a similar fashion, Algorithm W deduces a satisfactory solution for non-uniform lists, as in example 1.4.

Example 3.10 *(Non-homogeneous list)* The function

$$\lambda x.((\texttt{cons}\ 1)\ x)$$

can be applied safely to any list. So, the application

$$((\lambda x.((\texttt{cons}\ 1)\ x))\ \text{`}(\texttt{true}\ \texttt{false}))$$

is well-typed using union types. We trace the execution of Algorithm W for this example. To simplify the illustration, we will assume that the constant list given as an argument has type

$$\text{`(}\texttt{true}, \texttt{false}) : [!\mathsf{cons}([!\mathsf{true}\ |!\mathsf{false} \mid \ldots]) \mid \ldots]$$

We will also abbreviate

$$[!\mathsf{true}\ |!\mathsf{false} \mid \ldots]$$

as $[!\mathsf{bool} \mid \ldots]$ The constructor `cons` is 2-ary, and is treated as polymorphic in its first argument. So, the initial type assumption is

$$\begin{array}{rcl} \texttt{cons} & : & [!\mathsf{fn}(\alpha, [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha)], [!\mathsf{cons}(\alpha) \mid \ldots]) \mid \ldots]) \mid \ldots] \\ 1 & : & [!\mathsf{suc} \mid \ldots] \end{array}$$

Type assignment for the outermost application performs type assignment on the λ-expression first :

$$\begin{array}{rll}
0 & A_0 & ((\lambda x.((\texttt{cons}\ 1)\ x))\ \text{`}(\texttt{true}\ \texttt{false})) \\
1 & A_0 & \lambda x.((\texttt{cons}\ 1)\ x) \\
2 & A_0[x:\tau] & ((\texttt{cons}\ 1)\ x) \\
3 & A_0[x:\tau] & (\texttt{cons}\ 1) \\
4 & A_0[x:\tau] & \texttt{cons} \\
4 & = & \\
 & \multicolumn{2}{l}{[!\mathsf{fn}(\alpha, [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha)], [!\mathsf{cons}(\alpha \mid \ldots]) \mid \ldots]) \mid \ldots]} \\
3' & A_0[x:\tau] & 1 \\
3' & = & \\
 & [!\mathsf{suc} \mid \ldots] & \\
4,3' & \text{Unify} & \left\{ \begin{array}{c} [!\mathsf{fn}(\alpha, [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha)], [!\mathsf{cons}(\alpha) \mid \ldots]) \mid \ldots]) \mid \ldots] \\ [!\mathsf{fn}([!\mathsf{suc} \mid \ldots], \beta_1)] \end{array} \right. \\
4,3' & = & \\
 & \multicolumn{2}{l}{\alpha \leftarrow [!\mathsf{suc} \mid \ldots]} \\
 & \multicolumn{2}{l}{\beta_1 \leftarrow [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha)], [!\mathsf{cons}(\alpha) \mid \ldots]) \mid \ldots]} \\
3 & = & \\
 & \beta_1 &
\end{array}$$

Altering $A_0[x:\tau]$ to reflect the new substitutions

$$A_1 = A_0[x:\tau] \left[\begin{array}{c} \alpha \leftarrow [!\mathsf{suc} \mid \ldots] \\ \beta_1 \leftarrow [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha)], [!\mathsf{cons}(\alpha) \mid \ldots]) \mid \ldots] \end{array} \right]$$

Continuing

$$\begin{array}{rl}
2' & A_1 \quad x \\
2' & = \\
 & \tau \\
3,2' & \text{Unify}\left\{\begin{array}{c} [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha)], [!\mathsf{cons}(\alpha) \mid \ldots]) \mid \ldots] \\ [!\mathsf{fn}(\tau, \beta_2)] \end{array}\right. \\
3,2' & = \\
 & \tau \leftarrow [.\mathsf{nil} \mid .\mathsf{cons}(\alpha)] \\
 & \beta_2 \leftarrow [!\mathsf{cons}(\alpha) \mid \ldots] \\
2 & = \\
 & \beta_2
\end{array}$$

At this point the type assumption $A_0[x:\tau]$ is altered to :

$$A_2 = A_0[x:\tau]\left[\begin{array}{c} \alpha \leftarrow [!\mathsf{suc} \mid \ldots] \\ \beta_1 \leftarrow [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha)], [!\mathsf{cons}(\alpha) \mid \ldots]) \mid \ldots] \\ \tau \leftarrow [.\mathsf{nil} \mid .\mathsf{cons}(\alpha)] \\ \beta_2 \leftarrow [!\mathsf{cons}(\alpha) \mid \ldots] \end{array}\right]$$

So that the function rule gives

$$\begin{array}{rl}
1 & = \\
 & [!\mathsf{fn}(\tau, \beta_2) \mid \ldots] \\
 & = \\
 & [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha)], [!\mathsf{cons}(\alpha) \mid \ldots]) \mid \ldots]
\end{array}$$

where the type for the λ expression has been partially rewritten. By assumption, the type for the argument gives

$$0' \quad A_0 \quad [!\mathsf{cons}([!\mathsf{bool} \mid \ldots]) \mid \ldots]$$

so the unification step gives

$$\begin{array}{rl}
0',1 & \text{Unify}\left\{\begin{array}{c} [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha)], [!\mathsf{cons}(\alpha) \mid \ldots]) \mid \ldots] \\ [!\mathsf{fn}([!\mathsf{cons}([!\mathsf{bool} \mid \ldots]), \beta_3) \mid \ldots]] \end{array}\right. \\
0',1 & = \\
 & \beta_3 \leftarrow [!\mathsf{cons}([!\mathsf{bool} \mid !\mathsf{suc} \mid \ldots]) \mid \ldots] \\
0 & = \\
 & [!\mathsf{cons}([!\mathsf{bool} \mid !\mathsf{suc} \mid \ldots]) \mid \ldots]
\end{array}$$

after using the substitution value for α. Consequently, Algorithm W deduces

$$((\lambda x.((\texttt{cons}\ 1)\ x))\ `(\texttt{true false})) : \mathsf{cons}(\mathsf{true} + \mathsf{false} + \mathsf{suc})$$

after removing abbreviations, and re-interpreting the bracket terms.

As a final example, we follow the automated type assignment process for a fairly complicated function. The example function is recursive, and higher order, so all of the features of regular typing come into play. The purpose of the function is to determine when arbitrary boolean functions are tautologies – that is, are **`true`** regardless of their inputs. The function `taut` serves this purpose

```
taut =
λB.case B of
     true :  true
     false:  false
     fn : ((and (taut (B true))) (taut (B false)))
```

The function `taut` examines its input value to determine if it is a boolean constant or a function. If it is a boolean constant, then the constant itself determines tautologous nature — `true` is always a tautology, and `false` never is. On the other hand, if the input to `taut` is a function, then the function is a tautology iff it is tautologous on both boolean values. The function branch of the `case` statement performs the appropriate test for boolean functions. Example 3.11 indicates how the automated inference system handles this function.

Example 3.11 *(Tautology Function)* To perform the type inference, we must "de-sugar" the taut function:

```
taut =
(rec λT.λB.
     ((((case
          λd.true)
          λd.false)
          λd.((and (T (d true))) (T (d false)))
     B )
)
```

where `case` abbreviates $\texttt{case}_{\mathsf{true},\mathsf{false},\rightarrow}$. The initial type assumption for this problem includes the assumptions for `case` and `rec` as well as `true` and `false` The initial assumption A_0 is

$$
\begin{aligned}
\texttt{case} \;:\; & [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{true} \mid \ldots], \alpha)], \\
& \quad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{false} \mid \ldots], \alpha)], \\
& \quad\quad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{fn}(\gamma, \delta) \mid \ldots], \\
& \quad\quad\quad \alpha)], [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false} \mid .\mathsf{fn}(\gamma, \delta)], \\
& \quad\quad\quad\quad \alpha) \mid \ldots]) \mid \ldots]) \mid \ldots]) \mid \ldots] \\
\texttt{rec} \;:\; & [!\mathsf{fn}([!\mathsf{fn}(\alpha, \alpha) \mid \ldots], \alpha) \mid \ldots] \\
\texttt{and} \;:\; & [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}], \\
& \quad [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}], [!\mathsf{true} \mid !\mathsf{false} \mid \ldots]) \mid \ldots]) \mid \ldots] \\
\texttt{true} \;:\; & [!\mathsf{true} \mid \ldots] \\
\texttt{false} \;:\; & [!\mathsf{false} \mid \ldots]
\end{aligned}
$$

Note that α is a generic variable in both `case` and `rec`. Generic renaming will normally be indicated subscripting or priming. After discovering the type for `rec`, the main work for type assignment lies in discovering the types for the lambda expression argument to `rec`

0	A_0	$(\texttt{rec}\ \lambda T.\lambda B.\ldots)$
1	A_0	`rec`
1	$=$	
	$[!\mathsf{fn}([!\mathsf{fn}\alpha', \alpha' \mid \ldots], \alpha') \mid \ldots]$	
$1'$	A_0	$\lambda T.\lambda B.\ldots$
$1'$	$A_0[T : \tau_1]$	$\lambda B.\ldots$
2	$A_0[T : \tau_1][B : \tau_2]$	$(\texttt{case}\ldots)$

Note that we renamed the generic variable α to α' in the type for `rec`. Abbreviating $A_0[T:\tau_1][B:\tau_2]$ as A for simplicity, we continue

$$
\begin{array}{lll}
3 & A & ((\texttt{case}\ \lambda d.\texttt{true})\ldots \\
4 & A & \texttt{case} \\
4 & = & \\
 & \multicolumn{2}{l}{[!\mathsf{fn}([!\mathsf{fn}([!\mathsf{true} \mid \ldots], \alpha)],} \\
 & \multicolumn{2}{l}{\quad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{false} \mid \ldots], \alpha)],} \\
 & \multicolumn{2}{l}{\qquad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{fn}(\gamma,\delta) \mid \ldots],} \\
 & \multicolumn{2}{l}{\qquad\quad \alpha)], [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false} \mid .\mathsf{fn}(\gamma,\delta)],} \\
 & \multicolumn{2}{l}{\qquad\qquad \alpha) \mid \ldots]) \mid \ldots]) \mid \ldots]) \mid \ldots]} \\
3' & A & \lambda d.\texttt{true} \\
4' & A[d:\tau_3] & \texttt{true} \\
4' & = & \\
 & [!\mathsf{true} \mid \ldots] & \\
3' & = & \\
 & [!\mathsf{fn}(\tau_3, [!\mathsf{true} \mid \ldots]) \mid \ldots] &
\end{array}
$$

$$
4, 3' \quad \text{Unify} \left\{ \begin{array}{l}
\left(\begin{array}{l}
[!\mathsf{fn}([!\mathsf{fn}([!\mathsf{true} \mid \ldots], \alpha)], \\
\quad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{false} \mid \ldots], \alpha)], \\
\qquad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{fn}(\gamma,\delta) \mid \ldots], \\
\qquad\quad \alpha)], [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false} \mid .\mathsf{fn}(\gamma,\delta)], \\
\qquad\qquad \alpha) \mid \ldots]) \mid \ldots]) \mid \ldots]) \mid \ldots]
\end{array} \right) \\
\quad [!\mathsf{fn}([!\mathsf{fn}(\tau_3, [!\mathsf{true} \mid \ldots]) \mid \ldots], \beta_1)]
\end{array} \right.
$$

$$
\begin{array}{ll}
 & = \\
 & \tau_3 \leftarrow [!\mathsf{true} \mid \ldots] \\
 & \alpha \leftarrow [!\mathsf{true} \mid \ldots] \\
 & \beta_1 \leftarrow \left(\begin{array}{l}
[!\mathsf{fn}([!\mathsf{fn}([!\mathsf{false} \mid \ldots], \alpha)], \\
\quad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{fn}(\gamma,\delta) \mid \ldots], \\
\qquad \alpha)], [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false} \mid .\mathsf{fn}(\gamma,\delta)], \\
\qquad\quad \alpha) \mid \ldots]) \mid \ldots]) \mid \ldots]
\end{array} \right) \\
3 & = \\
 & \beta_1
\end{array}
$$

The `false` arm of the `case` statement is similar:

$$
\begin{array}{rll}
3'' & A & ((\texttt{case}\ \ldots)\ \lambda d.\texttt{false})\ldots \\
4'' & A & \lambda d.\texttt{false} \\
5 & A[d:\tau_4] & \texttt{false} \\
5 & = & \\
 & [!\mathsf{false} \mid \ldots] & \\
4'' & = & \\
 & [!\mathsf{fn}(\tau_4, [!\mathsf{false} \mid \ldots]) \mid \ldots] & \\
3, 4'' & \text{Unify} & \left\{ \begin{array}{l} \left(\begin{array}{l} [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{false} \mid \ldots], \alpha)], \\ \quad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{fn}(\gamma, \delta) \mid \ldots], \\ \qquad \alpha)], [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false} \mid .\mathsf{fn}(\gamma, \delta)], \\ \qquad\quad \alpha) \mid \ldots]) \mid \ldots]) \mid \ldots] \end{array} \right) \\ \quad [!\mathsf{fn}([!\mathsf{fn}(\tau_4, [!\mathsf{false} \mid \ldots]) \mid \ldots], \beta_2)] \end{array} \right. \\
3, 4'' & = & \\
 & \tau_4 \leftarrow [!\mathsf{false} \mid \ldots] & \\
 & \alpha \leftarrow [!\mathsf{true} \mid !\mathsf{false} \mid \ldots] & \\
 & \beta_2 \leftarrow \left(\begin{array}{l} [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{fn}(\gamma, \delta) \mid \ldots], \\ \quad \alpha)], [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false} \mid .\mathsf{fn}(\gamma, \delta)], \\ \qquad \alpha) \mid \ldots]) \mid \ldots] \end{array} \right) & \\
3'' & = & \\
 & \beta_2 &
\end{array}
$$

Note that the α instance for the `case` construct is updated as more information is acquired.

The fn arm of the case statement is substantially more complicated than its predecessors.

$$
\begin{array}{rll}
3''' & A & (\ldots(\texttt{case}\ldots)\ \lambda d.((\texttt{and}\ (T\ (d\ \texttt{true}))\ldots \\
4''' & A & \lambda d.((\texttt{and}\ (T\ (d\ \texttt{true}))\ldots \\
5 & A[d:\tau_5] & ((((\texttt{and}\ (T\ (d\ \texttt{true}))\ldots \\
6 & A[d:\tau_5] & \texttt{and} \\
6 & = & \\
 & \multicolumn{2}{l}{[!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}],} \\
 & \multicolumn{2}{l}{\quad [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}], [!\mathsf{true} \mid !\mathsf{false} \mid \ldots]) \mid \ldots]) \mid \ldots]} \\
6' & A[d:\tau_5] & (T\ (d\ \ldots \\
7 & A[d:\tau_5] & T \\
7 & = & \\
 & \tau_1 & \\
7' & A[d:\tau_5] & (d\ \texttt{true}) \\
8 & A[d:\tau_5] & d \\
8 & = & \\
 & \tau_5 & \\
8' & A[d:\tau_5] & \texttt{true} \\
8' & = & \\
 & [!\mathsf{true} \mid \ldots] & \\
8, 8' & \text{Unify} \left\{ \begin{array}{c} \tau_5 \\ [!\mathsf{fn}([!\mathsf{true} \mid \ldots], \beta_3)] \end{array} \right. & \\
 & = & \\
 & \multicolumn{2}{l}{\tau_5 \leftarrow [!\mathsf{fn}([!\mathsf{true} \mid \ldots], \beta_3)]} \\
7' & = & \\
 & \beta_3 &
\end{array}
$$

Continuing:

$$
\begin{array}{rl}
7,7' & \text{Unify}\begin{cases} \quad \tau_1 \\ [!\mathsf{fn}(\beta_3,\beta_4)] \end{cases} \\
 & = \\
 & \tau_1 \leftarrow [!\mathsf{fn}(\beta_3,\beta_4)] \\
6' & = \\
 & \beta_4 \\
6,6' & \text{Unify}\begin{cases} [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}], \\ \qquad [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}],[!\mathsf{true} \mid !\mathsf{false} \mid \ldots]) \mid \ldots]) \mid \ldots] \\ \qquad\qquad\qquad [!\mathsf{fn}(\beta_4,\beta_5)] \end{cases} \\
 & = \\
 & \beta_4 \leftarrow [.\mathsf{true} \mid .\mathsf{false}] \\
 & \beta_5 \leftarrow [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}],[!\mathsf{true} \mid !\mathsf{false} \mid \ldots]) \mid \ldots] \\
(6\ 6') & \beta_5
\end{array}
$$

The second argument to `and` has a similar derivation:

$$
\begin{array}{rl}
6'' & A[d:\tau_5] \quad (T\ (d\ \texttt{false} f)) \\
7'' & A[d:\tau_5] \quad T \\
7'' & = \\
 & \tau_1 \\
6''' & A[d:\tau_5] \quad (d\ \texttt{false}) \\
7''' & A[d:\tau_5] \quad d \\
7''' & = \\
 & \tau_5 \\
7'''' & A[d:\tau_5] \quad \texttt{false} \\
7'''' & = \\
 & [!\mathsf{false} \mid \ldots] \\
7''', 7'''' & \text{Unify} \left\{ \begin{array}{c} \tau_5 \\ [!\mathsf{fn}([!\mathsf{false} \mid \ldots], \beta_6)] \end{array} \right. \\
 & = \\
 & \tau_5 \leftarrow [!\mathsf{fn}([!\mathsf{true} \mid !\mathsf{false} \mid \ldots], \beta_6)] \\
 & \beta_6 \leftarrow \beta_3 \\
6''' & = \\
 & \beta_6 \\
6'', 6''' & \text{Unify} \left\{ \begin{array}{c} \tau_1 \\ [!\mathsf{fn}(\beta_6, \beta_7)] \end{array} \right. \\
 & = \\
 & \beta_7 \leftarrow \beta_4 \\
6'' & = \\
 & \beta_7
\end{array}
$$

Again, note the updating of the type of the argument of d to include false as well as true.

With the second argument of the `and` construct, we conclude the derivation of the last `case` subexpression.

$$
\begin{array}{rl}
(6\ 6'), 6'' & \text{Unify}\left\{\begin{array}{c} \beta_5 \\ [!\mathsf{fn}(\beta_7, \beta_8)] \end{array}\right. \\
& = \\
& \text{Unify}\left\{\begin{array}{c} [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}], [!\mathsf{true} \mid !\mathsf{false} \mid \ldots]) \mid \ldots] \\ [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false}], \beta_8)] \end{array}\right. \\
& = \\
& \beta_8 \leftarrow [!\mathsf{true} \mid !\mathsf{false} \mid \ldots] \\
5 & = \\
& [!\mathsf{true} \mid !\mathsf{false} \mid \ldots] \\
4''' & = \\
& [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{true} \mid !\mathsf{false} \mid \ldots], \beta_6)], [!\mathsf{true} \mid !\mathsf{false} \mid \ldots]) \mid \ldots] \\
3'', 4''' & \text{Unify}\left\{\begin{array}{c} \left(\begin{array}{c} [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{fn}(\gamma, \delta) \mid \ldots], \\ \alpha)], [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false} \mid .\mathsf{fn}(\gamma, \delta)], \\ \alpha) \mid \ldots]) \mid \ldots] \end{array}\right) \\ \left(\begin{array}{c} [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{fn}([!\mathsf{true} \mid !\mathsf{false} \mid \ldots], \beta_6)], \\ [!\mathsf{true} \mid !\mathsf{false} \mid \ldots]) \mid \ldots], \beta_9)] \end{array}\right) \end{array}\right. \\
& = \\
& \gamma \leftarrow [!\mathsf{true} \mid !\mathsf{false} \mid \ldots] \\
& \delta \leftarrow \beta_6 \\
& \beta_9 \leftarrow [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false} \mid .\mathsf{fn}(\gamma, \delta)], \alpha) \mid \ldots] \\
(3\ 3'\ 3''\ 3''') & = \\
& \beta_9
\end{array}
$$

With the type assignment finished for the `case` application, we can proceed with the type assignment for the function definition:

$$
\begin{array}{rl}
3'''' & A \quad B \\
3'''' & = \\
 & \tau_2 \\
(3\ 3'\ 3''\ 3'''), 3'''' & \text{Unify} \left\{ \begin{array}{c} [!\mathsf{fn}([.\mathsf{true} \mid .\mathsf{false} \mid .\mathsf{fn}(\gamma, \delta)], \alpha) \mid \ldots] \\ [!\mathsf{fn}(\tau_2, \beta_{10})] \end{array} \right. \\
 & = \\
 & \tau_2 \leftarrow [.\mathsf{true} \mid .\mathsf{false} \mid .\mathsf{fn}(\gamma, \delta)] \\
 & \beta_{10} \leftarrow \alpha \\
2 & = \\
 & [!\mathsf{fn}(\tau_2, \alpha) \mid \ldots] \\
1' & = \\
 & [!\mathsf{fn}(\tau_1, [!\mathsf{fn}(\tau_2, \alpha) \mid \ldots]) \mid \ldots] \\
1, 1' & \text{Unify} \left\{ \begin{array}{c} [!\mathsf{fn}([!\mathsf{fn}\alpha', \alpha' \mid \ldots], \alpha') \mid \ldots] \\ [!\mathsf{fn}([!\mathsf{fn}(\tau_1, [!\mathsf{fn}(\tau_2, \alpha) \mid \ldots]) \mid \ldots], \beta_{11})] \end{array} \right. \\
 & = \\
 & \tau_1 \leftarrow \alpha' \\
 & \alpha' \leftarrow [!\mathsf{fn}(\tau_2, \alpha) \mid \ldots] \\
 & \beta_{11} \leftarrow \alpha' \\
0 & = \\
 & \alpha'
\end{array}
$$

The unification step for this last part of the derivation bears closer examination, as *circular* unification is truly required here:

Performing the substitutions for τ_1,τ_2, and α gives

$$\text{Unify}\left\{\begin{array}{c}[!\mathsf{fn}([!\mathsf{fn}(\alpha',\alpha')],\alpha')\mid\ldots]\\ [!\mathsf{fn}([!\mathsf{fn}(\tau_1,[!\mathsf{fn}(\tau_2,\alpha)\mid\ldots])\mid\ldots],\beta_{11})]\end{array}\right.$$

$$=$$

$$\text{Unify}\left\{\begin{array}{c}\tau_1\\ [!\mathsf{fn}(\tau_2,\alpha)\mid\ldots]\end{array}\right.$$

$$=$$

$$\text{Unify}\left\{\begin{array}{c}[!\mathsf{fn}(\beta_3,[.\mathsf{true}\mid.\mathsf{false}])]\\ [!\mathsf{fn}([.\mathsf{true}\mid.\mathsf{false}\mid.\mathsf{fn}(\gamma,\delta)],\alpha)\mid\ldots]\end{array}\right.$$

$$=$$

$$\text{Unify}\left\{\begin{array}{c}[!\mathsf{fn}(\beta_3,[.\mathsf{true}\mid.\mathsf{false}])]\\ \left(\begin{array}{c}[!\mathsf{fn}([.\mathsf{true}\mid.\mathsf{false}\mid.\mathsf{fn}([!\mathsf{true}\mid!\mathsf{false}\mid\ldots],\beta_3)],\\ [!\mathsf{true}\mid!\mathsf{false}\mid\ldots])\mid\ldots]\end{array}\right)\end{array}\right.$$

$$=$$

$$\beta_3\leftarrow[.\mathsf{true}\mid.\mathsf{false}\mid.\mathsf{fn}([!\mathsf{true}\mid!\mathsf{false}\mid\ldots],\beta_3)]$$
$$\alpha\leftarrow[!\mathsf{true}\mid!\mathsf{false}]$$

Note that the unification substitution for β_3 is circular. Rewriting the type assignment for `taut` in conventional notation, we see that

$$\texttt{taut}:\beta_3\rightarrow(\mathsf{true}+\mathsf{false})$$

where

$$\beta_3=(\mathsf{true}+\mathsf{false}+((\mathsf{true}+\mathsf{false})\rightarrow\beta_3))$$

In this case, β_3 can be seen as arbitrary arity boolean functions (curried).

3.6.1 Some Anomalies

According to theorem 3.8, any type assignment produced by the $\mathcal{R}$-term algorithm is sound. In a few cases, however, the algorithmic method produces a less informative type than could be derived by using the inference rules differently. The intuitive reason for such a difference lies in the nature of the solution method. Since the form of the slack variable equations is *limited*, some solutions to the original inequalities are inevitably lost for some cases

An example of such behavior is the `deep` function of example 2.8.

Example 3.12 *(The* **deep** *function)*

$$\begin{array}{l}
\mathbf{deep} = \\
\texttt{rec}\ (\lambda D.\lambda n. \\
\quad \texttt{case}_{\texttt{z,suc}} \\
\quad\quad (\lambda d.\texttt{cons}(\texttt{z})(\texttt{nil}) \\
\quad\quad (\lambda d.\texttt{cons}(D(\texttt{pred}(d)))(\texttt{nil})) \\
\quad\quad (n)
\end{array}$$

where `rec` is the least fixed point function. We will not subscript the `case` primitive here.

The initial environment A_0 is

$$\begin{array}{rcl}
\texttt{case} & : & [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{z} \mid \ldots], \alpha)], \\
& & \quad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{suc} \mid \ldots], \alpha)], \\
& & \quad\quad [!\mathsf{fn}([.\mathsf{z} \mid .\mathsf{suc}], \alpha) \mid \ldots]) \mid \ldots]) \mid \ldots] \\
\texttt{rec} & : & [!\mathsf{fn}([!\mathsf{fn}(\alpha, \alpha) \mid \ldots], \alpha) \mid \ldots] \\
\texttt{cons} & : & [!\mathsf{fn}(\alpha, [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha)], [!\mathsf{cons}(\alpha) \mid \ldots]) \mid \ldots]) \mid \ldots] \\
\texttt{pred} & : & [!\mathsf{fn}([!\mathsf{suc}], [!\mathsf{z} \mid !\mathsf{suc} \mid \ldots]) \mid \ldots] \\
\texttt{nil} & : & [!\mathsf{nil} \mid \ldots] \\
\texttt{z} & : & [!\mathsf{z} \mid \ldots]
\end{array}$$

All the α's are generic type variables

The initial steps of the algorithm assign unknown types to D, n initially

$$\begin{array}{lll}
0 & A_0 & (\texttt{rec}\ (\lambda D.\lambda n.\ldots) \\
1 & A_0 & \texttt{rec} \\
1 & = & \\
 & [!\mathsf{fn}([!\mathsf{fn}\alpha', \alpha' \mid \ldots], \alpha') \mid \ldots] & \\
1' & A_0 & (\lambda D.\lambda n.\ldots \\
1' & A_0[D : \tau_1] & \lambda n.\ldots \\
2 & A_0[D : \tau_1][n : \tau_2] & (\texttt{case}\ (\lambda d.\ldots
\end{array}$$

Using A to abbreviate $A_0[D : \tau_1][n : \tau_2]$, then

$$
\begin{array}{lll}
3 & A & (\texttt{case}\ (\lambda d.(\texttt{cons z nil}))\ldots) \\
4 & A & \texttt{case} \\
4 & = & \\
\end{array}
$$

$$
\begin{array}{l}
[!\mathsf{fn}([!\mathsf{fn}([!\mathsf{z} \mid \ldots], \alpha)], \\
\qquad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{suc} \mid \ldots], \alpha)], \\
\qquad\qquad [!\mathsf{fn}([.\mathsf{z} \mid .\mathsf{suc}], \alpha) \mid \ldots]) \mid \ldots]) \mid \ldots]
\end{array}
$$

Typing the first arm of the case expression (and resolving) yields

$$
\begin{array}{lll}
3' & A & (\lambda d.(\texttt{cons z nil}) \\
4' & A[d:\tau_3] & (\texttt{cons z nil}) \\
5 & A[d:\tau_3] & \texttt{cons} \\
5 & = & \\
 & \multicolumn{2}{l}{[!\mathsf{fn}(\alpha_1, [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha_1)], [!\mathsf{cons}(\alpha_1) \mid \ldots]) \mid \ldots]) \mid \ldots]} \\
5' & A & \texttt{z} \\
5' & = & \\
 & [!\mathsf{z} \mid \ldots] &
\end{array}
$$

$$
5, 5' \quad \text{Unify} \left\{
\begin{array}{l}
[!\mathsf{fn}(\alpha_1, [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha_1)], \\
\qquad [!\mathsf{cons}(\alpha_1) \mid \ldots]) \mid \ldots]) \mid \ldots] \\
\qquad [!\mathsf{fn}([!\mathsf{z} \mid \ldots], \beta_1)]
\end{array}
\right.
$$

$$
\begin{array}{ll}
 & = \\
 & \alpha_1 \leftarrow [!\mathsf{z} \mid \ldots] \\
 & \beta_1 \leftarrow [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha_1)], [!\mathsf{cons}(\alpha_1) \mid \ldots]) \mid \ldots] \\
5, 5' & = \\
 & [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}([!\mathsf{z} \mid \ldots])], [!\mathsf{cons}([!\mathsf{z} \mid \ldots]) \mid \ldots]) \mid \ldots] \\
5'' & A \qquad \texttt{nil} \\
5'' & = \\
 & [!\mathsf{nil} \mid \ldots]
\end{array}
$$

$$
5, 5', 5'' \quad \text{Unify} \left\{
\begin{array}{l}
[!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}([!\mathsf{z} \mid \ldots])], [!\mathsf{cons}([!\mathsf{z} \mid \ldots]) \mid \ldots]) \mid \ldots] \\
\qquad [!\mathsf{fn}([!\mathsf{nil} \mid \ldots], \beta_2)]
\end{array}
\right.
$$

$$
\begin{array}{ll}
4' & = \\
 & [!\mathsf{cons}([!\mathsf{z} \mid \ldots]) \mid \ldots] \\
3' & = \\
 & [!\mathsf{fn}(\tau_3, [!\mathsf{cons}([!\mathsf{z} \mid \ldots])) \mid \ldots] \mid \ldots]
\end{array}
$$

Before typing the second arm of the `case` statement, we must unify the current arm with the first argument to the `case` statement.

$$3', 4'' \quad \text{Unify}\left\{\begin{array}{l} [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{z} \mid \ldots], \alpha)], \\ \qquad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{suc} \mid \ldots], \alpha)], \\ \qquad\qquad [!\mathsf{fn}([.\mathsf{z} \mid .\mathsf{suc}], \alpha) \mid \ldots]) \mid \ldots]) \mid \ldots] \\ \quad [!\mathsf{fn}([!\mathsf{fn}(\tau_3, [!\mathsf{cons}([!\mathsf{z} \mid \ldots])) \mid \ldots] \mid \ldots], \beta_3)] \end{array}\right.$$

$$=$$

$$\tau_3 \leftarrow [!\mathsf{z} \mid \ldots]$$

$$\alpha \leftarrow [!\mathsf{cons}([!\mathsf{z} \mid \ldots]) \mid \ldots]$$

$$\beta_3 \leftarrow [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{suc} \mid \ldots], \alpha)], [!\mathsf{fn}([.\mathsf{z} \mid .\mathsf{suc}], \alpha) \mid \ldots]) \mid \ldots]$$

Typing the second arm of case now gives

$$\begin{array}{rll}
5'' & A & \ldots\ (\lambda d.(\texttt{cons}\ (D\ (\texttt{pred}\ d))\ \texttt{nil}) \\
6' & A[d:\tau_5] & (\texttt{cons}\ (D\ (\texttt{pred}\ d))\ \texttt{nil}) \\
7' & A[d:\tau_5] & \texttt{cons} \\
7' & = & \\
 & \multicolumn{2}{l}{[!\mathsf{fn}(\alpha_2, [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha_2)], [!\mathsf{cons}(\alpha_2) \mid \ldots]) \mid \ldots]) \mid \ldots]} \\
8 & A[d:\tau_5] & (D\ (\texttt{pred}\ d)) \\
9 & A[d:\tau_5] & D \\
 & = & \\
 & \tau_1 & \\
9' & A[d:\tau_5] & (\texttt{pred}\ d) \\
10 & A[d:\tau_5] & \texttt{pred} \\
 & = & \\
 & \multicolumn{2}{l}{[!\mathsf{fn}([!\mathsf{suc}], [!\mathsf{z}\ |!\mathsf{suc} \mid \ldots]) \mid \ldots]} \\
10' & A[d:\tau_5] & d \\
 & = & \\
 & \tau_5 &
\end{array}$$

$$10, 10' \quad \text{Unify}\left\{\begin{array}{l} [!\mathsf{fn}([!\mathsf{suc}], [!\mathsf{z}\ |!\mathsf{suc} \mid \ldots]) \mid \ldots] \\ \qquad\qquad [!\mathsf{fn}(\tau_5, \beta_4)] \end{array}\right.$$

$$=$$

$$\tau_5 \leftarrow [!\mathsf{suc}]$$

$$\beta_4 \leftarrow [!\mathsf{z}\ |!\mathsf{suc} \mid \ldots]$$

Continuing,

$$
\begin{array}{rl}
9' & = \\
 & \beta_4 \\
9, 9' & \text{Unify} \left\{ \begin{array}{c} \tau_1 \\ [!\mathsf{fn}(\beta_4, \beta_5)] \end{array} \right. \\
 & = \\
 & \tau_1 \leftarrow [!\mathsf{fn}(\beta_4, \beta_5)] \\
8 & = \\
 & \beta_5 \\
7', 8 & \text{Unify} \left\{ \begin{array}{c} [!\mathsf{fn}(\alpha_2, [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha_2)], \\ \qquad [!\mathsf{cons}(\alpha_2) \mid \ldots]) \mid \ldots]) \mid \ldots] \\ [!\mathsf{fn}(\beta_5, \beta_6)] \end{array} \right. \\
 & = \\
 & \alpha_2 \leftarrow \beta_5 \\
 & \beta_6 \leftarrow [!\mathsf{fn}([.\mathsf{nil} \mid .\mathsf{cons}(\alpha_2)], [!\mathsf{cons}(\alpha_2) \mid \ldots]) \mid \ldots] \\
7'' & A[d : \tau_5] \;\; \texttt{nil} \\
 & = \\
 & [!\mathsf{nil} \mid \ldots] \\
7', 7'', 8 & \text{Unify} \left\{ \begin{array}{c} \beta_6 \\ [!\mathsf{fn}([!\mathsf{nil} \mid \ldots], \beta_7)] \end{array} \right. \\
 & = \\
 & \beta_7 \leftarrow [!\mathsf{cons}(\alpha_2) \mid \ldots] \\
6' & = \\
 & \beta_7 \\
5'' & = \\
 & [!\mathsf{fn}(\tau_5, \beta_7) \mid \ldots]
\end{array}
$$

Unifying this result with the previous `case` arm gives

$$
\begin{array}{rl}
3, 4'', 5'' & \text{Unify} \left\{ \begin{array}{c} \beta_3 \\ [!\mathsf{fn}([!\mathsf{fn}(\tau_5, \beta_7) \mid \ldots], \beta_8)] \end{array} \right. \\
 & = \\
 & \tau_5 \leftarrow [!\mathsf{suc}] \\
 & \beta_7 \leftarrow \alpha \\
 & \beta_8 \leftarrow [!\mathsf{fn}([.\mathsf{z} \mid .\mathsf{suc}], \alpha) \mid \ldots]
\end{array}
$$

So, we can type the entire expression up to the `rec` operator:

$$
\begin{array}{rl}
3'' & A \quad n \\
 & = \\
 & \tau_2 \\
3, 4'', 5'', 3'' & \text{Unify} \begin{cases} \quad \beta_8 \\ [!\mathsf{fn}(\tau_2, \beta_9)] \end{cases} \\
 & = \\
 & \tau_2 \leftarrow [.\mathsf{z} \mid .\mathsf{suc}] \\
 & \beta_9 \leftarrow \alpha \\
3 & = \\
 & \alpha \\
2 & = \\
 & [!\mathsf{fn}(\tau_2, \alpha) \mid \ldots] \\
1' & = \\
 & [!\mathsf{fn}(\tau_1, [!\mathsf{fn}(\tau_2, \alpha) \mid \ldots]) \mid \ldots]
\end{array}
$$

To clarify what happens next, we rewrite $1'$ using substituted information to give

$$
\begin{array}{l}
1' \quad = \\
\quad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{z} \mid !\mathsf{suc} \mid \ldots], \\
\qquad \beta_5)], \\
\qquad\qquad [!\mathsf{fn}([.\mathsf{z} \mid .\mathsf{suc}], [!\mathsf{cons}(\beta_5) \mid \ldots]) \mid \ldots]) \mid \ldots]
\end{array}
$$

where $\beta_5 = [!\mathsf{z} \mid \ldots]$.

Finally, we are ready to complete the type assignment, encorporating the type for `rec`.

$$
1,1' \quad \text{Unify} \begin{cases} [!\mathsf{fn}([!\mathsf{fn}(\alpha',\alpha') \mid \ldots], \alpha') \mid \ldots] \\ \quad [!\mathsf{fn}([!\mathsf{fn}([!\mathsf{fn}([!\mathsf{z} \mid !\mathsf{suc} \mid \ldots], \beta_5)], \\ \qquad\qquad [!\mathsf{fn}([.\mathsf{z} \mid .\mathsf{suc}], \\ \qquad\qquad\qquad [!\mathsf{cons}(\beta_5) \mid \ldots]) \mid \ldots]) \mid \ldots], \beta_{10})] \end{cases}
$$

$$
\begin{aligned}
&= \\
&\alpha' \leftarrow [!\mathsf{fn}([!\mathsf{z} \mid !\mathsf{suc}], \texttt{fix}\ \beta_5.[!\mathsf{z} \mid !\mathsf{cons}(\beta_5) \mid \ldots])] \\
&\beta_{10} \leftarrow \alpha' \\
0 \quad &= \\
&[!\mathsf{fn}([!\mathsf{z} \mid !\mathsf{suc}], \texttt{fix}\ \beta_5.[!\mathsf{z} \mid !\mathsf{cons}(\beta_5) \mid \ldots])]
\end{aligned}
$$

The final type is valid for the function `deep`, but the output type is has a spurious z present. The extra component is introduced in the unification step $7', 8$.

This unification step is, in reality a reflection of the restriction of solutions to discriminative union types. Informally, we might proceed

$$
\begin{aligned}
D &: (\mathsf{z} + \mathsf{suc}) \to \beta \text{ for some } \beta \\
(\texttt{cons}\ (D\ \texttt{pred}\ldots) &: \mathsf{cons}(\beta) \\
(\texttt{case}\ (\ldots \mathsf{cons}(\mathsf{z})\ \ldots (D\ ) &: \mathsf{cons}(\mathsf{z}) + \mathsf{cons}(\beta)
\end{aligned}
$$

The expression $\mathsf{cons}(\mathsf{z})+\mathsf{cons}(\beta)$ is *not* discriminative. So, we must convert the output to $\mathsf{cons}(\gamma)$. This entails converting β to $\mathsf{z} + \Delta$. So that the type for the `case` branch becomes $\mathsf{cons}(\mathsf{z} + \Delta)$.

If we could retain non-discriminative types, then the final output

$$
\texttt{fix}\ \beta..(\mathsf{cons}(z) + \mathsf{cons}(\beta))
$$

would be derived, as expected.

Another form of anomalous is really an aspect of the inference system itself. In this case, the lack of a single "best" type manifests as an $\mathcal{R}$-term that can only be translated to a *set* of regular types. The example below is an adaptation of example 3.8.

Example 3.13 *(The* **twice** *Example)* Consider the application

$$(\texttt{twice}\ F)$$

where

$$\texttt{twice} \equiv \lambda f.\lambda x.(f\ (f\ x))$$

and

$$F : \mathsf{a} + \mathsf{b} \to \mathsf{a}$$

for some types a, b Simple tracing yields:

$$\texttt{twice} : (\alpha \to \alpha) \to \alpha \to \alpha$$

So, the type assumption A_0 for the application is

$$\begin{array}{rcl} \texttt{twice} & : & [!\mathsf{fn}([!\mathsf{fn}(\alpha,\alpha)],[!\mathsf{fn}(\alpha,\alpha) \mid \ldots]) \mid \ldots] \\ \texttt{F} & : & [!\mathsf{fn}([f_1.\mathsf{a} \mid f_2.\mathsf{b}],[!\mathsf{a} \mid \ldots]) \mid \ldots] \end{array}$$

Note that the flag variables for F are explicit. So, we trace

$$\begin{array}{rl}
0 & A_0 \quad (\texttt{twice}\ F) \\
1 & A_0 \quad \texttt{twice} \\
 & = \\
 & [!\mathsf{fn}([!\mathsf{fn}(\alpha,\alpha)],[!\mathsf{fn}(\alpha,\alpha) \mid \ldots]) \mid \ldots] \\
1' & A_0 \quad F \\
 & = \\
 & [!\mathsf{fn}([f_1.\mathsf{a} \mid f_2.\mathsf{b}],[!\mathsf{a} \mid \ldots]) \mid \ldots] \\
1,1' & \text{Unify} \left\{ \begin{array}{c} [!\mathsf{fn}([!\mathsf{fn}(\alpha,\alpha)],[!\mathsf{fn}(\alpha,\alpha) \mid \ldots]) \mid \ldots] \\ [!\mathsf{fn}([!\mathsf{fn}([f_1.\mathsf{a} \mid f_2.\mathsf{b}],[!\mathsf{a} \mid \ldots]) \mid \ldots],\beta)] \end{array} \right. \\
 & = \\
 & \alpha \leftarrow [!\mathsf{a} \mid f_2.\mathsf{b}] \\
 & \beta \leftarrow [!\mathsf{fn}(\alpha,\alpha) \mid \ldots] \\
0 & = \\
 & [!\mathsf{fn}(\alpha,\alpha) \mid \ldots]
\end{array}$$

The flag variable f_2, however, appears on both sides of the fn constructor in the final type. So, a unique discriminative union cannot be obtained.

Consequently, we instantiate f_2 as $-, !$ respectively giving the following set of discriminative types

$$(\texttt{twice}\ F) : \left\{ \begin{array}{l} \mathsf{a} \rightarrow \mathsf{a} \\ (\mathsf{a} + \mathsf{b}) \rightarrow (\mathsf{a} + \mathsf{b}) \end{array} \right\}$$

In essence, both derivations of example 3.8 are encoded in the type.

Chapter 4

Inserting Explicit Run Time Checks

As explained earlier, *no* type checker can pass all "good" programs, and exclude all bad ones. The discriminative polyregular type checker from chapter 3 can assign types to a broad class of programs. Nevertheless, some programs will not pass even that liberal type checker. For example:

Example 4.1 *(Function with no type)*

$$N_1 = \lambda\ f.\ \texttt{if}\ f(\texttt{true})\ \texttt{then}\ f(5) + f(7)\ \texttt{else}\ f(7)$$

Attempting to type N leads to incompatible types for the argument f. To be appropriate for the `if` test, $f : \texttt{true} \rightarrow \texttt{true} + \texttt{false}$. Similarly, the first alternative requires $f : \texttt{suc} \rightarrow \texttt{0} + \texttt{suc}$. There is no unifier for the two different types of f, so there is no type for N_1. The function N_1 is not badly defined, however, because $N_1(\lambda\ x.x)$ never goes wrong.

In addition, a type checker may successfully type an expression, but produce a restricted type, limiting possible uses of a function. Consider the following related example

Example 4.2 *(An anomaly)*

$$N_2 = \lambda\ f.\ \texttt{if}\ f(\texttt{true})\ \texttt{then}\ f(5)\ \texttt{else}\ f(7)$$

In this case, $N_2 : \mathsf{true} + \mathsf{suc} \rightarrow \mathsf{true} + \mathsf{false}$, but $N_2(\lambda\ x.x)$ is also well defined, but does not type check.

In both examples, the program is meaningful, but the type analysis is too coarse to assign an appropriate type. Consequently, the statically typed languages of chapter 2 will either reject or severely limit these example programs. An equivalent dynamically typed language, on the other hand, will certainly accept these programs, and allow them to be used in their full generality.

A soft type system must be equally as flexible. Furthermore, the soft type system must also produce a statically type correct program as its output, so that the program validation and compilation advantages of static typing may be retained. The soft type system *transforms* programs by inserting explicit run-time checks[16]. The system should, however, strive to minimize the number of run-time checks a program requires.

To illustrate the idea:

Example 4.3 *(Modified Example 4.1)* Let $\mathsf{c}_1 : \mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false} \rightarrow \mathsf{z} + \mathsf{suc}$, and let $\mathsf{c}_2 : \mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false} \rightarrow \mathsf{true} + \mathsf{false}$. Then consider program N_1'

$$N_1' = \lambda\ f.\ \texttt{if}\ \mathsf{c}_2(f(\texttt{true}))\ \texttt{then}\ \mathsf{c}_1(f(5)) + \mathsf{c}_1(f(7))\ \texttt{else}\ f(7)$$

The type for N_1' is now $(\mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false} \rightarrow \mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false}) \rightarrow \mathsf{z} + \mathsf{suc}$. Also, the application $N_1'(\lambda\ x.x)$ is now type correct.

Similarly:

Example 4.4 *(Modified example 4.2)* Let $\mathsf{c}_1, \mathsf{c}_2$ be as in example 4.3. Then the function N_2 can be modified to N_2' using only one change to indicate that N_2' is type correct.

$$N_2' = \lambda\ f.\ \texttt{if}\ \mathsf{c}_2(f(\texttt{true}))\ \texttt{then}\ f(5)\ \texttt{else}\ f(7)$$

Now, the type system infers the type for N_2' as:

$$(\mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false} \rightarrow \mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false}) \rightarrow \mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false}$$

So, $N_2'(\lambda\ x.x)$ now type checks.

We cannot compare the meanings of the original programs N_1, N_2 with their respective altered programs N_1', N_2' without a semantics for $\mathsf{c}_1, \mathsf{c}_2$. To that end, we define:

[16]In practice, static type systems must also retain some run-time checks. Normally, the errors produced by the run-time system for a statically typed language are not *called* type errors.

Definition 4.1 *(Semantics for* $\mathsf{c}_1, \mathsf{c}_2$ *)*

$$(\mathcal{E}[\![\mathsf{c}_1]\!]\rho)(v) \;=\; \begin{cases} v & v = \mathrm{In}_{\mathsf{0}}(v') \\ v & v = \mathrm{In}_{\mathsf{suc}}(v') \\ \mathbf{fault} & \text{otherwise} \end{cases}$$

$$(\mathcal{E}[\![\mathsf{c}_2]\!]\rho)(v) \;=\; \begin{cases} v & v = \mathrm{In}_{\mathsf{true}}(v') \\ v & v = \mathrm{In}_{\mathsf{false}}(v') \\ \mathbf{fault} & \text{otherwise} \end{cases}$$

Note the use of the **fault** element. This element is intended to signify that a run-time check has failed.

Definition 4.1 indicates that $\mathsf{c}_{1,2}$ verify their arguments are elements of some given set of summands of D. If so, the argument is returned, otherwise, **fault** is returned. Put another way, the $\mathsf{c}_{1,2}$ functions have function as run-time checks for their respective functions. Consequently, it is no surprise that:

$$\begin{aligned} \mathcal{E}[\![N_1]\!]\rho &= \mathcal{E}[\![N_1']\!]\rho \\ \mathcal{E}[\![N_2]\!]\rho &= \mathcal{E}[\![N_2']\!]\rho \end{aligned}$$

To summarize, for each example, we were able to insert explicit run-time checks that did not alter the semantics of the original programs, but the modified program could be assigned a type.

Ultimately, we seek a method for *automatically* inserting the necessary run-time checking functions with the same properties. That is, given any program P, if P does not type check, we seek a P' such that:

1. P' is obtained from P by inserting run-time checking functions
2. P' type checks
3. $\mathcal{E}[\![P']\!]\rho = \mathcal{E}[\![P]\!]\rho$, that is, the (dynamically typed) semantics of the programs are identical.

We divide the analysis of the insertion of run-time checks into four parts. First, section 4.1 defines the dynamically typed semantics, introduces the semantics of explicit run-time checking, and indicates an appropriate method for inserting run-time checks in programs without modifying their semantics. Second, section 4.2 analyzes the soundness of explicitly checked programs with respect to the type inference system. Third, section 4.3 provides a method for automating the method of section 4.1, additionally indicating that the derived program passes the type checker. Finally, section 4.4 discusses how the modified programs provide possible information for optimization and verification.

4.1 Dynamic Typing and Explicit Run-Time Checks

In order to discuss the meaning of dynamically typed Exp programs, and compare them to statically typed Exp programs, we must give the semantics for Exp programs when they are viewed as dynamically typed. The semantics for dynamically typed programs is nearly identical to the statically typed semantics, except that dynamically typed programs never produce **wrong** as a value. They produce **fault** instead.

Using the same conventions as in chapter 2, we define

Definition 4.2 *(Dynamically typed semantics)* Let $R = \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}$ be a set of constructors. The auxiliary function $\mathcal{D}_R$ verifies that its argument belongs to one of the summands specified by R:

$$\mathcal{D}_R(d) = \begin{cases} d & d = \mathrm{In}_{D_{\mathtt{r}_1}}(d') \\ \vdots & \\ d & d = \mathrm{In}_{D_{\mathtt{r}_n}}(d') \\ \mathbf{fault} & \text{otherwise} \end{cases}$$

This is generalized to tuples as

$$\mathcal{D}_{R_1,\ldots,R_k}(\langle d_1, \ldots, d_k \rangle) = \begin{cases} \langle d_1, \ldots, d_k \rangle & \text{for all } i, 1 \leq i \leq k \\ & \mathcal{D}_{R_i}(d_i) \neq \mathbf{wrong} \\ \mathbf{fault} & \text{otherwise} \end{cases}$$

We define one more auxiliary function C_f:

$$C_f(v) = \begin{cases} f & v = \mathrm{In}_{D \to D}(f) \\ \mathbf{fault} & \text{otherwise} \end{cases}$$

With these in mind, the dynamically typed semantics for the Exp core is:

$$[\![x]\!]\eta \;=\; \eta(x)$$

$$
\begin{array}{rcl}
[\![\lambda x.e]\!]\eta & = & \mathrm{In}_{D\to D}(f) \\
 & & \quad \text{such that } f(v) = [\![e]\!]\eta[x=v] \\
[\![(e_1\ e_2)]\!]\eta & = & \begin{cases} v_1 & v_1 \in \{\mathbf{wrong}, \mathbf{fault}\} \\ v_1 & v_2 \in \{\mathbf{wrong}, \mathbf{fault}\} \\ v_1(v_2) & \text{otherwise} \end{cases} \\
 & & \text{where } v_1 = C_f([\![e_1]\!]\eta), v_2 = [\![e_2]\!]\eta \\
[\![\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2]\!]\eta & = & [\![e_2]\!]\eta[x = v_1],\ \text{where } v_1 = [\![e_1]\!]\eta
\end{array}
$$

The semantics for constructors is identical to the statically typed semantics:

$$\mathcal{E}[\![\mathtt{c}]\!]\eta = \begin{cases} \mathrm{In}_{D_{\mathtt{C}}}(*) & n = 0 \\ \mathrm{In}_{D\to D}(K) & n \neq 0 \end{cases}$$

where

$$K = \lambda d_1 \ldots d_n.\mathrm{In}_{D_{\mathtt{C}}}(\mathcal{D}_{R_1,\ldots,R_n}(\langle d_1, \ldots, d_n\rangle))$$

The semantics for selectors and `case` functions are analogous

$$\mathcal{E}[\![s^i_c]\!]\eta = \mathrm{In}_{D\to D}(S)$$

where

$$S(v) = \begin{cases} p_i(\mathrm{Out}_{D_c}(v)) & \mathcal{D}_C(v) \notin \{\mathbf{fault}, \mathbf{wrong}\} \\ v & \text{otherwise} \end{cases}$$

and

$$\mathcal{E}[\![\mathtt{case}_{\langle \mathtt{c}_1,\ldots,\mathtt{c}_k\rangle}]\!]\eta = \mathrm{In}_{D\to D}(C)$$

where

$$C = \lambda d f_1 \ldots f_k. \begin{cases} f_1(d) & d = \mathrm{In}_{c_1}(d') \\ & \vdots \\ f_k(d) & d = \mathrm{In}_{c_k}(d') \\ \mathbf{fault} & d \neq \mathbf{wrong} \\ \mathbf{wrong} & \text{otherwise} \end{cases}$$

To formally account for explicit run-time checks, we must add some functional constants be added to the programming language. These additional constants represent the functions that perform the explicit checks. Intuitively, a run-time check may only inspect the "tag" of a value, but nothing more. That is, a run-time check may verify that a certain object is a *function*, but it may *not* check that the object is a function from $\mathsf{int} \to \mathsf{int}$. For the functional languages under consideration, a "tag" is represented by a type constructor. A run-time check, then, verifies that a value has an appropriate tag. In other words, a run-time check *confirms* that a value is an acceptable element for the operation in question. In addition, the type checker typically infers a type for the checked expression that restricts the set of possible inputs to the run-time checking function. For example, a value v may be know to be either an int or bool, and the run-time check merely confirms that v is indeed of type int. Seen in this, way, a run-time check *narrows* the choices for values. For these reasons, we will refer to run-time checks as either a *confirmations* or *narrowers*.

Definition 4.3 *(Syntax for run-time checks)* Let C be the set of type constructors for the language. Then for every $T \subset S \subseteq C$ define an S, T *narrower* constant ${}_S c_T$. Let the set of all such narrowers for a language be N.

The notation for narrowers may use $+$ instead of the set brackets $\{\cdot\}$. For example:

Example 4.5 *(Some typical confirmation constants)* Let $C = \{\rightarrow, t_1, t_2\}$. Then ${}_{(\rightarrow + t_1)}c_{t_1}$ is a $\{\rightarrow, t_1\}, \{t_1\}$-narrower. Likewise, ${}_{t_1+t_2+\rightarrow}c_{t_1+t_2}$ is a $\{t_1 + t_2 + \rightarrow\}, \{t_1 + t_2\}$-narrower.

Having added some new constants to the programming language, we must now define the semantics for these constants. The semantics of a narrower are best understood by observing that a "tag" from a *semantic* viewpoint identifies a given summand of the data domain D. Recall that for each type constructor $t \in C$, there is a corresponding summand of $D = D_{t_1} \oplus \ldots D_t \oplus \ldots$ (see section 2.3.1)

Definition 4.4 *(Semantics of narrowers)* Let c be an S, T-narrower. Identify summands of D by subscript, such as D_{t_1} indicates the summand corresponding to the constructor t_1. Then define $\mathcal{E}[\![{}_S c_T]\!]\rho$ by

$$\mathcal{E}[\![{}_S c_T]\!]\rho(v) = \begin{cases} \perp & v = \perp \\ v & \text{if there is some } t_i \in T, \\ & \text{and some } v' \in D \\ & \text{s.t. } v = \mathrm{In}_{D_{t_i}} v' \\ \textbf{fault} & \text{Otherwise} \end{cases}$$

The types assigned to narrowers must also reflect "sufficient generality". Intuitively, functions that inspect only a "tag" should not impose any other restrictions on the value. Consequently, a "tag" type should be "as polymorphic as possible", and narrower's types should be expressed using these general types.

Definition 4.5 *(Tag types)* Let c be any type constructor of arity n. Let $t_1, \ldots, t_n$ be n distinct type variables. Then $c(t_1, \ldots, t_n)$ is the tag type for constructor c. Similarly, if $S = \{c_1, \ldots, c_k\}$ is a set of type constructors, then the tag type for $S = c_1(t_1, \ldots, t_n) + c_2(t_{(n+1)}, \ldots,)$. In other words, the tag type for a set of constructors is the union type of all the individual tag types. The tag type of a set S is denoted $\mathrm{Tagt}(S)$

The type for a S, T-narrower, then is defined in

Definition 4.6 *(Types for narrowers)* If ${}_Sc_T$ is a narrower, then

$$ {}_Sc_T : \forall t_i . \mathrm{Tagt}(S) \rightarrow \mathrm{Tagt}(T) $$

where the quantification is over all type variables in $\mathrm{Tagt}(S), \mathrm{Tagt}(T)$.

The act of explicitly run-time checking a program using narrowers is a form of program transformation. That is, some original program, written without explicit checks, is syntactically transformed into a program that may have the explicit checks "inserted". Furthermore, the derived statically typed program should have the same meaning as the dynamically typed source program. This property can be realized by mimicing the action of implicit run-time checks. Run-time errors in dynamically typed programs have two distinct sources:

1. Primitive operations: data constructors, data selectors, and `case` statements

2. expressions of the form $(f x)$, where f is *not* a functional value

Each different source of run-time error gives rise to a different kind of program transformation. We consider each source in turn.

First, we analyze the transformation strategy for primitive operations. The primitive functions functions of the programming language are:

1. Value Constructors

2. Data Selectors

3. `case` Functions

For each different kind of primitive, we define an *explicitly-checked analog* for the primitive.

Definition 4.7 *(Explicitly checked analogs of primitive functions)*

- **Value constructors:** Let b be a constructor, with Arity(b) > 0, and signature:

  $$\texttt{constructor}\ b(T_1, T_2, \ldots, T_n)$$

  where $T_j = d_{j_1} + \ldots d_{j_k}$. According to chapter 2, each d_{j_l} in a T_j expression is a constructor name. Furthermore, each constructor name has a corresponding type constructor name. Hence, each T_j has an associated set of type constructor names. Let T_j synonymously stand for the associated set of type constructors. Then any b' where

  $$b' \equiv \lambda x_1 \ldots x_n.(b\ e_1\ \ldots\ e_n)$$

  and

  $$\begin{array}{lcl} e_j & \equiv & x_j \\ & \text{OR} & \\ e_j & \equiv & (sc_{T_j} x_j) \text{ for some narrower } sc_{T_j} \end{array}$$

  is an explicitly checked analog of b.

  Note that if $T_j = C$, where C is the entire set of type constructors, there is *no* narrowing constant of the form sc_C, so in these cases, $e_j \equiv x_j$.

- **Data selectors:** Let s be a selector for value constructor b. Then any s' where

  $$s' \equiv \lambda x.(s\ e)$$

  and

  $$\begin{array}{lcl} e & \equiv & x \\ & \text{OR} & \\ e & \equiv & (sc_{\{\mathsf{b}\}} x) \text{ for some narrower } sc_{\{\mathsf{b}\}} \end{array}$$

  is an explicitly checked analog of s

- $\texttt{case}$ **functions:** According to chapter 2, each case function is indexed by a set of type constructors. Consequently, for any $\texttt{case}_T$, T a set of type constructors, and $|T| = k$, then any $\texttt{case}'_T$ where

$$\texttt{case}'_T \equiv \lambda s x_1 \ldots x_k.(\texttt{case}_T\ e\ x_1\ \ldots\ x_n)$$

and

$$\begin{array}{rcl} e & \equiv & x \\ & \text{OR} & \\ e & \equiv & ({}_S c_T x) \text{ for some narrower } {}_S c_{\mathsf{T}} \end{array}$$

We often abbreviate the phrase f' is an "explicitly checked analog" of f by

$$f' \ \text{ e.c.a.p. } \ f$$

Using the concept of an explicitly checked analog, we define one kind of program transformation.

Definition 4.8 *(Ecap replacement)* Let P be a program expression, and let $\texttt{b}$ be a subexpression of P at location l, where $\texttt{b}$ is a constant of the first order specification. Let b' e.c.a.p. b. Then P' is an *e.c.a.p. replacement* of P using b at l iff $P = P'$, or P' is identical to P, except that the subexpression b at location l is replaced by b'.

The second source of run-time errors, expressions of the form (fx), require a slightly different transformation strategy. This strategy depends on a a certain subset of narrowers, the $\texttt{IsF}$ narrowers.

Definition 4.9 *(The* $\texttt{IsF}$ *narrower)* Any narrower ${}_S c_{\{\rightarrow\}}$ is an $\texttt{IsF}_S$ narrower.

An *explicitly checked application analog* is defined by:

Definition 4.10 *(Explicitly checked application analog)* Let $(f\ a)$ be an application expression. Then an explicitly checked application is any expression of the form $((\texttt{IsF}_S\ f)\ a)$ We indicate this relationship as

$$((\texttt{IsF}_S\ f)\ a) \text{ e.c.a.a. } (f\ a)$$

Replacing a applicative subexpression with its explicitly checked analog defines ecaa replacement.

Definition 4.11 *(Ecaa replacement)* Let P be a program expression, and let $(f\ a)$ be the subexpression of P at location l. Let

$$((\texttt{IsF}_S\ f)\ a) \text{ e.c.a.a. } (f\ a)$$

Then P' is an ecaa-replacement for P iff $P = P'$ or P' is identical to P, except that the subexpression at location l is $((\texttt{IsF}_S\ f)\ a)$.

The complete program transformation strategy, then, is finite application of either ecap-replacement, or ecaa-replacement. A program P' is said to be an *explicitly-checked replacement* of P when:

Definition 4.12 *(Ec-replacement)* Let P be a program expression of Exp. Then P' is an explicitly-checked replacement for P iff there is a finite sequence

$$P_0 = P, P_1, \ldots, P_m = P'$$

where P_{i+1} is either an ecaa replacement or an ecap replacement of P_i.

The most important property of Ec-replacement is that it preserves the (dynamically typed) semantics of programs. We will show that if P' is an ec-replacement of P then $\mathcal{E}[\![P']\!]\rho = \mathcal{E}[\![P]\!]\rho$. Intuitively, the proof of this fact relies on the observation that ecap replacement preserves the semantics of constants, and ecaa replacement preserves the semantics of application. These intuitions are formalized by the following lemmas.

Lemma 4.1 *(Ecap preserves constant semantics)* Suppose b' e.c.a.p. b. Then for any environment ρ,

$$\mathcal{E}[\![b']\!]\rho = \mathcal{E}[\![b]\!]\rho$$

Proof Consider the case where $\texttt{b}$ is a constructor. Let $\mathbf{b} = \mathcal{E}[\![\texttt{b}]\!]\rho$. Let

$$b' = \lambda x_1 \ldots x_n.(\texttt{b}\ e_1\ \ldots\ e_n)$$

Then

$$((\mathcal{E}[\![b']\!]\rho)\ v_1\ \ldots\ v_n) = (\mathbf{b}\ \mathcal{E}[\![e_1]\!]\rho'\ \ldots\ \mathcal{E}[\![e_n]\!]\rho')$$

where $\rho' = \rho[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]$. According to definition 4.7, $e_i = x_i$,, or $e_i = ({}_S c_{T_i}\ x_i)$. If $e_i = x_i$, then $\mathcal{E}[\![e_i]\!]\rho' = v_i$. If $e_i = ({}_S c_{T_i}\ x_i)$, then $\mathcal{E}[\![e_i]\!]\rho' =_S \mathbf{c}_{T_i}(v_i)$. There are two cases to consider. Suppose first that all $v_i \in T_i$. Then ${}_S\mathbf{c}_{T_i}(v_i) = v_i$, by definition 4.4. Consequently, $\mathcal{E}[\![e_i]\!]\rho' = v_i$, and

$$((\mathcal{E}[\![b']\!]\rho)\ v_1\ \ldots\ v_n) = (\mathbf{b}\ v_1 \ldots\ v_n)$$

Now suppose that some $v_k \notin T_k$. Then

$$(\mathbf{b}\ v_1\ \ldots\ v_n) = \mathbf{fault}$$

If $e_k = x_k$, then $\mathcal{E}[\![e_k]\!]\rho' = v_k$, and

$$(\mathbf{b}\ \mathcal{E}[\![e_1]\!]\rho'\ \ldots\ v_k \ldots\ \mathcal{E}[\![e_n]\!]\rho') = \mathbf{fault}$$

If $e_k = ({}_S c_{T_i}\ x_k)$ then $\mathcal{E}[\![e_k]\!]\rho' = \mathbf{wrong}$, so

$$(\mathbf{b}\ \mathcal{E}[\![e_1]\!]\rho'\ \ldots\ \mathbf{fault} \ldots\ \mathcal{E}[\![e_n]\!]\rho') = \mathbf{fault}$$

For either choice of e_k

$$((\mathcal{E}[\![b']\!]\rho)\ v_1\ \ldots\ v_n) = \mathbf{fault} = (\mathbf{b}\ v_1 \ldots\ v_n)$$

and the lemma is proved.

The proof for selectors and `case` are similar. □

Likewise,

Lemma 4.2 *(Ecaa preserves application semantics)* Let

$$((\mathtt{IsF}_S\ f)\ a)\ \text{e.c.a.a.}\ (f\ a)$$

Then for any environment ρ,

$$\mathcal{E}[\![((\mathtt{IsF}_S\ f)\ a)]\!]\rho = \mathcal{E}[\![(f\ a)]\!]\rho$$

Proof By definition 4.2

$$\mathcal{E}[\![(f\ a)]\!]\rho = (\mathcal{E}[\![f]\!]\rho\ \mathcal{E}[\![a]\!]\rho)$$

and

$$\mathcal{E}[\![((\mathtt{IsF}_S\ f)\ a)]\!]\rho = ((\mathbf{IsF}_S\ \mathcal{E}[\![f]\!]\rho)\ \mathcal{E}[\![a]\!]\rho)$$

If $\mathcal{E}[\![f]\!]\rho \in D \to D$, then

$$(\mathbf{IsF}_S\ \mathcal{E}[\![f]\!]\rho) = \mathcal{E}[\![f]\!]\rho$$

And so

$$((\mathbf{IsF}_S\ \mathcal{E}[\![f]\!]\rho)\ \mathcal{E}[\![a]\!]\rho) = (\mathcal{E}[\![f]\!]\rho\ \mathcal{E}[\![a]\!]\rho)$$

If $\mathcal{E}[\![f]\!]\rho \notin D \to D$, then

$$(\mathcal{E}[\![f]\!]\rho\ \mathcal{E}[\![a]\!]\rho) = \mathbf{fault}.$$

Also, in this case

$$(\mathbf{IsF}_S\ \mathcal{E}[\![f]\!]\rho) = \mathbf{fault}$$

So

$$((\mathbf{IsF}_S\ \mathcal{E}[\![f]\!]\rho)\ \mathcal{E}[\![a]\!]\rho) = (\mathbf{fault}\ \mathcal{E}[\![a]\!]\rho) = \mathbf{fault}$$

In all cases

$$\mathcal{E}[\![(f\ a)]\!]\rho = (\mathcal{E}[\![f]\!]\rho\ \mathcal{E}[\![a]\!]\rho)$$

which shows the lemma □

These lemmas enable us to show

Theorem 4.1 *(Ec-replacement preserves semantics)* Let P be a program expression. Let P' be derived from P by either a single ecap-replacement step, or a single ecaa-replacement step. Then $\mathcal{E}[\![P]\!]\rho = \mathcal{E}[\![P']\!]\rho$

Proof The strategy is structural induction.

Case P = x There are no ecaa-replacements or ecap replacements possible for this case, so the theorem is trivially true.

Case P = c If c is first order constant, then only ecap may be valid. If Arity(c) = 0, then no replacement is possible, so the theorem is trivially true. If Arity(c) > 0, then $P' = c'$, where c' e.c.a.p. c is a possible ec-replacement. By lemma 4.1, $\mathcal{E}[\![P']\!]\rho = \mathcal{E}[\![P]\!]\rho$.

Case $\mathbf{P} = \lambda\mathbf{x.e}$ In this case, any replacement step must apply to the subexpression e. Let e' be the resulting replacement program. Let v be any value, then by the induction hypothesis

$$\mathcal{E}[\![e']\!]\rho[x \leftarrow v] = \mathcal{E}[\![e]\!]\rho x \leftarrow v$$

By semantics for λ expressions, this means

$$(\mathcal{E}[\![\lambda x.e']\!]\rho\ v) = (\mathcal{E}[\![\lambda x.e]\!]\rho\ v)$$

Since v is arbitrary, we conclude

$$\mathcal{E}[\![\lambda x.e']\!]\rho = \mathcal{E}[\![\lambda x.e]\!]\rho$$

Case $\mathbf{P} = (\mathbf{f}\ \mathbf{a})$ For this case, there are three possibilities:

1. $P' = (f'\ a)$
2. $P' = (f\ a')$
3. $P' = ((\mathtt{IsF}_S\ f)\ a)$

Considering each case in turn, if $P = (f'\ a)$, then by inductive hypothesis

$$\mathcal{E}[\![f']\!]\rho = \mathcal{E}[\![f]\!]\rho$$

So,

$$(\mathcal{E}[\![(f']\!]\rho\ \mathcal{E}[\![a]\!]\rho) = (\mathcal{E}[\![f]\!]\rho\ \mathcal{E}[\![a]\!]\rho)$$

Similarly, if $P' = (f\ a')$, then the inductive hypothesis again gives

$$\mathcal{E}[\![a']\!]\rho = \mathcal{E}[\![a]\!]\rho$$

So,

$$(\mathcal{E}[\![(f]\!]\rho\ \mathcal{E}[\![a']\!]\rho) = (\mathcal{E}[\![f]\!]\rho\ \mathcal{E}[\![a]\!]\rho)$$

In the final case, P' e.c.a.a. P, so lemma 4.2 shows the theorem for this final subcase.

Case $\mathbf{P} = \mathtt{let}\ \mathbf{x} = \mathbf{e_1}\ \mathtt{in}\ \mathbf{e_2}$ For this case $P' = \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2$, or $P' = \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2'$. Supposing $P' = \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2$, by the induction hypothesis,

$$\mathcal{E}[\![e_1]\!]\rho = \mathcal{E}[\![e_1']\!]\rho$$

Consequently,

$$\rho[x \leftarrow \mathcal{E}[\![e_1]\!]\rho] = \rho[x \leftarrow \mathcal{E}[\![e_1']\!]\rho]$$

So,

$$\mathcal{E}[\![e_2]\!]\rho[x \leftarrow \mathcal{E}[\![e_1]\!]\rho] = \mathcal{E}[\![e_2]\!]\rho[x \leftarrow \mathcal{E}[\![e_1']\!]\rho]$$

which means

$$\mathcal{E}[\![\texttt{let}\ x = e_1'\ \texttt{in}\ e_2]\!]\rho = \mathcal{E}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]\rho$$

The other case is similarly straightforward.

□

An obvious corollary gives the desired semantic preservation property.

Corollary 4.1 Let P' be an ec-replacement of P. Then $\mathcal{E}[\![P]\!]\rho = \mathcal{E}[\![P']\!]\rho$.

Proof By induction on the length of the replacement sequence $P_0, \ldots, P_n$.

Case $\mathbf{n = 0}$. In this case, $P \equiv P'$, so the theorem is trivially true.

Case $\mathbf{n = k + 1}$.

By inductive hypothesis,

$$\mathcal{E}[\![P]\!]\rho = \mathcal{E}[\![P_k]\!]\rho$$

We know P_{k+1} is derived from P_k by a single ecaa-replacement, or a single ecap-replacement. So, by theorem 4.1

$$\mathcal{E}[\![P_k]\!]\rho = \mathcal{E}[\![P_{k+1}]\!]\rho$$

Consequently,

$$\mathcal{E}[\![P]\!]\rho = \mathcal{E}[\![P_{k+1}]\!]\rho$$

□

4.2 Soundness of type inference for explicitly checked programs

The type language analysis of chapter 2 dealt with statically typed constants that did not return the **fault** value. Consequently, to insure that our type inference and type assignment methods are still valid requires reworking the soundness theorem. To do so, we must alter the semantics of types slightly. The semantics for types detailed in definition 2.26 did not take into account the **fault** value. For type inference in the presence of run-time errors to be sound, **fault** must be a member of every type. We refer to the extended type semantics by $\mathcal{T}^e[\![\cdot]\!]$ The definition for $\mathcal{T}^e[\![\cdot]\!]$ follows the notational conventions of definition 2.26

Definition 4.13 *(Extended type semantics)*

Let ν : TyVars $\rightarrow \mathcal{I}$ be a valuation for free type variables. Denote by $\mathcal{I}(S)(J)$ the ideal $\mathrm{In}_S(J)$, where J is an ideal of S, and S is a summand of D. Additionally, $\mathcal{I}(S) \equiv \mathcal{I}(S)(\mathrm{Triv})$. Let μ stand for the fixed point operator over the ideals using the MPS-metric. Let $F = \{\textbf{fault}\}$ Then define $\mathcal{T}^e[\![\cdot]\!]$ as:

$$\begin{array}{rcl}
\mathcal{T}^e[\![c]\!]\nu & = & \mathcal{I}(D_{\mathbf{c}}) \cup F \\
 & & \text{for 0-ary functions (constants)} \\
\mathcal{T}^e[\![c(t_1,\ldots,t_n)]\!]\nu & = & \mathcal{I}(D_{\mathbf{c}})\langle \mathcal{T}^e[\![t_1]\!]\nu,\ldots,\mathcal{T}^e[\![t_n]\!]\nu)\rangle \cup F \\
 & & \text{for } c \neq\rightarrow \\
\mathcal{T}^e[\![t_1 \rightarrow t_2]\!]\nu & = & \mathcal{I}(D \rightarrow D)(\mathcal{T}^e[\![t_1]\!]\nu \rightarrow \mathcal{T}^e[\![t_2]\!]\nu) \cup F \\
 & & \text{NOTE: } \rightarrow \text{ on the rhs is} \\
 & & \text{the ideal function constructor} \\
\mathcal{T}^e[\![t_1 + t_2]\!]\nu & = & \mathcal{T}^e[\![t_1]\!]\nu \cup \mathcal{T}^e[\![t_2]\!]\nu \\
\mathcal{T}^e[\![\mathtt{fix}\ \alpha.t]\!]\nu & = & \mu\ i.f(i) \cup F \\
 & & \text{where } f(i) = \mathcal{T}^e[\![t]\!]\nu[\alpha = i]
\end{array}$$

Using the extended type semantics, we must re-verify the soundness of the subtype inference system — both the algebraic system and the inference system. As in chapter 2, the actual details of these proofs are all presented in separate section, namely section 4.5.

Theorem 4.2 *(Soundness of Algebraic Subtype Inference)* For any first order regular type expression T, $\mathcal{T}^e[\![\Sigma(T)]\!]\nu = \mathcal{T}^e[\![T]\!]\nu$.

Similarly, the soundness of the type judgment rules of definition 2.19 is established by

Theorem 4.3 *(Extended Soundness for Inference Rules)* The inference rules in definition 2.19 are sound with respect to the extended semantics.

Continuing our analysis, we show that the type inference system for program expressions is sound with respect to the extended semantics.

Theorem 4.4 *(Extended soundness of type inference)*

Let $A \vdash e : \sigma$ be any inference from definition 2.30. Let ρ be an environment respecting A, and let ν be any valuation. Then $\mathcal{E}[\![e]\!]\rho : \mathcal{T}^e[\![\sigma]\!]\nu$

The extended semantics for types, then, respects all of the inference processes established in chapter 2. The interpretation of "well-typed", however, is different from its chapter 2 counterpart. The principle

Well typed programs do not go wrong

must be replaced by

Well typed explicitly checked programs do not go wrong, but they may signal a run-time error.

4.3 Automating the Insertion Process

Using the methods previously established, a programmer may modify a faulty program by applying some number of ecaa or ecap replacement steps to produce a new program. Theorem 4.1 assures that the new program is semantically identical to his original program. If, in addition, the new program type checks then theorem 4.4 provides the further assurance that the output of the program either has the indicated type, or some run-time check is activated. Consequently, *if* the programmer can find some ec-replacement for his original program that also type checks, then he is assured that the replacement program performs the same computation as his original program, and is completely safe. The scenario proposed above, however suffers from two defects:

1. No assurance has been given that some ec-replacement program that type checks can always be found.

2. The programmer must manually insert the narrowing functions (perhaps making several tries), violating the minimal text principle for soft typing.

The remedy for these defects comes in the form of an automated transformation of a program to an ec-replacement program that type-checks. Since the method always succeeds, it resolves both objections.

Intuitively, the method can be thought of as enumerating all possible ec- replacement programs, and performing automatic type inference on each one until some solution is found. Again, as in chapter 3, we use a Rémy encoding instead of actual enumeration to solve the problem. An entire set of narrowers can be represented as a single Rémy term thusly:

Example 4.6 *(Narrowers via a Rémy encoding)* Let the set of type constructors be $\{a(t), b, c\}$. Then consider the set of narrowers $_S n_{\{c\}}$. By enumeration, this set is

$$_{a(t)+c}n_c$$

$$_{a(t)+b+c}n_c$$

$$_{b+c}n_c$$

This set of narrowers can be tabularized as

a	a argument	b	c	narrower
!	t	$-$	!	$_{a(t)+c}n_c$
!	t	!	!	$_{a(t)+b+c}n_c$
$-$	t	!	!	$_{b+c}n_c$

And finally, using the Rémy notion of summarizing via variables, the above table becomes

$$\mathcal{R}(v_1, t, v_2, !)^n \mathcal{R}(-, t', -, !)$$

Observe the lack of $-$ flags in the argument positions for the set of narrowers.

The transformation that "removes" negative information from the argument types of primitive functions gives a compact representation of all possible ec analogs for the primitive. These ideas are formalizes in the following treatment.

Definition 4.14 *(Ec general type)* Let $S = \{c_i\}$ be a set of type constructors. Define the *ec general type* of S, notated $\mathrm{ECG}(S)$ as

$$\mathcal{R}(\ldots, \mathit{fl}_i, a_i^1, a_i^2, \ldots)$$

where

$$\mathit{fl}_i = \begin{cases} ! & c_i \in S \\ f_i & \text{otherwise} \end{cases}$$

where each f_i is a distinct variable. Each a_i^j is distinct as well. Additionally, define

$$\mathrm{ECG}(\tau) = \tau$$

for any type variable τ

The expressive ability of this type definition can be seen in

Lemma 4.3 *(Ec expression correspondence)* Let T be either a set of constructors, or a type variable. Then there is a $1-1$ correspondence between flag substitution instances of $\mathrm{ECG}(T)$ and all valid e_k expressions (definition 4.7) for $x : T$.

Proof First, suppose T is a type variable. In this case, there is only one flag substitution, namely the identity. According to chapter 2, this means that $x \in C$, where C is the entire set of constructors. By definition 4.3, there is no narrower ${}_S n_C$, since $C \supseteq S$ for any S. Consequently, there is only one valid e_k expression, namely x itself. So, associate Id and x. If $T = C$, the same argument holds.

Consequently, suppose $T \subset C$. Then let V be the flag substitution

$$V(f_i) = \begin{cases} - & c_i \notin T \\ ! & \text{otherwise} \end{cases}$$

Then let V correspond to the e_k expression x. For any other flag defining substitution V', $V'(f_k) =!$, for some k s.t. $c_k \notin T$. In this case, definition 4.14 yields

$$\mathrm{ECG}(T \cup \{c_k \,|\, V'(f_k) =!\}) = V'(\mathrm{ECG}(T))$$

Letting $S = T \cup \{c_k \,|\, V'(f_k) =!\}$, the substitution V' corresponds to

$$({}_S n_T \ x)$$

$\square$

As a corollary, we note that `IsF` narrowers are a special case of narrowers in general so we have

Corollary 4.2 *(Application replacement corresponds)* There is a $1-1$ correspondence between valid ecaa replacements for $(f \ a)$ and flag substitutions of ECG($\rightarrow$).

Proof `IsF` is a narrower with $T = \{\rightarrow\}$. So, by lemma 4.3, any valid $((\texttt{IsF}_S \ f) \ a)$ corresponds to some flag defining substitution $\square$

One other useful corollary is an observation about the proof. Note that the $e_k \equiv x$ possibility occurs exactly for the flag substitution produces $-$ for any constructor not in T.

Corollary 4.3 *(Id substitution correspondence)* If $V(f_i) = -$, for $c_i \notin T$,then $e_k \equiv x$ is the corresponding expression.

Proof By construction in the proof of lemma 4.3 $\square$

The set of $\mathcal{R}$ terms derived in accordance with definition 4.14 are terms without any occurrences of $-$ flags. In particular, this set of $\mathcal{R}$-terms is a subset of $\Sigma(\{\mathcal{R}, !\}, X)$. These !-only terms have an important property: For any set of these terms, there is always a circular unifier ! The use of circularity is indispensable, as the proof of the following lemma indicates:

Lemma 4.4 *(!-only $\mathcal{R}$-terms circular unify)* Let

$$\{(L_1, R_1), (L_2, R_2), \ldots, (L_m, R_m)\}$$

be a set of !-only $\mathcal{R}$-term pairs. Then there exists a substitution S such that

$$S(L_1) = S(R_1), S(L_2) = S(R_2), \ldots, S(L_m) = S(R_m)$$

That is, for any set of !-only $\mathcal{R}$-term pairs, there exists a unifier.

Proof By induction on the maximum number of $\mathcal{R}$ symbols in the L_k terms.

Case max $=$ **0** For this case, the set of terms must be

$$(x_1, R_1), (x_2, R_2), \ldots, (x_m, R_m)$$

where the x_i's are variables. Since we are using *circular* unification, the substitution

$$S(x_i) = R_i$$

produces the desired solution. Circular unification is critical. Without it, an occur check may prevent solutions

Case max $=$ $\mathbf{k+1}$. Assuming the theorem for all sets of pairs where L_j has k or fewer $\mathcal{R}$ symbols, consider a set of terms that has at least one pair (L_s, R_s) such that L_s contains $k+1$ $\mathcal{R}$ symbols. If R_s is a variable, then, again, by circular unification, the problem is solved in this case, so consider $R_s = \mathcal{R}(\ldots)$. For each such pair, by !-only restriction, any element in a flag position L_s or R_s must be either !, or a variable. Let F be the set of flag variables for (L_s, R_s). Let S_f be the substitution

$$S(f_i) = v_i$$

where f_i is a flag variable in position i, and v_i is the associated symbol in the opposite term. Then S_f is well defined, since v_i must be either a variable or !. So,

$$S_f(L_s), S_f(R_s)$$

agree on the flag positions. For the argument positions $A^i_{L,R}$, construct the following set of pairs

$$\{(L_1, R_1), \ldots\} - \{L_s, R_s\} \bigcup_i \{A^i_L, A^i_R\}$$

Each A^i_L has k or fewer $\mathcal{R}$ symbols, so by induction, there is a substitution S' solving the system. In particular,

$$S'(A^i_L) = S'(A^i_R)$$

So, $(S' \circ S_f)(L_s) = (S' \circ S_f)(R_s)$, and consequently, $S' \circ S_f$ is the required substitution for the original set of pairs

□

The correspondence lemmas, and the unification property of $\mathcal{R}$-terms are the basic tools for constructing the automatic insertion method. Intuitively, the idea is to replace all applications, and primitive constants with by their ec analogs. The types derived for narrowing and primitives, however, are represented by the ECG translation. Then, automatic type assignment using the methods of chapter 3 yields an assignment of types to expressions. The assignment method terminates as a consequence of the unification property. After the conclusion of the assignment process, we know that any flag defining substitution of the derived type for a narrower yields a valid ec-replacement by the correspondence lemmma. Hence *any* flag defining substitution for the new program can be translated into an ec-replacement of the original program. Therefore, we pick one according to some "reasonableness" criterion, and produce an ec-replacement program that is type correct, satisfying the goal of this chapter. The formal development follows.

First, we define the transformed program syntax. For this treatment, one need only alter the applicative terms. The transformed program is called the *ec-template.*

> **Definition 4.15** *(Ec-template)* Let P be an Exp program expression (without explicit narrowers). Let P' be the program derived from P by replacing every subterm of P of the form $(e_1\ e_2)$ by $((Isf\ e_1)\ e_2)$. P' is the *ec template* for P

Given the ec-template P' for P, the next step in the narrower insertion process is to run algorithm W on P', with a special set of type assumptions. For convenience, algorithm W is repeated below, with the Rémy encodings made explicit. For our purposes, the U is the circular unification algorithm

> **Definition 4.16** *(Algorithm W (again))* $W(A, e) = (S, \tau)$ where
>
> i). if e is x, a variable or a constant, and $A(x) = \alpha_1 \ldots \alpha_n \tau'$, then $S = \mathrm{Id}$ and $\tau = \tau'[\alpha_i \leftarrow \beta_i]$, where the β_i's are new.
>
> ii). if e is $(e_1\ e_2)$ then let $W(A, e_1) = (S_1, \tau_1)$ and $W(S_1 A, e_2) = (S_2, \tau_2)$, and $U(S_2\tau_1, \mathcal{R}(\ldots, !_{\{\rightarrow\}}, \tau_2, \beta, \ldots,) = V$, where β is new; Then $S = VS_2S_1$ and $\tau = V\beta$.

iii). if e is $\lambda x.e_1$, let β be a new type variable. Let $W(A[x:\beta], e_1) = (S_1, \tau_1)$; Then $S = S_1$, and $\tau = \mathcal{R}(\ldots, !_{\{\rightarrow\}}, S_1\beta, t_1, \ldots)$.

iv). if e is `let` $x = e_1$ `in` e_2, then let $W(A, e_1) = (S_1, \tau_2)$ and $W(S_1A[x:\overline{S_1A}\tau_1, e_2]) = (S_2, \tau_2)$; then $S = S_2S_1$ and $\tau = \tau_2$.

The algorithm fails whenever any of the above conditions are not met.

The main theorem establishes that algorithm W always succeeds for type assumptions that are !-only $\mathcal{R}$-terms (or variables). To simplify notation, let $\mathcal{R}_! = \{$!-only $\mathcal{R}$-terms$\}$

Theorem 4.5 *(Algorithm W succeeds for P')* Let P be an exp program. Further, suppose all free variables x of P have $A(x)$ defined. Finally suppose $A(x) \in \mathcal{R}_!$ for all x

Then $W(A, P) = (S, \tau)$ for some S, τ, and $\tau \in \mathcal{R}_!$, and $S(y) \in \mathcal{R}_!$ for all y.

Proof By structural induction on the shape of P.

Case e is x, a variable or a constant . By assumption, $A(x) \in \mathcal{R}_!$,so the theorem is trivially true in this case.

Case e is $(e_1\ e_2)$ By the inductive hypothesis, $W(A, e_1)$ succeeds with $S_1, \tau_1 \in \mathcal{R}_!$. Also, by the inductive hypothesis, since $S_1(y) \in \mathcal{R}_!$, $S_1A(y) \in Rs$, so inductive hypothesis applies to $W(S_1A, e_2)$. So S_2, τ_2 satisfies the conditions of the theorem. So $S_2\tau_1 \in \mathcal{R}_!$. Also, by definition,

$$\mathcal{R}(\ldots, !_{\{\rightarrow\}}, \tau_2, \beta, \ldots,) \in \mathcal{R}_!$$

So, by lemma 4.4,

$$U(S_2\tau_1, \mathcal{R}(\ldots, !_{\{\rightarrow\}}, \tau_2, \beta, \ldots,)$$

exists. Call this unifier V. Since V is a unifier over $\mathcal{R}_!$, $V(y) \in \mathcal{R}_!$. Consequently, $VS_2S_1(y) \in \mathcal{R}_!$, and $V\beta \in \mathcal{R}_!$. So, the theorem is true in this case.

Case e **is** $\lambda x.e_1$ If β is a new type variable, then $\beta \in \mathcal{R}_!$ by definition, so the inductive hypothesis applies to $A[x : \beta], e_1$. So, $W(A[x : \beta], e_1) = (S_1, \tau_1)$ with S_1, τ_1 as specified by the theorem. So $S_1\beta \in \mathcal{R}_!$, and by definition

$$\mathcal{R}(\ldots, !_{\{\rightarrow\}}, S_1\beta, t_1, \ldots) \in \mathcal{R}_!$$

So, the theorem holds in this case

Case e **is** `let` $x = e_1$ `in` e_2 . Again, by induction, $W(A, e_1) = (S_1, \tau_1)$ has the appropriate properties. So, $S_1 A$ is a type assumption of the appropriate kind, and since closure operator merely renames variables, $\overline{S_1 A}$ is a valid $\mathcal{R}_!$ only substitution. Hence, the inductive hypothesis applies to $S_1 A[x : \overline{S_1 A}\tau_1, e_2]$, and therefore

$$W(S_1 A[x : \overline{S_1 A}\tau_1, e_2]) = (S_2, \tau_2)$$

with S_2, τ_2 as required by the theorem. Hence $S_2 S_1$ also satisfies the theorem so $(S_2 S_1, \tau_2)$ meets the requirements of the theorem for this case.

$\square$

We note that the primitive constants of Exp can be assigned types in $\mathcal{R}_!$ so that both theorem 4.5 and lemma 4.3 apply. This typing for the primitive operations is termed the *ec constant template.*

Definition 4.17 *(ec constant template)* Let p be a first order primitive, with constructor constraints

$$p : T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$$

where T_i are sets of constructors, or variables. Let

$$p : \mathrm{ECG}(T_1) \rightarrow \mathrm{ECG}(T_2) \rightarrow \mathrm{ECG}(T_n)$$

be the ECG typing for p. Then the ec template for p is

$$p : \mathcal{R}(!_{\{\rightarrow\}}, \mathrm{ECG}(T_1), \mathcal{R}(!_{\{\rightarrow\}}, \mathrm{ECG}(T_2), \mathcal{R}(\ldots), \ldots), \ldots,)$$

where all flag and argument variables are distinct. In addition, define the ec template for `IsF` constant to be

$$\mathcal{R}(!_{\{!\}}, \mathcal{R}(!_{\{!\}}, \alpha, \beta, \ldots), \mathcal{R}(!_{\{!\}}, \alpha, \beta, \ldots) \ldots)$$

where all unmentioned variables are distinct.

Lemma 4.3 can be used in an elementary way to extend the correspondence to ec constant templates.

Lemma 4.5 *(Ec template correspondence)* Each flag defining substitution for an ec template corresponds to a valid ec analog.

Proof Let F be a flag defining substitution for some ec template. Then F is a flag defining substitution on its components as well. So, for each ECG subexpression, F defines some e_k according to the correspondence lemma (4.3). Thus, any F defines an ecap replacement function:

$$p_F = \lambda x_1 \ldots x_n.(p\ e_{1,F}\ e_{2,F} \ldots\ e_{m,F}\ )$$

where each $e_{i,F}$ is the e_k determined by F.

Similarly, any flag defining substitution F for the `IsF` family is, in particular, a flag defining substitution on the first argument of the form

$$\mathcal{R}(!_{\{!\}}, \alpha, \beta, \ldots)$$

Hence by lemma 4.3(second part), F determines a unique e_k: Either $((\mathtt{IsF}_T\ f)\ a)$ or $(f\ a)$. $\square$

Given these associations, we can now prove that any program P has an ec replacement program that type checks.

Theorem 4.6 *(Type correct ec replacement programs always exist)* Let P be any Exp program without free variables, and let A^* be the standard type assumption on the first order constants. Then there exists an ec replacement P^*, and a substitution S^*, and a type τ^* such that

$$S^* A^* \vdash P^* : \tau^*$$

Proof By construction. Let P' be the ec template. Let A_0 be the set of type assumptions that consists of the ec constant template for each primitive function, plus the ec-template for `IsF`. Then, by theorem 4.5, $W(A_0, P')$ succeeds with (S, τ) as its output. Let F be a flag defining substitution for the flag variables of the ec-templates in A_0, then, by syntactic completeness of algorithm W for $\vdash_{\mathcal{M}}$[17]

$$FSA_0 \vdash_{\mathcal{M}} FP' : F\tau$$

By lemma 4.5, however, F defines an ecap or an ecaa analog for each appearance of any constant. Hence FP' is a valid ec replacement program. Let $FP' = P^*$. Furthermore, by theorem 3.8, for any flag defining substitution S_f

$$C^*(S_f FSA_0) \vdash P^* : C^*(S_f F\tau)$$

Letting $S^* = C^*(S_f FS)$, and $\tau^* = C^*(S_f F\tau)$ satisfies the theorem. $\square$

Theorem 4.6 indicates an entire family of type correct ec replacements, one for each flag defining substitution F. Given the freedom of choice, one would naturally like to choose a "reasonable" F so that the number of narrowers (explicit run-time checks) is "frugal". While we cannot prove optimality, the following F^* seems a good choice.

Definition 4.18 *(A good F)* Let P' be the program of theorem 4.6, with all subexpressions annotated with their types. Then construct F^* by

$$F^*(f_i) = \begin{cases} - & f_i \neq ! \\ ! & \text{otherwise} \end{cases}$$

Intuitively, sets as many flags as possible to $-$.

Remark. Each *instance* must be considered individually, because some variables are generic (quantified), and there may be different instances in different parts of the program

[17] $\vdash_{\mathcal{M}}$ is the inference relation for the Milner system of definition 2.29

4.4 Some Observations

The purpose of the analysis in this section is to analyze what to do when the type checker of chapter 3 fails. The intuitive solution is to retain run-time checks in subexpressions where the typing process fails. The methods of this chapter use explicit functions in the form of narrowers to formalize the notion of "run-time check". The main technical results of this chapter are:

1. For a program that has explicit checks inserted for parameters of primitive functions, and in the function position for applications, type correctness insures the safety of the computation, without altering the computation.

2. The type assignment method of chapter 3 provides a mechanism for automatically constructing the explicitly checked programs.

Setting the technical considerations aside for the moment, some informal analysis of the proposed system suggests some intriguing possibilities. In particular, we would like to make some informal remarks about the information content of the narrower functions.

First, note that in some cases the source set is not very much bigger than the target set. That is, for narrower ${}_S n_T$, the set $S - T \neq \emptyset$, but it may have a small cardinality. An illustration of this phenomenon appears in example 4.1. The narrower $\mathsf{c}_1 : \mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false} \to \mathsf{z} + \mathsf{suc}$. This is the appropriate narrower *even if the first order constructors consist of many more constructors* than just the 4. The difference $S - T$ for this example indicates that the expression in question is "not too far" from being type correct. Perhaps this information might be of some benefit to a programmer in a soft typing system, giving an indication in some informal way of the degree of "wrongness" of the program.

Another potential benefit of narrowers with a small $S - T$ difference comes in the actual implementation of a run-time check. A small cardinality for the $S - T$ difference implies that fewer bits of a run-time type tag must be examined. For example, suppose run-time tags are full 32-bit words. Then $|S - T| = 2$ for c_1. Hence, only two bits of the 32 tag must be investigated to perform the type-check enforced by c_1. Bit comparison instructions on most hardware is faster than general compare-and-jump instructions.

4.5 Proofs for Chapter 4

4.5.1 Theorem 4.2 *(Soundness of Algebraic Subtype Inference)*

For any first order regular type expression T, $\mathcal{T}^e[\![\Sigma(T)]\!]\nu = \mathcal{T}^e[\![T]\!]\nu$.

Proof For this proof, we use the same notational extensions as in theorem 2.7. In particular, if E is a set of expressions,

$$\mathcal{T}^e[\![E]\!]\nu = \bigcup_{e \in E} \mathcal{T}^e[\![e]\!]\nu$$

We use structural induction.

Case $\mathbf{T = x, x \in}$ TyVars .

Here $\Sigma(x) = \{x\}$, so $\mathcal{T}^e[\![\{x\}]\!]\nu = \mathcal{T}^e[\![x]\!]\nu$.

Case $\mathbf{T = c_0, c_0 \in M}, \text{Arity}(\mathbf{c}) = \mathbf{0}$.

As for variables, $\mathcal{T}^e[\![\{c_0\}]\!]\nu = \mathcal{T}^e[\![c_0]\!]\nu$ by definition.

Case $\mathbf{T = c_n(T_1, \ldots, T_n)}$.

By the first order hypothesis, $c_n \neq \rightarrow$. Thus, it is sufficient to consider only tuples of ideals.

$$\mathcal{T}^e[\![\Sigma(T_i)]\!]\nu = \mathcal{T}^e[\![T_i]\!]\nu$$

by inductive hypothesis. Thus,

$$\langle \mathcal{T}^e[\![T_1]\!]\nu \ldots \mathcal{T}^e[\![T_n]\!]\nu \rangle = \langle \mathcal{T}^e[\![\Sigma(T_1)]\!]\nu \ldots \mathcal{T}^e[\![\Sigma(T_n)]\!]\nu \rangle$$

$$\mathcal{I}(D_{\mathbf{c_n}})(\langle \mathcal{T}^e[\![T_1]\!]\nu \ldots \mathcal{T}^e[\![T_n]\!]\nu \rangle) = I(D_{\mathbf{c_n}})\langle \mathcal{T}^e[\![\Sigma(T_1)]\!]\nu \ldots \mathcal{T}^e[\![\Sigma(T_n)]\!]\nu \rangle$$

So,

$$\begin{aligned}\mathcal{I}(D_{\mathbf{c_n}})(\langle \mathcal{T}^e[\![T_1]\!]\nu \ldots \mathcal{T}^e[\![T_n]\!]\nu \rangle) \cup F = \\ \mathcal{I}(D_{\mathbf{c_n}})\langle \mathcal{T}^e[\![\Sigma(T_1)]\!]\nu \ldots \mathcal{T}^e[\![\Sigma(T_n)]\!]\nu \rangle \cup F\end{aligned}$$

Hence,

$$\mathcal{T}^e[\![\Sigma(c_n(T_1, \ldots, T_n))\nu]\!] = \mathcal{T}^e[\![c_n(T_1, \ldots, T_n]\!]\nu$$

Case $\mathbf{T = T_1 + T_2}$.

By induction, $\mathcal{T}^e[\![T_i]\!]\nu = \mathcal{T}^e[\![\Sigma(T_i)]\!]\nu$, so

$$\mathcal{T}^e[\![T_1]\!]\nu \cup \mathcal{T}^e[\![T_2]\!]\nu = \mathcal{T}^e[\![\Sigma(T_1)]\!]\nu \cup \mathcal{T}^e[\![\Sigma(T_2)]\!]\nu$$

And thus, $\mathcal{T}^e[\![T_1 + T_2]\!] = \mathcal{T}^e[\![\Sigma(T_1 + T_2)]\!]$

Case $\mathbf{T} = \texttt{fix x.}.\mathbf{T}'$.

By definition 2.15, $\Sigma(T) = \Sigma(T')^{*x}$. By definition 2.14, $\Sigma(T')^{*x} = \sup T'_i$, where T'_i is the Kleene sequence

$$\begin{aligned} T'_0 &= T'[x \leftarrow \emptyset] \\ T'_{i+1} &= T'[x \leftarrow T'_i] \end{aligned}$$

By lemma 2.3

$$\begin{aligned} \mathcal{T}^e[\![T_0]\!]\nu &= \mathcal{T}^e[\![T']\!]\nu[x \leftarrow \{\bot\}] \\ \mathcal{T}^e[\![T'_{i+1}]\!]\nu &= \mathcal{T}^e[\![T']\!]\nu[x \leftarrow \mathcal{T}^e[\![T_i]\!]\nu] \end{aligned}$$

Note that

$$\mathcal{T}^e[\![T_0]\!]\nu = F_{T'}(\{\bot\})$$

By simple induction, (using above as base case) we note

$$\mathcal{T}^e[\![T'_i]\!]\nu = F^{i+1}_{T'}(\{\bot\})$$

The right hand side of the above is a banach sequence for $\mu\ x.F_{T'}\nu$ beginning at $\{\bot\}$. Consequently,

$$\lim \mathcal{T}^e[\![T'_i]\!]\nu = \mu x.F_{T'}\nu = \mathcal{T}^e[\![\texttt{fix}\ x.T']\!]\nu$$

The sequence T'_i is known to be telescoping, so $\mathcal{T}^e[\![T'_i]\!]\nu$ is a telescoping series of ideals. By previous analysis, it is a cauchy sequence, so

$$\sup \mathcal{T}^e[\![T'_i]\!]\nu = \mathcal{T}^e[\![\texttt{fix}\ x.T']\!]\nu$$

Elementary set manipulations yield

$$\sup \mathcal{T}^e[\![T'_i]\!]\nu = \mathcal{T}^e[\![\sup T'_i]\!]\nu$$

So

$$\mathcal{T}^e[\![\sup T'_i]\!]\nu = \mathcal{T}^e[\![\texttt{fix}\ x.T']\!]\nu$$

This is, by definition 2.14

$$\mathcal{T}^e[\![\Sigma(T'^{*x})]\!]\nu = \mathcal{T}^e[\![\Sigma(\texttt{fix}\ x.T')]\!]\nu = \mathcal{T}^e[\![\texttt{fix}\ x.T']\!]\nu$$

$\square$

4.5.2 Theorem 4.3 *(Extended Soundness for Inference Rules)*

The inference rules in definition 2.19 are sound with respect to the extended semantics. Note the use of F. from definition 2.27

Proof By structural induction on the shape of the proof.

Case $\mathbf{T_1 \subseteq T_2, T_1, T_2}$ first order This is theorem 4.2

Case $\mathbf{T \subseteq T}$ $\mathcal{T}^e[\![T]\!]\nu \subseteq \mathcal{T}^e[\![T]\!]\nu$.

Case $\mathbf{T_1 \subseteq T_1 + T_2}$ $\mathcal{T}^e[\![T_1]\!]\nu \subseteq \mathcal{T}^e[\![T_1]\!]\nu \cup \mathcal{T}^e[\![T_2]\!]\nu$

Case $\mathbf{T_1 \subseteq T_2, T_2 \subseteq T_3 \vdash T_1 \subseteq T_3}$ Again, by elementary properties of sets: If $\mathcal{T}^e[\![T_1]\!]\nu \subseteq \mathcal{T}^e[\![T_2]\!]\nu, \mathcal{T}^e[\![T_2]\!]\nu \subseteq \mathcal{T}^e[\![T_3]\!]\nu$, then $\mathcal{T}^e[\![T_1]\!]\nu \subseteq \mathcal{T}^e[\![T_3]\!]\nu$.

Case $\mathbf{T_3 \subseteq T_1, T_2 \subseteq T_4 \vdash (T_1 \to T_2) \subseteq T_3 \to T_4}$ By hypothesis for this case, and inductive hypothesis,

$$\mathcal{T}^e[\![T_3]\!]\nu \subseteq \mathcal{T}^e[\![T_1]\!]\nu, \mathcal{T}^e[\![T_2]\!]\nu \subseteq \mathcal{T}^e[\![T_4]\!]\nu$$

By definition 2.22, and definition 4.13

$$\mathcal{T}^e[\![T_1 \to T_2]\!]\nu = \{f \mid f(\mathcal{T}^e[\![T_1]\!]\nu) \subseteq \mathcal{T}^e[\![T_2]\!]\nu\} \cup F$$

But since $\mathcal{T}^e[\![T_3]\!]\nu \subseteq \mathcal{T}^e[\![T_1]\!]\nu$, any f with $f(\mathcal{T}^e[\![T_1]\!]\nu) \subseteq \mathcal{T}^e[\![T_2]\!]\nu$ is also an f

$$f(\mathcal{T}^e[\![T_3]\!]\nu) \subseteq \mathcal{T}^e[\![T_2]\!]\nu$$

Likewise, since $\mathcal{T}^e[\![T_2]\!]\nu \subseteq \mathcal{T}^e[\![T_4]\!]\nu$, any f such that $f(\mathcal{T}^e[\![T_3]\!]\nu) \subseteq \mathcal{T}^e[\![T_2]\!]\nu$ is also an f

$$f(\mathcal{T}^e[\![T_3]\!]\nu) \subseteq \mathcal{T}^e[\![T_4]\!]\nu$$

Consequently,

$$f \in \{g \mid g(\mathcal{T}^e[\![T_3]\!]\nu) \subseteq \mathcal{T}^e[\![T_4]\!]\nu\}$$

And obviously,

$$W \subseteq W$$

So

$$\mathcal{T}^e[\![(T_1 \to T_2)]\!]\nu \subseteq \mathcal{T}^e[\![(T_3 \to T_4)]\!]\nu$$

Case $\texttt{fix x.T} = \mathbf{T}[\texttt{x} \leftarrow \texttt{fix x.T}]$ By definition 4.13,

$$\mathcal{T}^e[\![\texttt{fix}\ x.T]\!]\nu = \mu F_T \nu \cup F$$

Similarly,

$$\begin{aligned} \mathcal{T}^e[\![T[x \leftarrow \texttt{fix}\ x.T]]\!]\nu &= \mathcal{T}^e[\![T]\!]\nu[x \leftarrow \mathcal{T}^e[\![\texttt{fix}\ x.T]\!]] \\ &= F_T\nu(\mu F_T \nu) \cup F \\ &= \mu F_T \nu \cup F \text{ by definition of fixed point} \end{aligned}$$

Case FIX2 The hypotheses for this rule is

$$t_1 \subseteq t_2 \vdash T_1[x \leftarrow t_1] \subseteq T_2[x \leftarrow t_2]$$

So, let $I_1 \subseteq I_2$ Since $T_1 \subseteq T_2$, the induction hypothesis yields

$$\mathcal{T}^e[\![T_1]\!]\nu[x \leftarrow I_1] \subseteq \mathcal{T}^e[\![T_2]\!]\nu[x \leftarrow I_2]$$

By definition 2.27

$$I_1 \subseteq I_2\ rmimplies\ F_{T_1}(I_1) \subseteq F_{T_2}(I_2)$$

This implies

$$F^n_{T_1}(I_1) \subseteq F^n_{T_2}(I_2)$$

Consequently,

$$\mu F_{T_1} \subseteq \mu F_{T_2}$$

Or,

$$\mathcal{T}^e[\![\texttt{fix}\ x.T_1]\!]\nu \subseteq \mathcal{T}^e[\![\texttt{fix}\ x.T_2]\!]$$

□

4.5.3 Theorem 4.4 *(Extended soundness of type inference)*

Let $A \vdash e : \sigma$ be any inference from definition 2.30. Let ρ be an environment respecting A, and let ν be any valuation. Then $\mathcal{E}[\![e]\!]\rho : \mathcal{T}^e[\![\sigma]\!]\nu$

Proof By structural induction on the last step of the inference.

Case Taut By hypothesis, ρ respects A, so

$$\mathcal{E}[\![x]\!]\rho : \mathcal{T}^e[\![A(x)]\!]\nu$$

Case Inst By rule requirements, $A \vdash e : \sigma$. So $\mathcal{E}[\![e]\!]\rho : \mathcal{T}^e[\![\sigma]\!]\nu$ by inductive hypothesis. Write $\sigma = \forall\vec{\alpha}.t, \sigma' = \forall\vec{\alpha}.t'$. This is possible by renaming bound variables. Then lemma 2.2 establishes that

$$\mathcal{T}^e[\![t]\!]\nu = \mathcal{T}^e[\![t']\!]\nu$$

Consequently, $v : \mathcal{T}^e[\![\sigma]\!]\nu$ implies $v : \mathcal{T}^e[\![\sigma']\!]\nu$.

Case Gen By inductive hypothesis,

$$\mathcal{E}[\![e]\!]\rho : \mathcal{T}^e[\![\sigma]\!]\nu$$

for all valuations ν. Since α is not free in A, any valuation $\nu[\alpha \leftarrow I]$ also maintains the theorem. In particular, let t be a regular monotype, and I_t the ideal $\mathcal{T}^e[\![t]\!]\nu$. Then

$$\mathcal{E}[\![e]\!]\rho : \mathcal{T}^e[\![\sigma]\!]\nu[\alpha \leftarrow I_t]$$

But t was arbitrary, so

$$\mathcal{E}[\![e]\!]\rho : \mathcal{T}^e[\![\forall\alpha\sigma]\!]\nu$$

Case Sub By inductive hypothesis

$$\mathcal{E}[\![e]\!]\rho : \mathcal{T}^e[\![\tau]\!]\nu$$

By theorem 4.3, if $\tau \subseteq \tau'$

$$\mathcal{T}^e[\![\tau]\!]\nu \subseteq \mathcal{T}^e[\![\tau']\!]\nu$$

Thus,

$$\mathcal{E}[\![e]\!]\rho : \mathcal{T}^e[\![\tau']\!]\nu$$

by transitivity of set inclusion.

Case Abst For any $v \in \mathcal{T}^e[\![\tau']\!]\nu$, $\rho[x \leftarrow v : \mathcal{T}^e[\![\tau']\!]\nu]$ respects $A[x : \tau']$. So, by inductive hypothesis,

$$\mathcal{E}[\![e_1]\!]\rho[x \leftarrow v : \mathcal{T}^e[\![\tau']\!]\nu] \in \mathcal{T}^e[\![\tau]\!]\nu$$

By definition 2.5

$$\mathcal{E}[\![\lambda\ x.e_1]\!]\rho(v) = \mathcal{E}[\![e_1]\!]\rho[x \leftarrow v]$$

Consequently

$$\mathcal{E}[\![\lambda\ x.e_1]\!]\rho(v) \in \mathcal{T}^e[\![\tau]\!]\nu$$

whenever $v \in \mathcal{T}^e[\![\tau']\!]\nu$. By definition 2.22, this means

$$\mathcal{E}[\![\lambda\ x.e_1]\!]\rho \in \mathcal{T}^e[\![\tau' \rightarrow \tau]\!]\nu$$

Case App By inductive hypothesis,

$$\begin{aligned} \mathcal{E}[\![f]\!]\rho &\in \mathcal{T}^e[\![\tau_1]\!]\nu \rightarrow \mathcal{T}^e[\![\tau_2]\!]\nu \\ \mathcal{E}[\![e]\!]\rho &\in \mathcal{T}^e[\![\tau_1]\!] \end{aligned}$$

By definition 2.22, any element $F \in \mathcal{T}^e[\![\tau_1]\!]\nu \rightarrow \mathcal{T}^e[\![tau_2]\!]\nu$ with $f \neq$ **wrong** has the property

$$F(\mathcal{T}^e[\![\tau_1]\!]\nu) \subseteq \mathcal{T}^e[\![\tau_2]\!]\nu$$

Since if $\mathcal{E}[\![f]\!]\rho$ is one such element

$$\mathcal{E}[\![f]\!]\rho(\mathcal{E}[\![e]\!]) \in \mathcal{T}^e[\![\tau_2]\!]$$

If $\mathcal{E}[\![f]\!]\rho = \textbf{wrong}$, then definition 2.5 insures that

$$\textbf{wrong}(\rho(\mathcal{E}[\![e]\!]) = \textbf{wrong} \in \mathcal{T}^e[\![\tau_2]\!]$$

Case Let By the inductive hypothesis,

$$\mathcal{E}[\![e_1]\!]\rho : \mathcal{T}^e[\![\sigma]\!]\nu$$

From definition 2.5

$$\mathcal{E}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] = \mathcal{E}[\![e_2]\!]\rho[x \leftarrow \mathcal{E}[\![e_1]\!]\rho]$$

So $\rho[x \leftarrow \mathcal{E}[\![e_1]\!]\rho]$ respects the assumption

$$A \cup \{x : \sigma\}$$

Consequently, the inductive hypothesis applies to the second condition for this inference rule, and we have

$$\mathcal{E}[\![e_2]\!]\rho[x \leftarrow \mathcal{E}[\![e_1]\!]\rho] : \tau$$

Putting this together yields

$$\mathcal{E}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] : \tau$$

$\square$

Chapter 5

Perspectives: Related Work and Future Work

To put this thesis in perspective, we must compare it to related work and identify directions for future research. The analysis of related work appears in section 5.1. Following that, section 5.2 explores some avenues for future research based on the thesis results. Finally, section 5.3 summarizes the research contribution of the thesis.

5.1 Related Work

At the inception of this research, in January of 1987, there was virtually no literature seeking to integrate the benefits of static typing into a dynamically typed language. In early 1989, however, the static typing community began to investigate language constructs that incorporated some dynamic typing ideas into statically typed languages. These results are philosophically similar to the viewpoint espoused in this thesis. These philosophically similar investigations are the subject of section 5.1.1.

Even though philosophically similar work now exists, the primary work forming the basis for this thesis came from the enormous body of static typing literature. The static typing literature that had the most influence on the thesis are analyzed in section 5.1.2

5.1.1 Philosophically Similar Work

In 1989, Abadi, Cardelli, Pierce, and Plotkin [1] devised a statically typed language containing two new language elements, `typecase`, and `dynamic` and a new type, `Dynamic`. These purpose of these new elements was to add dynamic typing features to a statically typed language. The `typecase` construct is analogous the suite of `case` functions introduced as part of the first order specification. That is, `typecase` checks the type tag of its argument, and performs subsequent computation based on that tag. The selector value, however, must have type `Dynamic`.

The `dynamic` feature converts static values to type Dynamic by explicitly tagging them with a given type. That is, `dynamic` $x : T$ creates a value of type Dynamic with tag T. The user, however, must supply the T.

While the addition of these features increases the expressiveness of a language having only static typing, these features do not accommodate the soft typing paradigm for the following reasons:

1. The system is declarative. That is, types must be supplied by the user both as declarations, and for the `dynamic` construct.

2. There are no unions, or recursive types. That is, all union and recursive phenomena must be accommodated via explicit use of `dynamic` and `typecase`.

3. The specific order of tagging makes a difference. That is, `dynamic` $5 : int$, and `dynamic` (`dynamic` $5 : int$) are different values. Consequently, a programmer must not only know the underlying types, but also the tagging pattern.

   In contrast, a value in a dynamically typed language has only one type tag, so alternative "tagging patterns" cannot arise.

4. The language is still statically typed. That is, certain programs are still rejected.

In retrospect, the design goal for this language was to incorporate a dynamically typed sublanguage in a statically typed host language. To maintain the integrity of the static typing, the system must include a wall to separate the dynamic capabilities from the static ones. As such, the properties of the type Dynamic language differ significantly from the desired properties of a softly typed language.

Thatte [49] considers a system he calls "Quasi-static typing" to accommodate a merger of static and dynamic typing. The salient features of his system are:

1. A type Ω, such that any type $\tau \leq \Omega$. Furthermore, the $\leq$ relation extends monotonically over all type constructors, except for the usual antimonotonic behavior of the arrow constructor's first argument. The type Ω is intended to simulate dynamic types.

2. The use of coercion functions both of the form $\uparrow_\tau^\Omega$ and $\downarrow_\tau^\Omega$ to implement run-time checking.

Thatte's system resembles our soft typing system in three important ways. First, the Ω type introduces a notion of subtyping into the type system. Furthermore, the subtype relation is structural. This means that the "tagging pattern" anomaly of the `dynamic` is eliminated. Second, the coercion functions $\downarrow^{\Omega}_{\tau}$ operate as narrowers for a quasi-static system. The other coercions appear to be injections, or "tagging" functions. Third, Thatte automatically inserts the run-time checks (coercions) in such a way as to insure a type correct program. So, Thatte does not reject any programs. These similarities make quasi-static typing a close cousin of soft typing.

There are a few significant differences separating soft typing and quasi-static typing. Most importantly, the quasi-static type discipline requires argument type declaration. The program

$$(\lambda x : \mathsf{Int}.x\ 5)$$

is fundamentally different from

$$(\lambda x : \Omega.x\ 5)$$

in that the *declaration* affects whether or not a run-time coercion is inserted. The former program has no coercions, whereas the latter program is implemented as:

$$(\lambda x : \Omega.x\ \ \uparrow^{\Omega}_{\mathsf{Int}} 5)$$

Furthermore, argument type declarations preclude parametric polymorphism.

Another distinction between quasi-static typing and soft typing is, again, the absence of unions and recursive types. The types are included in soft typing systems to avoid excessive false error reporting. Thatte himself states

> ...interaction may become tedious if intended checks are numerous

These limitations mean a quasi-static system will almost certainly "cry wolf" more often than a soft typing system. An additional advantage derived from union types is the spectrum of coercions is much wider. As in example 4.1, the function in question can be made safe with a narrower whose type is

$$\mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false} \rightarrow \mathsf{z} + \mathsf{suc}$$

where the difference between the source and target has cardinality 2. In a quasi-static system, the only coercion available has Ω as its source, so all possible types in the system must be checked, not merely the 4 listed above.

In essence, a quasi-static system is *almost* a soft typing system. The quasi-static systems, however emphasize the *static* aspect too heavily to be a viable soft typing system.

5.1.2 Related Static Type Systems

In the course of this research, a significant number of type systems and analysis methods came under scrutiny. All systems considered had some potential relevance to the soft typing problem. The following remarks indicate the salient features of a good cross section of the systems scrutinized during the course of this thesis.

Systems with similar constructs

Mishra and Reddy [35] develop a type system for Prolog programs that includes both union types and recursive types. Since the intended programming language is Prolog, however, there are no function types. The type assignment method used for this system, however, also recognizes the usefulness of restricting types to be discriminative.

Similarly, Amadio and Cardelli [2] describe an inference system accommodating both recursive types and subtypes. Their system makes use of rational trees in the inference process.

By using intersection (conjunctions) instead of unions, Coppo,Dezani-Ciancaglini, and Venneri [13] develop a type system that assigns a type to all lambda calculus normal forms. The system, however, is undecidable.

Systems with general subtyping

By viewing subtyping as coercion application, Mitchell [36] introduced a type system with an extremely liberal view of subtyping. He was the first to note that systems with subtypes cannot have principal types in the Milner sense. Consequently, the type expressions in his system included the inequalities as part of the type as in:

$$t, \{\alpha \subseteq \beta, (\alpha \to \beta) \subseteq t\} \vdash \lambda x.x : t$$

Fuh and Mishra [22, 23] improve on Mitchell's methods, and Fuh's thesis [21] addresses the problem of the actual insertion of the coercions necessary to support the supplied typing. These systems have three problem drawbacks when considered as possible soft typing systems

1. The type expressions are unintuitive and somewhat unwieldy. In spite of some success in reducing the number of constraints, types are still overly complicated

2. There is no parametric polymorphism

3. The decidable portion of this theory does not support functions as elements of types that may include non-functional objects

Pavel Curtis's thesis [15] develops a similar idea that he calls "constrained quantification". He retains the constraints as part of the type. In contrast to the Fuh-Mishra system, Curtis allows parametric polymorphism. The decidability of the system in its general form, however, is still an open question. Furthermore, the presence of constraints as part of the type still manifests the overcomplex expression objection to the Fuh-Mishra systems.

In conclusion, we note that *solving* the constraint systems generated is the subject of chapter 3.

"Object" oriented systems

One exceptionally populated area of type research literature involves the notion of "object-oriented" programming. These systems are especially interesting for soft typing considerations because the object oriented data, namely "records" and "variants", possess a property greatly resembling subtyping. This property is called inheritance in that body of literature.

The first discernable "object-oriented" type system originates with Cardelli [9]. This work introduces the notions of records and variants, and provides a type-checking algorithm for a simple language including the new constructs. Researchers building on this work include Stansifer [44], Wand [53, 55, 56], Jategaonkar-Mitchell [29] and Rémy [39]. As evidenced in chapters 3 and 4, the Rémy work provided a clear basis for the research in this thesis. The Rémy work provided two attractive features in comparison to others:

1. The technique encoded inheritance as standard parametric polymorphism, so that a standard Milner technique was applicable

2. The use of circular unification provided the needed mechanism for recursive types

5.2 Future Work

Research on extensions to any type discipline typically proceeds independently along two different directions. One direction involves handling more programming constructs. The other direction concerns strengthening the discipline to accommodate more programs.

Language extensions of interest include assignment, and advanced control constructs, such as `call/cc`. All of these constructs are present in Standard ML of New Jersey, as well as Scheme. Consequently, we would like to extend our type system to accommodate these programming language features, in much the same way as Tofte [50] for statically typed languages. In addition, the module constructs of ML are worthy of some attention.

As for improving the type methods, we see at least two interesting avenues for exploration. First, the inferior type assigned to the **deep** function merits attention. Some preliminary analysis indicates that the typing can be improved by inserting some special coercions. It remains to be seen if the method can be automated and generalized. Another promising area of research involves adding some conjunctive typing to the type system. One must be careful, as Coppo [13] has shown that conjunctive typing is, in general, undecidable. Perhaps some compromise along the lines proposed by Ghosh-Roy [26].

Concerning the coercion insertion method of chapter 4, we need to determine the "optimality" of the method. We need to determine if the number of coercions inserted can be made smaller or not. If it cannot, we should show this. If some other method produces a lesser number of these coercions, then it should be adopted. We conjecture that our method is optimal.

Finally, but perhaps most importantly, we need to produce a good quality implementation of these ideas so that reasonable comparisons with other systems can be made. One interesting experiment to run would compare the number of programs from some elementary scheme text that type check without coercions in the soft type system.

5.3 Conclusions

In broad terms, the contribution of this research is the development of the necessary technical tools to support the soft typing paradigm. The tools developed include:

1. A type system specifically designed to accommodate a large percentage of programs in dynamically typed languages. In particular, the type system incorporates union types, recursive types, and parametric polymorphism. Additionally,

a sound type inference system for annotating program expressions with these types is provided

2. A simple, well understood method for automatically assigning types to programs, without the need for any extraneous declarations.

3. A method for explicitly inserting run-time checks into programs for which the type assignment method fails. Consequently, *any* program may be safely executed.

In conclusion this research supports the following thesis:

> Soft typing is viable. Programmers may derive the verification benefits and optimization opportunities of static typing while retaining *all* the expressiveness of dynamic typing.

Bibliography

[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. In *Proceedings of the Sixteenth POPL Symposium*, 1989.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, August 1990.

[3] M.A. Arbib and Y. Give'on. Algebra automata i: Parallel programming as a prolegomena to the categorical approach. *Information and Control*, 12, 1968.

[4] Henk Barendregt and Kees Hemerik. Types in lambda calculi and programming languages. In *3rd ESOP*, 1990.

[5] Walter S. Brainerd. The minimalization of tree automata. *Information and Control*, 13, 1968.

[6] Stan Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Springer-Verlag, 1981.

[7] Burstall, MacQueen, and Sannella. Hope: An experimental applicative language. In *Proceedings of the first international LISP conference*, 1980.

[8] W. Büttner. Unification in the data structure sets. In *Proceedings of the 8th International Conference on Automated Deduction*, July 1986.

[9] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[10] Robert Cartwright. A constructive alternative to axiomatic data type definitions. In *Proceedings of 1980 LISP Conference*, 1980.

[11] Robert Cartwright. Types as intervals. Technical report, Rice Univesity, 1984-1985.

[12] Alain Colmerauer. Prolog and infinite trees. In K. L. Clark and S. A. Tarnlund, editors, *Logic Programming*, pages 231–251. Academic Press, 1982.

[13] M. Coppo, M. Dezani, and B. Venneri. Principal type scheme and λ-calculus semantics. In J. P. Hindley and J. R. Seldin, editors, *To H. B. Curry. Essays on Combinatory Logic, λ-Calculus and Formalism.* Academic Press, 1980.

[14] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, March 1983.

[15] Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox PARC, 1990.

[16] O. J. Dahl, C. A. R. Hoare, and E. W. Dijkstra. *Structured Programming.* Academic Press, 1972.

[17] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, 1982.

[18] Luis Manuel Martins Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

[19] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 1991.

[20] James Dugundji. *Topoloy.* Allyn and Bacon, 1966.

[21] You-Chin Fuh. *Design and Implementation of a Functional Language with Subtypes.* PhD thesis, State University of New York at Stony Brook, 1989.

[22] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *Conference Record of the European Symposium on Programming*, 1988.

[23] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT*, 1989.

[24] Gannon. An experimental evaluation of data type conventions. *Communications of ACM*, August 1977.

[25] Ferenc Géceg and Magnus Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[26] R. Ghosh-Roy. Conjunction type standard ml polymorphism. *Lisp and Symbolic Computation*, 3(IV):381–410, December 1990.

[27] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[28] Gérard Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris, 7 1976.

[29] Lalita A. Jategaonkar and John C. Mitchell. Type inference with subtypes. In *Proceedings of the 1988 Conference on LISP and Functional Programming*, 1988.

[30] Patrick Lincoln and Jim Christian. Adventures in associative-commutative unification. In *Proceedings of the 9th International Conference on Automated Deduction*, May 1988.

[31] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1983.

[32] D. B. MacQueen and Ravi Sethi. A semantic model of types for applicative languages. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, 1982.

[33] Michael Maher. Complete axiomatizations of finite, rational, and infinite trees. In *3rd Logic In Computer Science Conference*, 1988.

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

[35] Prateek Mishra and Uday S. Reddy. Declaration-free type checking. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, 1984.

[36] John C. Mitchell. Coercion and type inference. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1983.

[37] A. M. Morshedi and R. A. Tapia. Karmarkar as a classical method. Mathematical Sciences 87-7, Rice University, Aug 1987.

[38] Neal Nelson. Primitive recursive functionals with dependent types. Private communication, Feb 1990.

[39] Dider Rémy. Typechecking records and variants in a natural extension of ml. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.

[40] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of ACM*, December 1965.

[41] David A Schmidt. *Denotational Semantics.* Allyn and Bacon,Inc, 1986.

[42] Dana Scott. Data types as lattices. *Siam Journal of Computing*, 1976.

[43] Jorg H Siekmann. Unification theory. *Journal of Symbolic Computation*, 7(3 and 4), 1989.

[44] Ryan Stansifer. Type inference with subtypes. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988.

[45] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1979.

[46] R. A. Tapia. On the role of slack variables in quasi-newton methods for constrained optimization. In L.C.W. Dixon and G. P. Szegö, editors, *Numerical Optimization of Dynamic Systems*, pages 235–246. North-Holland Publishing Company, 1980.

[47] J. W. Thatcher. Characterizing derivation trees of context free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 1, 1967.

[48] J.W. Thatcher and J.B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1), 1966.

[49] Sattish Thatte. Quasi-static typing. In *Proceedings of the Seventeenth POPL Symposium*, 1990.

[50] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1987.

[51] David A. Turner. Miranda – a non-strict functional language with polymorphic types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1985.

[52] Mitchell Wand. A types-as-sets semantics for milner-style polymorphism. In *11th POPL*, 1984.

[53] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the Second Symposium on Logic in Computer Science*, 1987.

[54] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informatica*, X:115–122, 1987.

[55] Mitchell Wand. Corrigendum:complete type inference for simple objects. In *Proceedings of the Third Symposium on Logic in Computer Science*, 1988.

[56] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *4th Annual Symposium on Logic in Computer Science*, 1989.