IDEA GROUP PUBLISHING **ITB13243**

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

This paper appears in the publication, *Verification, Validation, and Testing in Software Engineering*
edited by Aristides Dasso, Ana Funes © 2007, Idea Group Inc.

Chapter XI

Static Type Systems: From Specification to Implementation

Pablo E. Martínez López,
LIFIA, Facultad de Informática, UNLP, Argentina

Abstract

Static type systems are fundamental tools used to determine properties of programs before execution. There exist several techniques for validation and verification of programs based on typing. Thus, type systems are important to know for the practicioner. When designing and implementing a technique based on typing systems, there is usually a gap between the formal tools used to specify it, and the actual implementations. This gap can be an obstacle for language designers and programmers. A better understanding of the features of a type system and how they are implemented can help enourmously to the good design and implementation of new and powerful verification methods based on type systems. This chapter addresses the problem of specifing and implementing a static type system for a simple language, but containing many of the subtleties found in bigger, mainstream languages. This contributes to the understanding of the techniques, thus bridging the gap between theory and practice. Additionally, the chapter contains a small survey of some of the existing developments in the static typing area and the static analysis based on typing.

Introduction

When someone speaks about verification and validation of programs, it is not very common that a static type system comes to one's mind. However, static typing techniques have been a foundation for many of the developments in the theory and practice of this area of computing science.

A static type system (Cardelli, 1997; Cardelli & Wegner, 1985; Curry & Feys, 1958; Hindley, 1969; Hindley, 1995; Milner, 1978; Pierce, 2002) is a fundamental tool used to determine properties of programs before execution — for example, the absence of some execution errors — and to provide rudimentary documentation when coding solutions in a programming language. The main motivation for the use of these systems is that every program with a type calculated statically — that is, based only on the text of the program and not on its computation — is free from some kinds of errors during execution. The classic motto coined by Robin Milner is "well-typed programs cannot go wrong" (1978). This guarantees a certain correctness of the code and also helps omit specific checks in the executable code to avoid these kinds of problems, thus obtaining a more efficient program.

When designing static type systems, a trade-off between expressiveness and decidability has to be made. The computation prescribed by a static type system is limited — it may happen that given a type system, a decidable inference algorithm does not exist; it is usually said that the system is not decidable. So, the choices are: Design a decidable system discarding some number of correct programs, or design a precise system but with a noncomputable notion of typing. Both alternatives have been thoroughly studied.

When choosing the first alternative, the goal is to maximize the expressiveness of the system — that is, the number of correct accepted programs be as large as possible, while minimizing the number of incorrect accepted ones — without losing decidability. In this line of research appear polymorphic type systems (Damas & Milner, 1982; Jim, 1996; Milner, 1978; Reynolds, 1983) as the one of ML (Clément, Despeyroux, Despeyroux, & Kahn, 1986; Rèmy & Pottier, 2004) or Haskell (Peyton Jones & Hughes, 1999), and systems with subtypes (Henglein & Rehof, 1997; Mitchell, 1991; Palsberg, Wand, & O'Keefe, 1997), overloading (Jones, 1994a; Jones, 1996; Thatte, 1992), recursive types (Tiuryn & Wand, 1993), records (Gaster & Jones, 1996; Rèmy, 1989), and so on.

In the second case, the idea is to design semiautomatic tools to help in the construction of programs, maximizing the ability to perform automatic type inference. In this line of research appear Girard's System F (Girard, 1989; Reynolds, 1974), and dependent type systems as the Type Theory of Martin-Löf (Nordström, Petersson, & Smith, 1990) — on which a tool like Alf (Thorsten, Altenkirch, Gaspes, Nordström, & von Sydow, 1994) is based — and the calculus of constructions (Coquand & Huet, 1988) — on which the Coq proof assistant (Bertot & Castéran, 2004; The Coq proof assistant, 2004) is based.

This chapter concentrates on the first of these two choices.

Static type systems are an integral part of the definition of a language, similar to the grammar defining the syntax of programs. Furthermore, just as there is a tool implementing the grammar — the parser — there is a tool implementing the type system. It can be either a type checker, or a type inferencer (the difference between typechecking and type

inference will be discussed in subsequent sections). Usually, there is a gap between the formal tools used to specify a type system, and the actual implementations. Once a type system has been designed, finding an algorithm for typechecking or type inference may be from really easy to extremely hard, or even impossible. There are usually two hindrances when implementing type inference: inherent subtleties of the system and techniques used to improve performance. This gap can be an obstacle for language designers and programmers. A better understanding of the features of a type system, and how they are implemented can help enourmously to the good design and implementation of new and powerful verification methods based on type systems.

Type systems are intrinsically related to the semantics of the language. The main property of a type system is *soundness*, which states that any typeable program is free of the errors addressed by the system. However, a purely syntactic treatment of types is enough for the purpose of this chapter — that is, no formal semantics will be given to the language, and no proof of soundness for the type system. These issues have been addressed in the literature — Reynolds (1998), for example.

What this chapter addresses is the problem of specifying and implementing a static type system for a simple language, but which contains many of the subtleties found in bigger, mainstream languages. The chapter begins explaining in detail the basic techniques for specification of the type systems and then discusses a straightforward way to implement it, showing the relationship between the two. This contributes to the understanding of the techniques, thus bridging the gap between theory and practice; this contribution may help in the design of other techniques for verification and validation of programs. Continuing with the chapter, a small survey of some of the existing developments in the static typing area and the static analysis based on typing are given, such as the addition of records and variant types, recursive and overloaded types, register allocation, sized types, type specialization, and so on, is given. Finally, some conclusions are drawn.

The Basic Language and Its Typing

The development of a formal system — a system used to capture essential features of the real world by means of a formal language and deductive mechanisms — always starts by defining the syntax of the languages involved. The language used in this chapter is a very simple one, but has enough constructs to present the more subtle problems of typing: It has numbers, Booleans, higher-order functions, tuples, and recursion. A functional language (Bird, 1998) has been chosen because of its simplicity, allowing for easy illustration of the features of the type system.

Functional programming is a programming paradigm that concentrates on the *description of values* rather than on the sequencing of commands. For that reason, its semantics is easy, assigning a value to each valid expression. The interesting feature about functional programming is the existence of a special kind of values, called *functions*, that are used to describe the relation between two sets of values and that can also be interpreted as the prescription of information transformation — that is, a *program*. In functional languages, functions are treated just like any other value, and thus can be

passed as parameters, returned as values, and used in data structures, and so the expressive power is very high.

- **Definition 2.1:** *Let x denote a term variable from a countably infinite set of variables, n, a number, and b, a Boolean (that is, either* **True** *or* **False***). A* term*, denoted by e, is an element of the language defined by the following grammar:*

$$
\begin{array}{lllll}
e & ::= & x & \mid n & \mid e + e \\
 & \mid & b & \mid e == e & \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \\
 & \mid & (e, \ldots, e) & \mid \pi_{n,n}\, e & \\
 & \mid & \lambda x.e & \mid e\ e & \mid \textbf{fix } e \\
 & \mid & \textbf{let } x = e \textbf{ in } e & &
\end{array}
$$

where + is chosen as a representative numeric operation, and == as a representative relational operation.

The expression $(e_1,\ldots,e_n)$ in the last definition represents a tuple of arity n for every positive number n (this is an ordered pair when $n = 2$), and $\pi_{i,n}\ e$ is the i-th projection of the n-tuple e. The expression $\lambda x.e$ is an anonymous function that returns e when given an argument x, and $e_1\ e_2$ is the application of the value of e_1 — expected to be a function — to argument e_2. The expression **let** is used to provide (self-contained, i.e., in a single expression) local definitions, and the expression **fix** e provides recursion vía fixpoint operators (Mitchell, 1996). A *fixpoint* of a given function f is a value of its domain that is returned without changes by f. When writing expressions, some conventions are taken: Addition, multiplication, and other operators have the usual precedence; functional application has higher precedence than any other operator, and associates to the left; and finally, parentheses are used whenever needed to eliminate ambiguity (even when no parenthesis are explicitly given in the grammar). This is an example of abstract syntax as seen in McCarthy (1962) and Reynolds (1998).

A program is any closed term of this language — that is, those with no free variables. To define the concept of free variable, the notion of *binder* (i.e., the *definition point* of a variable) is needed, together with its *scope* (i.e., where the definition is valid); those occurrences of the variable defined by the binder that appear in the scope are called bound occurrences, and the variable is said to be a bound variable; those variables that are not bound are said to be *free*. For the language used here, the binders are the λ and **let** expressions, and the scopes are the body of the λ-expression, and the expression following the **in** in the **let** expression, respectively.

Very simple examples are arithmetic expressions, like 1 + 1, and (4 - 1)*3, relational expressions, like $x == 0, 2 + 2 > 4$, etc., and example of tuples and projections, like (2, **True**), $\pi_{2,2}\,(\pi_{2,2}\,(1,(\textbf{True},3)))$, etc. Still simple, but less common, are anonymous functions, such as $(\lambda x.x)$, $(\lambda p.\pi_{1,2}\,p)$, etc.

Growing in complexity, there are examples using local definitions and conditionals:

let *fst* = $\lambda p.\ \pi_{1,2}\ p$
in *fst* (2,5)

and

let *max* = $\lambda x.\ \lambda y.$ **if** $x>y$ **then** x **else** y
in *max* 2 5

Finally, recursion is the most involved feature; the factorial is an example of a recursive function:

let *fact* = **fix** ($\lambda f.\ \lambda v.$ **if** $v == 0$
then 1
else $v * f\ (v - 1)$)
in (*fact* 2, *fact* 5)

Observe how recursion is expressed using the **fix** operator. An equivalent way to provide this is to use a construct similar to **let**, but allowing recursive definitions (typically called **letrec**, and assumed implicitly in many languages); it can be defined as syntactic sugar, by using **fix** in the following way:

letrec $f = e$ **in** e' ≡ **let** f = **fix** ($\lambda f.e$) **in** e'

In this way, the previous example can be rewritten as:

letrec *fact* = $\lambda v.$ **if** $v == 0$
then 1
else v * *fact* ($v - 1$)
in (*fact* 2, *fact* 5)

which is a more common way of finding recursion. Had this construct been added instead of **fix**, the set of binder expressions would have to be extended with it, and the scope of the new binder defined as both expressions appearing in the **letrec**.

A naive approach for ruling out typing errors is using syntactic categories to discriminate expressions of different types — for example, instead of having a single syntactic category *e* of programs, a naive designer would have defined one category e_i for integer

expressions, one category e_b for Boolean expressions, and so on. The main problem is that the set of typed terms is not context free. This is shown in two ways: First of all, there are expressions in the language that can take different types — for example, conditionals, or local definitions — and so they have to be replicated on each syntactic category; but second, and much more important, there is usually an infinite number of sets of values to discriminate, but there cannot be an infinite number of syntactic categories — context-free grammars are finite tools.

The specification of the type system is given in two steps: First, all the nonfunctional parts (that is, numbers, Booleans, and tuples) are presented, and then, functions are added to the system. The reason is that functions introduce many of the subtleties that make the system complex (and interesting).

Typing Nonfunctional Elements

The type system is specified as a relation between an expression and a *type*, denoted by τ, which is a term of a new language used to classify programs according to the kind of value they may produce when executed (if they terminate). In a naive interpretation, a type can be identified with a set of values.

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

  $\tau ::= \textbf{Int} \mid \textbf{Bool} \mid (\tau, \ldots, \tau) \mid \tau \to \tau$

Naive interpretations for types are the set of numbers for type **Int**, the set of truth values for type **Bool**, the cartesian product of types for tuples, and the function space for the function type ($\to$).

A key property that any static analysis must have is that it has to be *compositional* — that is, the analysis of a given program has to be constructed based on the result of the analysis of its subprograms. Type systems are no exception. However, the restriction that programs are closed terms complicates achieving this property, because subprograms do not fulfil it; they may contain free variables. This is a very common problem when defining a static analysis — that is, subexpressions may have different restrictions than full programs — and several ways to address it exist. Here, *environments* are used; they are similar to the symbol table of a compiler.

- **Definition 2.3:** *A* typing environment*, denoted by* Γ*, is a (finite) list of pairs composed by a variable and a type, written* $x : \tau$*, where all variables appear at most once.*

The idea of a typing environment is to provide information about the type of free variables. However, in order to allow flexibility (needed in the formal development to

prove the desired properties), larger environments — that is, with more than just those variables free in the term — can also be used when typing a term. An operation used for environments is the elimination of a pair containing a given variable x, denoted by Γ_x; this is needed before extending the environment with a new pair for x.

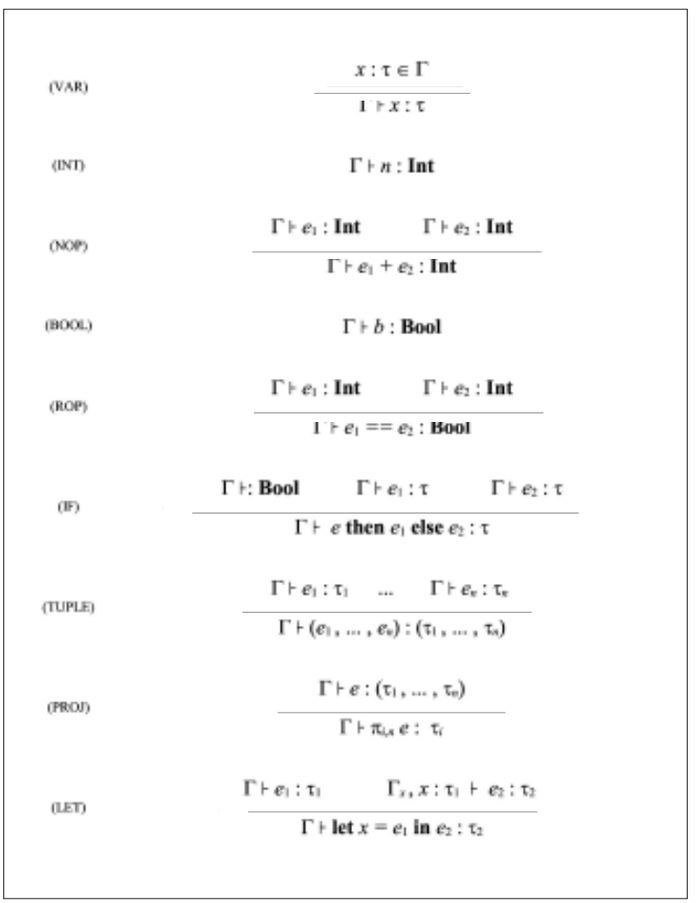
The typing relation takes three elements: a typing environment, a term, and a type. It is denoted by the statement $\Gamma \vdash e : \tau$ (written $\vdash e : \tau$ when Γ is empty); this statement is called a *typing judgement*. There are two ways to read a typing judgement: The first says that Γ, e, and τ are in the typing relation, and the other says that if *the typing environment* Γ *associates the free variables of e with some types, then e has type* τ. The former brings out the relational nature of the typing, while the latter emphasizes the algorithmic nature; the usual reading is the latter. When the elements appearing in the judgement are in the typing relation, the judgement is called *valid* (and the term, *well typed*) and if that is not the case, it is called *invalid* (and the term, *ill typed*). Thus, the typing relation is completely specified by the set of all valid judgements. For example, $\vdash 1 + 1 :$ **Int** is a valid judgement, while $\vdash 2 +$ (True,5) : **Bool** is invalid. It is important to understand that well typed terms are free of certain execution errors, but ill typed terms may either contain those errors, or simply cannot be assured that they are free of errors. In the example above, the term contains an error: The representation in memory for numbers and pair are surely different (e.g., two bytes for the number and three for the pair), but addition will expect to receive a two-bytes operand; thus if the addition is executed, it will either need to check that each memory location corresponds to the numbers, or wrongly access the memory portion of the pair as if it corresponds to a number! For an ill-typed term that does not contain errors, see the last example in Section "Designing an Algorithm for Type Inference."

There are several ways to determine when a typing judgement is valid. The most common is to use a system of rules, called the *typing system* or *type system*. This system is a particular case of a formal proof system (collection of rules used to carry out stepwise deductions). Each rule has some judgements as premises and a judgement as conclusion, allowing in this way the desired compositional nature of the specification; when the number of premises of a rule is zero, it is called an axiom. Rules are written with premises above a horizontal line, and the conclusion below it; if some condition is needed for the application of the rule, it is written by its side. A rule states that when all premises are valid (and the conditions satisfied), so is the conclusion; thus, axioms are direct assertions that a judgement is valid.

Using the rules, a tree, called a *derivation tree* or also a *typing derivation*, or *derivation* for short, can be constructed; it has judgements as labels, and instances of rules as nodes; leaves are instances of axioms. One important property of derivation trees is that the judgement labelling the root of these trees is always a valid judgement. The common method to determine the validity of a given judgement is to build a derivation tree with it as the root; if no such tree exists, the judgement is invalid.

The typing rules for the nonfunctional part of the language are given in Figure 1. The rules are named for easy reference. Rule (VAR) establishes that Γ, x, and t are in the typing relation for every Γ containing the pair $x : \tau -$ ◠r, using the algorithmic reading, if the typing environment G associates x with type τ, then x has type t. Rule (INT) states that numbers have type **Int** in any environment, and similarly, rule (BOOL) states that Boolean

Figure 1. Typing system (non-functional part)



constants have type **Bool**. Rules (NOP) and (ROP) provide the typing of arithmetic and relational operations, respectively. Rule (IF) gives the typing for conditional constructions: If the guard is a Boolean and the two branches have the same type t, then the conditional have type t as well. The typing for tuples and projections is given by rules (TUPLE) and (PROJ), with straightforward meaning. Finally, the typing for local definitions provided by **let** expressions is given by rule (LET): The free ocurrences of *x* in the body of the **let** are assumed to have the type of the definition of *x*, and then the whole expression have the type of the body.

Once the system has been given, derivation trees can be constructed to prove that a given judgement is valid. For example, in order to prove that $\Gamma \vdash 1 + 1 : \mathbf{Int}$ is valid, the following derivation can be constructed for any Γ:

$$\frac{\Gamma \vdash 1 : \mathbf{Int} \qquad \Gamma \vdash 1 : \mathbf{Int}}{\Gamma \vdash 1 + 1 : \mathbf{Int}}$$

On the other hand, there are no derivations for the judgement $\vdash 2 + (\mathbf{True}, 5) : \mathbf{Bool}$, because the only possibility to construct one is to use rule (NOP), but the second premise is not of the right form; it is expected that the operand must be an **Int**, and here it is a pair.

A more complex derivation is given for a term using a **let** and a conditional. The example has x and y as free variables, and for that reason, the pairs $x : \mathbf{Int}$ and $y : \mathbf{Int}$ must appear in the environment used in the judgements — in the derivation given, the environment $\Gamma^{xy} = x : \mathbf{Int}, y : \mathbf{Int}$ is used, which is the minimal useful one. (After introducing functions, the variable *max* shall be transformed into a function.)

$$\dfrac{\dfrac{\dfrac{\Gamma^{xy} \vdash x : \mathbf{Int} \quad \Gamma^{xy} \vdash y : \mathbf{Int}}{\Gamma^{xy} \vdash x > y : \mathbf{Bool}} \qquad \dfrac{\Gamma^{xy} \vdash x : \mathbf{Int} \quad \Gamma^{xy} \vdash y : \mathbf{Int}}{\Gamma^{xy} \vdash \mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y : \mathbf{Int}} \qquad \dfrac{\Gamma^{xy}, max : \mathbf{Int} \vdash max : \mathbf{Int} \quad \Gamma^{xy}, max : \mathbf{Int} \vdash 1 : \mathbf{Int}}{\Gamma^{xy}, max : \mathbf{Int} \vdash max + 1 : \mathbf{Int}}}{\Gamma^{xy} \vdash \mathbf{let}\ max = \mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y\ \mathbf{in}\ max + 1 : \mathbf{Int}}}$$

Observe, in particular, how the axiom stating that 1 has type **Int** in the right subtree of the derivation uses a larger environment than the minimal one needed for that judgement in isolation.

There are two main ways to use a type system: It can be used to check if a triple Γ, e, τ belongs to the relation, or, given Γ and e, a type τ can be constructed such that they are related (if it exists). The former is called *typechecking*, while the latter is called *type inference*.

One important property of this fragment of the type system is that it is *functional* — that is, given Γ and e, there exists at most one single type τ related to them. This facilitates the implementation of type inference, and reduces typechecking to infer the type and compare it with the given one. In order to compute the type of an expression, the implementation of type inference will recursively compute the type of each subexpression and then compare them to those needed to build a derivation. For example in (NOP), after the types of the operands have been obtained, they can be directly compared with **Int**, and if they are different, an error can be returned immediately, and similarly in other rules, as (IF). The functionality of the system comes from the fact that all the decisions have only one choice, and thus they can be taken locally. However, it is not always the case that static analyses are functional. Several decisions have multiple choices, and the information to decide about the right one can appear in some other part of the derivation. (See the example of the identity function in the next section.) So, more complex techniques are needed for implementation.

Typing Functions

The real expressive power of this language comes with the use of functions. The expression ($\lambda x.e$) denotes an anonymous function: When this function is given an argument in place of x, the body e of the function is returned. Thus, the type of this expression is a function space: If under the assumption that x has type τ_2, e has type τ_1, then the function has type ($\tau_2 \to \tau_1$). Corresponding with this notion, the application ($f\,e$) is the operation that provides the argument e to a function f: if f is the function ($\lambda x.e'$), the meaning of ($f\,e$) is the value of the body e', where x is substituted by e. For example, when f is ($\lambda x.x+x$), the meaning of (f 2) is 2 + 2, that is, 4.

A common notational convention is that function types associate to the right, in accordance with the convention that function application associates to the left; thus, when considering the function *plus*, defined as ($\lambda x.\lambda y.x + y$), the application is written (*plus* 2 3) — for ((*plus* 2) 3) — while its type is written (**Int** → **Int** → **Int**) — for (**Int** → (**Int** → **Int**)); the meaning is that *plus* is a function taking an integer x and returning a function of y into the result. By means of this convention it is simpler to understand *plus* as a function "with two arguments," even when formally it has only one. This is called *currification*; see later.

The typing rules for the functional fragment of the language, originally due to Hindley (1969), are given in Figure 2. Rule (LAM) establishes the typing of functions as explained above. Rule (APP) establishes the typing of applications: When e_1 is a function and e_2 has the type of its argument, then the application ($e_1\,e_2$) has the type of the result. Finally, rule (FIX) establishes the typing for recursive expressions: As (**fix** e) is a fixpoint of e, its type is that of the arguments, or identically, the results, of the function.

Figure 2. Typing system (functional part)

$$\text{(LAM)}\quad \frac{\Gamma_x, x : \tau_2 \vdash e : \tau_1}{\Gamma \vdash \lambda x.e : \tau_2 \to \tau_1}$$

$$\text{(APP)}\quad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

$$\text{(FIX)}\quad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \mathbf{fix}\ e : \tau}$$

The simplest example is that of the function mapping a number to its successor:

$$\dfrac{\dfrac{x : \mathbf{Int} \vdash x : \mathbf{Int} \qquad x : \mathbf{Int} \vdash 1 : \mathbf{Int}}{x : \mathbf{Int} \vdash x + 1 : \mathbf{Int}}}{\vdash \lambda x.x + 1 : \mathbf{Int} \to \mathbf{Int}}$$

The operation of addition forces both the argument *x* and the result to have type **Int**.

A slightly more complex example is that of a function projecting an element from a tuple:

$$\dfrac{\dfrac{\dfrac{p : (\mathbf{Int}, \mathbf{Int}) \vdash p : (\mathbf{Int}, \mathbf{Int})}{p : (\mathbf{Int}, \mathbf{Int}) \vdash \pi_{1,2}\, p : \mathbf{Int}}}{\vdash \lambda p.\ \pi_{1,2}\, p : (\mathbf{Int}, \mathbf{Int}) \to \mathbf{Int}} \qquad \dfrac{\Gamma^{fst} \vdash fst : (\mathbf{Int}, \mathbf{Int}) \to \mathbf{Int} \qquad \dfrac{\Gamma^{fst} \vdash 2 : \mathbf{Int} \qquad \Gamma^{fst} \vdash 5 : \mathbf{Int}}{\Gamma^{fst} \vdash (2,5) : (\mathbf{Int}, \mathbf{Int})}}{\Gamma^{fst} = fst : (\mathbf{Int}, \mathbf{Int}) \to \mathbf{Int} \vdash fst\ (2,5) : \mathbf{Int}}}{\vdash \mathbf{let}\ fst = \lambda p.\ \pi_{1,2}\, p\ \mathbf{in}\ fst\ (2,5)\ : \mathbf{Int}}$$

Observe that the built-in construct $\pi_{1,2}$ that cannot be used alone as a full expression, can be transformed into a function *fst* by means of anonymous functions. The typing of *fst* is chosen as to match the typing of the argument (2,5); this issue is discussed below. As before, the assigning of names to contexts is used to simplify the reading of the derivation.

Another interesting example is a function "with two arguments:" As mentioned before, that is represented by a higher-order function taking one argument and returning another function that completes the work. (This is called *currification*, and the function is said to be *curried*.) Here, the function calculating the maximum of two numbers has been chosen (a similar derivation can be constructed for function *plus* above):

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{(as before)}}{\Gamma^{xy} = x : \mathbf{Int}, y : \mathbf{Int} \vdash \mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y : \mathbf{Int}}}{x : \mathbf{Int} \vdash \lambda y.\ \mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y : \mathbf{Int} \to \mathbf{Int}}}{\vdash \lambda x.\ \lambda y.\ \mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y : \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int}} \qquad \dfrac{\dfrac{\Gamma^{max} \vdash max : \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int} \qquad \Gamma^{max} \vdash 2 : \mathbf{Int}}{\Gamma^{max} \vdash max\ 2 : \mathbf{Int} \to \mathbf{Int}} \qquad \Gamma^{max} \vdash 5 : \mathbf{Int}}{\Gamma^{max} = max : \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int} \vdash max\ 2\ 5 : \mathbf{Int}}}{\vdash \mathbf{let}\ max = \lambda x.\ \lambda y.\ \mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y\ \mathbf{in}\ max\ 2\ 5 \quad : \mathbf{Int}}$$

The derivation of the conditional in the body of the function has already been given in Section "Typing Nonfunctional Elements." It can be noted that the typing of $(\lambda x.\lambda y.\ ...)$ gives $(\mathbf{Int} \to (\mathbf{Int} \to \mathbf{Int}))$, but the notational convention is used. Observe that it is the comparison between x and y which forces their type to be **Int**.

An important example is that of a recursive function; first the function itself is typed:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma^{fv} \vdash v : \mathbf{Int} \quad \Gamma^{fv} \vdash 0 : \mathbf{Int}}{\Gamma^{fv} \vdash v{=}{=}0 : \mathbf{Bool}} \quad \Gamma^{fv} \vdash 1 : \mathbf{Int} \quad \dfrac{\Gamma^{fv} \vdash f : \mathbf{Int} \to \mathbf{Int} \quad \dfrac{\Gamma^{fv} \vdash v : \mathbf{Int} \quad \Gamma^{fv} \vdash 1 : \mathbf{Int}}{\Gamma^{fv} \vdash v - 1 : \mathbf{Int}}}{\Gamma^{fv} \vdash f(v-1) : \mathbf{Int}}}{\Gamma^{fv} \vdash v * f(v-1) : \mathbf{Int}}}{\Gamma^{fv} = f : \mathbf{Int} \to \mathbf{Int},\ v : \mathbf{Int} \vdash \mathbf{if}\ v{=}{=}0\ \mathbf{then}\ 1\ \mathbf{else}\ v * f(v-1) : \mathbf{Int}}}{f : \mathbf{Int} \to \mathbf{Int} \vdash \lambda v.\ \mathbf{if}\ v{=}{=}0\ \mathbf{then}\ 1\ \mathbf{else}\ v * f(v-1) : \mathbf{Int} \to \mathbf{Int}}}{\vdash \lambda f.\ \lambda v.\ \mathbf{if}\ v{=}{=}0\ \mathbf{then}\ 1\ \mathbf{else}\ v * f(v-1) : (\mathbf{Int} \to \mathbf{Int}) \to (\mathbf{Int} \to \mathbf{Int})}}{\vdash \mathbf{fix}\ (\lambda f.\ \lambda v.\ \mathbf{if}\ v{=}{=}0\ \mathbf{then}\ 1\ \mathbf{else}\ v * f(v-1)) : \mathbf{Int} \to \mathbf{Int}}$$

and having the derivation for the function, a complete expression using it can be typed:

$$\dfrac{\dfrac{\text{(as before)}}{\vdash \mathbf{fix}\ (\lambda f.\ \lambda v.\ \mathbf{if}\ v{=}{=}0\ \mathbf{then}\ 1\ \mathbf{else}\ v * f(v-1)) : \mathbf{Int} \to \mathbf{Int}} \quad \dfrac{\dfrac{\Gamma^{fact} \vdash fact : \mathbf{Int} \to \mathbf{Int} \quad \Gamma^{fact} \vdash 2 : \mathbf{Int}}{\Gamma^{fact} \vdash fact\ 2 : \mathbf{Int}} \quad \text{(idem)}}{\Gamma^{fact} \vdash fact\ 5 : \mathbf{Int}}}{\Gamma^{fact} = fact : \mathbf{Int} \to \mathbf{Int} \vdash (fact\ 2, fact\ 5) : (\mathbf{Int}, \mathbf{Int})}$$

$$\vdash \mathbf{let}\ fact = \mathbf{fix}\ (\lambda f.\ \lambda v.\ \mathbf{if}\ v{=}{=}0\ \mathbf{then}\ 1\ \mathbf{else}\ v * f(v-1))\ \mathbf{in}\ (fact\ 2, fact\ 5)$$

Oddly enough, after the addition of functions the type system is no longer functional — that is, there are expressions that admit more than one typing. The simplest example of one of these expressions is the identity function; it can be typed as the identity on numbers:

$$\dfrac{x : \mathbf{Int} \vdash x : \mathbf{Int}}{\vdash \lambda x.x : \mathbf{Int} \to \mathbf{Int}}$$

and it can also be typed as the identity on Booleans:

$$\frac{x : \mathbf{Bool} \vdash x : \mathbf{Bool}}{\vdash \lambda x.x : \mathbf{Bool} \to \mathbf{Bool}}$$

In fact, it can be typed as the identity on any fixed type τ:

$$\frac{x : \tau \vdash x : \tau}{\vdash \lambda x.x : \tau \to \tau}$$

At first sight, the loosing of the functionality is not a very important issue. However, it has a high impact in the implementation of the type inference algorithm. For example, in the following typing derivation, the subderivation for the identity function must produce a function from numbers to numbers:

$$\frac{\dfrac{x : \mathbf{Int} \vdash x : \mathbf{Int}}{\vdash \lambda x.x : \mathbf{Int} \to \mathbf{Int}} \qquad \dfrac{\Gamma^{idI} \vdash id : \mathbf{Int} \to \mathbf{Int} \qquad \Gamma^{idI} \vdash 3 : \mathbf{Int}}{\Gamma^{idI} = id : \mathbf{Int} \to \mathbf{Int} \vdash id\ 3 : \mathbf{Int}}}{\vdash \mathbf{let}\ id = \lambda x.x\ \mathbf{in}\ id\ 3 : \mathbf{Int}}$$

But suppose that the number 3 is changed by the Boolean **True**. Will that mean that the expression has no longer a type? The answer is no. The following derivation can be constructed:

$$\frac{\dfrac{x : \mathbf{Bool} \vdash x : \mathbf{Bool}}{\vdash \lambda x.x : \mathbf{Bool} \to \mathbf{Bool}} \qquad \dfrac{\Gamma^{idB} \vdash id : \mathbf{Bool} \to \mathbf{Bool} \qquad \Gamma^{idB} \vdash \mathbf{True} : \mathbf{Bool}}{\Gamma^{idB} = id : \mathbf{Bool} \to \mathbf{Bool} \vdash id\ \mathbf{True} : \mathbf{Bool}}}{\vdash \mathbf{let}\ id = \lambda x.x\ \mathbf{in}\ id\ \mathbf{True} : \mathbf{Bool}}$$

The typing of the identity function is different in the two examples. However, in a type inference algorithm, when making the recursive call to produce a type for the identity function, there is no information about which type must be chosen. The problem is that the decision about which type to assign to the function's argument *x* is needed inside the derivation of the defining expression, but cannot be taken until the body is processed; however, the type of the defining expression is needed to process the body.

The problem just mentioned is common to many static analysis techniques: The analysis of certain subexpressions may depend on information coming from the context where they are used. In the type system, the information coming from the context is the type of the argument in the rule (LAM). The solution to this problem is to use a mechanism

to defer decisions and another one to take those deferred decisions and communicate them to the rest of the algorithm. Such a solution is presented in the next section.

The Implementation of Type Inference

In the previous section, the specification of a type system has been studied, and the problem regarding the implementation of type inference was stated. In this section, an algorithm is completely designed and implemented. The correctness and completeness of the given algorithm is stated, and although the proof is not given, similar proofs can be found in the literature — for example, Jones (1994a).

This section is divided into two parts. Firstly, the design of the algorithm is presented in the style of proof systems; however, the rules presented here are completely functional, and so they can be read directly as the design of a program. This design, originally introduced by Rèmy (1989), uses two formal mechanisms: variables and substitutions. Additionally, the correctness and completeness of the algorithm are stated as two theorems. Second, an almost direct coding in the language Java is presented, discussing its relation with the given design. More efficient ways to code the algorithm can be given, but the one presented here illustrates better the important features to be taken into account, without the clutter of subtle implementation tricks.

Designing an Algorithm for Type Inference

For the design of the algorithm for type inference, two mechanisms are needed. The first one is the addition of type variables in order to be able to express decisions that are deferred: When a decision has to be made but there is not enough information available, a new type variable is introduced; later, when the information becomes available, the variable can be replaced by the correct information. These new type variables are not part of the language presented to the programmer, but only a tool used inside the implementation (this will change when adding polymorphism). So, for the algorithm, the actual type language used is the following one:

$$\tau ::= t \mid \textbf{Int} \mid \textbf{Bool} \mid (\tau, \ldots, \tau) \mid \tau \rightarrow \tau$$

The new syntactic category t stands for a countably infinite set of type variables (disjoint from the set of term variables). Observe that type variables can appear in any place where a type is needed; however, only ground types (i.e., with no variables) are accepted as result of the algorithm.

The second mechanism needed to design the type inference algorithm is one allowing the recording of decisions taken, in order to propagate them to the rest of the derivation tree. Substitutions are perfect for this task.

- **Definition 3.1:** *A* substitution*, denoted by S, is a function from type variables to types (formally, $S : t \to \tau$), where only a finite number of variables are mapped to something other than themselves.*

The denotation for the application of a substitution S to a variable t is written $(S\ t)$. The substitution mapping every variable to itself — the identity substitution — is denoted **Id**. The substitution mapping variable t to type τ, and all the rest to themselves is denoted $[t := \tau]$, so $[t := \tau]\ t = \tau$, and $[t := \tau]\ t' = t'$ for $t' \neq t$. Similarly, a substitution mapping t_i to type τ_i for $i = 1..n$ is denoted $[t_1 := \tau_1, \ldots, t_n := \tau_n]$.

Substitutions are extended to operate on types in a homomorphic way—that is, every variable t appearing in the type being substituted is replaced by $(S\ t)$. Formally this is denoted as S^*, and defined as:

$$\begin{cases} S^* t = St \\ S^* \text{Int} = \text{Int} \\ S^* \text{Bool} = \text{Bool} \\ S^*(\tau_1, \ldots, \tau_n) = (S^*\tau_1, \ldots, S^*\tau_n) \\ S^*(\tau_1 \to \tau_n) = (S^*\tau_1) \to (S^*\tau_n) \end{cases}$$

By abuse of notation, the form $(S\ \tau)$ is also used for types.

The composition of substitutions is defined using the extension to types; the composition of substitution S with substitution T is denoted $(S\ T)$, and defined as $(S\ T)\ t = S^*\ (T\ t)$. However, by using the abuse of notation, this definition can also be written as $(S\ T)\ t = S\ (T\ t)$, thus resembling the standard notion of function composition.

The final notion regarding substitutions is its extension to operate on typing environments. The idea is that applying a substitution to an environment amounts to applying the substitution to every type appearing in the environment. Thus, if $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, then $S^*\ \Gamma = x_1 : S^*\ \tau_1, \ldots, x_n : S^*\ \tau_n$. Once again, the notation can be abused, thus defining $S\ \Gamma = x_1 : S\ \tau_1, \ldots, x_n : S\ \tau_n$ instead.

Variables are used to defer decisions, and substitutions are used to record a decision that has been taken. But, where are decisions actually taken? In order to formalize that idea, the notion of *unifier* has to be introduced. A unifier for types τ and τ' is a substitution S such that $S\ \tau = S\ \tau'$. If given two types, there exists a unifier for them, the types are said to be *unifiable*.

Unifiers are not unique, but they can be ordered by a more general order: A unifier S for τ and τ' is more general than a unifier T for the same types, denoted $S > T$, if there exist a substitution R such that $T = R\ S$. The composition of substitutions is used in the definition. The idea is that a unifier is more general than another one if it takes less decisions to unify the two types; the substitution R is encoding the remaining decisions taken by T. Given two unifiable types, there always exists a most general unifier, that is,

a unifier that is more general than any other for those types; most general unifiers are not unique, however they differ only in the name of variables. This can be observed in the above-mentioned substitution R: It only maps variables to variables (and it is called a *renaming* for that reason).

For example, taking $\tau = \mathbf{Int} \to t_1 \to t_2$ and $\tau' = \tau_3 \to \mathbf{Bool} \to t_4$, two possible unifiers for τ and τ' are $S_1 = [t_1{:=}\mathbf{Bool}, t_2{:=}t_4, t_3{:=}\mathbf{Int}]$ and $S_2 = [t_1{:=}\mathbf{Bool}, t_2{:=}t_5 \to t_5, t_3{:=}\mathbf{Int}, t_4{:=}t_5 \to t_5]$; in this case, $S_1 > S_2$, being $R = [t_4{:=}t_5 \to t_5]$; it is the case also that S_1 is a most general unifier. Another most general unifier for the two types is $S_3 = [t_1{:=}\mathbf{Bool}, t_2{:=}t_6, t_3{:=}\mathbf{Int}, t_4{:=}t_6]$; observe that $S_1 > S_3$, by virtue of $R_1 = [t_4{:=}t_6]$ and also that $S_3 > S_1$, by virtue of $R_2 = [t_6{:=}t_4]$ — observe that both R_1 and R_2 are renamings.

It may be the case that given two types, there exist no unifier for them. There are two sources for this situation in the language presented here. The first one is common to any language with constants and type constructors: Two expressions constructed with different constructors cannot be unified. For example, there is no unifier for types **Int** and **Bool**, or for types **Bool** and **Int** → **Int**, and so on. The second one is subtler: It is related with the unification of a variable t with a type containing an occurrence of such a variable. For example, there are no unifiers for t_1 and $(t_1 \to \mathbf{Int})$, because the only choice is to substitute t_1 for an infinite type $((\ldots \to \mathbf{Int}) \to \mathbf{Int}) \to \mathbf{Int})$; however this language contains no infinite types. There are languages of types where there is a solution for the unification of these two types. (See the extension of the typing system with recursive types in Section *Recursive Types*.)

The first part of the implementation is to design an algorithm calculating the most general unifier of two given types. This algorithm, called *unification*, is written using judgements of the form $\tau \sim^S \tau'$, meaning that S is a most general unifier for τ and τ'. The rules defining the unification system are given in Figure 3.

Figure 3. Unification algorithm

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

$$\mathbf{Bool} \sim^{\mathbf{Id}} \mathbf{Bool}$$

$$t \sim^{[t:=\tau]} \tau \qquad (t \text{ does not occur in } \tau)$$

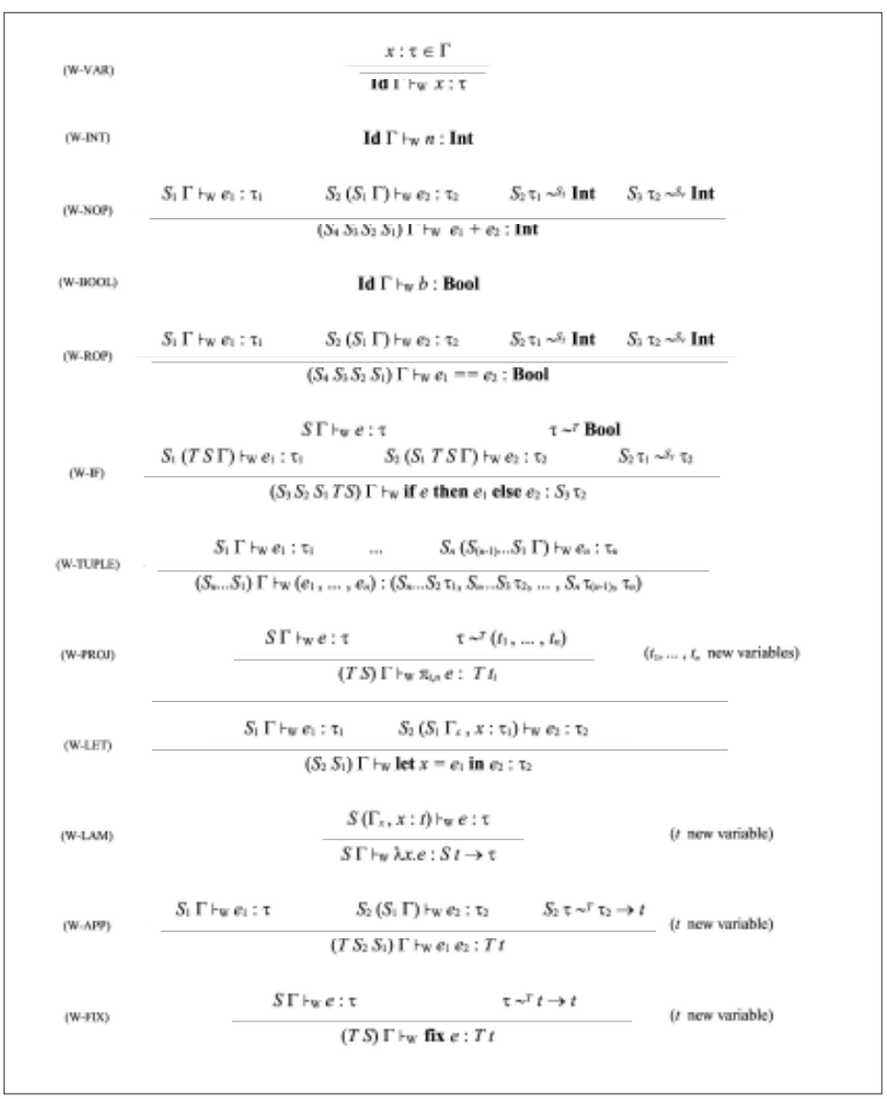
$$\tau \sim^{[t:=\tau]} t \qquad (t \text{ does not occur in } \tau)$$

$$\frac{\tau_1 \sim^{S_1} \tau'_1 \qquad S_1\,\tau_2 \sim^{S_2} S_1\,\tau'_2}{\tau_1 \to \tau_2 \sim^{S_2 S_1} \tau'_1 \to \tau'_2}$$

$$\frac{\tau_1 \sim^{S_1} \tau'_1 \qquad S_1\,\tau_2 \sim^{S_2} S_1\,\tau'_2 \qquad \ldots \qquad S_{(n-1)}\ldots S_1\,\tau_n \sim^{S_n} S_{(n-1)}\ldots S_1\,\tau'_n}{(\tau_1, \ldots, \tau_n) \sim^{S_n \ldots S_1} (\tau'_1, \ldots, \tau'_n)}$$

These rules can be read as function calls: the types are passed to the unification function, and the substitution is returned as the result. Observe, in particular, how in the second premise of the rule for functions, the substitution S_1 produced by the first recursive call is applied to types τ_2 and τ'_2 before making the second recursive call; the reason for this is to propagate decisions that may have been taken in the first call. Correctness is given by the following proposition.

Figure 4. Type inference algorithm



- **Proposition 3.2:** *Types* τ *and* τ' *are unifiable iff* $\tau \sim^S \tau$' *for some substitution S satisfying that* $S\,\tau = S\,\tau$' *and for all T such that* $T\,\tau = T\,\tau$', $S > T$. *That is, the types* τ *and* τ' *are unifiable iff the unification algorithm returns a substitution S that is a most general unifier for* τ *and* τ'.

A coding of the unification algorithm must take into account the possibility of failure, encoding it in an appropriate form — for example, by returning a boolean establishing the success of the operation, and the substitution in case that the boolean is **True**.

The algorithm for type inference is also specified by using a formal system whose rules that can be understood as function calls. The judgments used for this system have the form $S\ \Gamma \vdash_W e : \tau$, where the environment Γ and the expression e are inputs, and the substitution S and the type τ are outputs (the W in the judgement is there for historical purposes. Such was the name used by Milner for his inference algorithm). The rules for this system are given in Figure 4.

The rules look much more complicated than they really are, because of the need to propagate substitutions obtained in each recursive call to subsequent calls. In particular, the notation $S_2\ (S_1\ \Gamma)$ is used to indicate that the environment $(S_1\ \Gamma)$ is used as input, and substitution S_2 is obtained as output. In contrast with $(S_2\ S_1)\ \Gamma$, used to indicate that Γ is the input and the composition $(S_2\ S_1)$, the output (see rules (W-NOP), (W-ROP), (W-IF), (W-TUPLE), (W-LET), and (W-APP)). Another important thing to observe is that no assumption is made about the type returned by a recursive call. Instead, unification is used to make decisions forcing it to have the desired form afterwards (see rules (W-NOP), (W-ROP), (W-IF), (W-PROJ), (W-APP), and (W-FIX)). A final observation is concerned with the side conditions of the form (t new variable). The idea is that t is a variable not used in any other part of the derivation tree. This is a global condition, and any implementation has to take care of it.

As an example, the use of the algorithm to type the identity function is presented:

$$\frac{\mathbf{Id}\ (x : t_x) \vdash_W x : t_x}{\mathbf{Id}\ \varnothing \vdash_W \lambda x.x : t_x \to t_x} \qquad (t_x \text{ new variable})$$

Observe how a type containing variables is returned (together with the identity substitution).

Considering again the two uses of the identity function on numbers and booleans, but using the algorithm to obtain the derivations, it can be seen that the recursive call for typing the identity is always the same, and that it is the unification in rule (W-APP) which gives the variable its right value.

$$\frac{\dfrac{\mathbf{Id}\ (x : t_x) \vdash_W x : t_x \quad \mathbf{Id}\ \Gamma^{id} \vdash_W id : t_x \to t_x \quad \mathbf{Id}\ \Gamma^{id} \vdash_W 3 : \mathbf{Int} \quad t_x \to t_x \sim^{[t_x:=\mathbf{Int},\ t:=\mathbf{Int}]} \mathbf{Int} \to t}{\mathbf{Id}\ \varnothing \vdash_W \lambda x.x : t_x \to t_x \quad [t_x:=\mathbf{Int},\ t:=\mathbf{Int}]\ (\Gamma^{id} = id : t_x \to t_x) \vdash_W id\ 3 : \mathbf{Int} = [t_x:=\mathbf{Int},\ t:=\mathbf{Int}]\ t}}{[t_x:=\mathbf{Int},\ t:=\mathbf{Int}]\ \varnothing \vdash_W \mathbf{let}\ id = \lambda x.x\ \mathbf{in}\ id\ 3 : \mathbf{Int}}$$

The relation of this new system with the old one — stated in Theorems 3.3 and 3.4 — is simple because the original system has a very important property: There is at most one rule for every construct in the language. This property is expressed by saying that the system is *syntax-directed*. Syntax-directed systems are relatively easy to implement because they provide only one rule for every construct. When specifying the extension to polymorphism, this property will be lost, giving much more work to implement that system. (See Section "Adding Parametric Polymorphism.")

The correctness and completeness of the algorithm are stated in the following theorems.

- **Theorem 3.3:** *If* $S\,\Gamma \vdash_W e : \tau$, *then* $(S\,\Gamma) \vdash e : \tau$. *That is, the type returned by the algorithm is a type for e.*

- **Theorem 3.4:** *If* $(S\,\Gamma) \vdash e : \tau$ *for some S, then* $T\,\Gamma \vdash_W e : \tau'$ *for some T and* τ', *and such that there exists R satisfying that* $S = R\,T$, *and* $\tau = R\,\tau'$. *In other words, any type for e can be obtained by a suitable substitution from a type given by the algorithm.*

These two properties, together with the fact that the system $\vdash_W$ is functional (up to the renaming of new variables), state that in order to type a term it is enough to run the algorithm once and later calculate the right substitution on every use.

However, as presented, this system only allows the calculation of the substitution once and for all, instead of at every use.

$$\frac{\mathbf{Id}\ (x : t_x) \vdash_W x : t_x \qquad \mathbf{Id}\ \Gamma^{id} \vdash_W id : t_x \to t_x \qquad \mathbf{Id}\ \Gamma^{id} \vdash_W \mathbf{True} : \mathbf{Bool} \qquad t_x \to t_x \sim^{[t_x:=\mathbf{Bool},\ t:=\mathbf{Bool}]} \mathbf{Bool} \to t}{\mathbf{Id}\ \varnothing \vdash_W \lambda x.x : t_x \to t_x \qquad [t_x:=\mathbf{Bool},\ t:=\mathbf{Bool}]\ (\Gamma^{id} = id : t_x \to t_x) \vdash_W id\ \mathbf{True} : \mathbf{Bool} = [t_x:=\mathbf{Bool},\ t:=\mathbf{Bool}]\ t}$$

$$[t_x:=\mathbf{Bool},\ t:=\mathbf{Bool}]\ \varnothing \vdash_W \mathbf{let}\ id = \lambda x.x\ \mathbf{in}\ id\ \mathbf{True} : \mathbf{Bool}$$

When typing (*id* **True**), the variable t_x from the type of *id* was already substituted by **Int**, and so, the unification with **Bool** fails. For this reason, the term has no type, and is thus rejected by the type system. However, the term has no errors; it is the type system that has not enough expressive power. This kind of problem can be fixed by extending the system with parametric polymorphism. (This system is sketched in Section "Adding Parametric Polymorphism.")

The design of the algorithm is complete. In the following section, a coding of this algorithm using the object-oriented paradigm in the language C++ (Stroustrup, 2000) is presented.

Coding Type Inference with Objects

The presentation of the algorithm given before has been divided into three parts: the representation of terms and types, the unification algorithm, and the type inference

algorithm. In a functional language, where the usual concern is how data is created (Nordlander, 1999), this separation would have been clear. Terms and types would have been represented as algebraic types, and unification and type inference would have been functions on those types. However, the object-oriented paradigm concentrates in how data is used — it being a dual view of programming (Nordlander, 1999) — and thus, the separation is not so clear. Recalling that the goal of this chapter is to bridge the gap between specification and implementation, that the gap is broader in the object-oriented paradigm than in the functional one, and that object-oriented languages are much more widespread, the implementation explained here will be done for the object-oriented paradigm. An implementation using a functional language can be found in Martínez López (in press).

The coding in the object-oriented paradigm uses several common patterns, such as the Composite, Singleton, or Template Methods (Gamma, Helm, Johnson, & Vlissides, 1995), so, for an experienced object-oriented programmer with a thorough understanding of the previous sections, it will not present a challenge.

In order to represent terms and types, the pattern Composite has been used. Abstract classes Term and Type represent those elements, and its subclasses indicate the corresponding alternative; each subclass has internal variables for its parts and the corresponding getters and (internal) setters. This idea is depicted in Figure 5 graphically using class diagrams (Object Management Group, 2004).

Type inference is represented by means of a message to terms, and type unification by means of a message to types. Abstract methods are used to allow each subclass to have the right behaviour by providing a hook method. This is an example of the Template Method Pattern. For unification, the abstract method unify() is redefined by each subclass, providing the right behaviour through hook methods. Additionally, unification uses double dispatching: When a type receives the message of unify with some other type, it sends to it the message unifyWithXX(), where XX is its own class. For example, class Function redefines unify'() to the following:

```
void Function:: unify'(Type *t){
                    t->unifyWithFunction(this);
        }
```

and unifyWithFunction to:

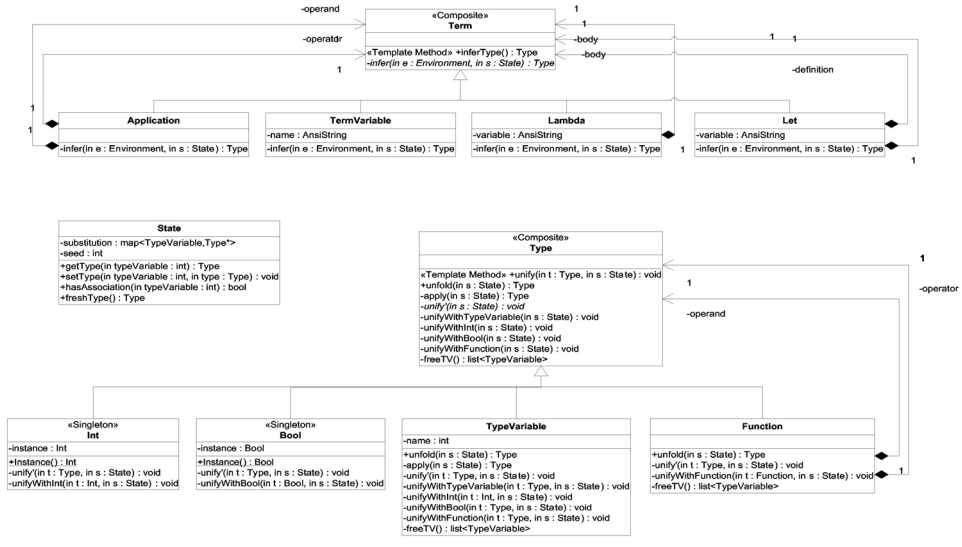
```
void Function:: unifyWithFunction(Function *t){
                    this->getOperator()->unify  (t->getOperator());
                    this->getOperand()->unify(t->getOperand());
        }
```

where getOperator() and getOperand() are the getters of Function's components.

Regarding type inference, the important coding decision is to represent the substitution being calculated as a global element. This is obtained by the instance variable state of class GlobalState in class Term. Instances of GlobalState contain two components:

- the current substitution, represented as a mapping between TypeVariables and Types, with operations setType() to associate a variable with a type, hasAssociation()

Figure 5. Class diagram for type inference in the object-oriented paradigm



informing if a given variable has an association, and getType() returning the type associated to a variable if it exists, and

- a seed, with an operation freshType() used to calculate a new type variable (one never used before in the derivation; thus its global nature) in those rules that require it.

Instead of performing the application of the substitution to every element immediately, the substitution is recorded in the state, and applied only when the type has to be accessed. The coding of this behaviour is given by the methods apply() and unfold() of class Type: apply() represents the application of a substitution to a type variable (i.e., it stops the application of the substitution as soon as something different from a variable is found), while unfold() represents the application of a substitution to a type (i.e., it applies the substitution to all the parts of the type, recursively). The difference between apply() and unfold() is due to the behaviour of unification. When unification is performed, new associations between type variables and types (corresponding to the decisions taken by the algorithm) are made on every subterm, and those new associations would have not been taken into account by unfold(); instead, apply() is used at every subterm, performing an incremental unfolding that has taken into account the last associations calculated. The code of apply() and unfold() is identical at class TypeVariable, but differs in Function.

```
type* Type:: unfold(State s){
        return this;
}
Type* TypeVariable:: unfold(State s){
        if (!s->hasAssociation(this->getName)) return this;
        else return (s->getType(this->getName())->unfold(s));
}
Type* Function:: unfold(State s){
        this->setOperand(this->getOperand()->unfold(s));
        this->setOperator(this->getOperator()->unfold(s));
        return this;
}
Type* Type:: apply(State s){
        return this;
}
Type* TypeVariable:: apply(State s){
        if (!s->hasAssociation(this->getName)) return this;
        else return (s->getType(this->getName())->apply(s));
}
```

The coding of method inferType in class Term just resets the global state with a new state (with the identity substitution and a base seed), and then it calls the abstract method infer, letting the subclasses provide the actual behaviour through hook methods. Environments used in the internal method infer are represented as mappings from TermVariables to Types.

```
Type* Term:: inferType(){
        State* state = new State;
        state->initialState();
        Environment* g = new Environment;
        Type* t = this->infer(g,state)->unfold(state);
        delete g;
        return t;
}
```

Adding Parametric Polymorphism

In the last example of the section "Designing an Algorithm for Type Inference" a term having no type, but without errors, has been given. That example shows the limitations of the simple type system presented. In this section, the basic type system is extended with parametric polymorphism, making it more powerful, and so, capable of typing terms like the one presented in the example.

Recall that the problem was that the simple type system only allows the calculation of the substitution once and for all, instead of at every use. In order to solve the problem, some mechanism allowing multiple different instances for a given variable is needed. The solution is to quantify universally a type with variables, thus indicating that the

quantified variables can take many different values. However, this has to be done with care. If universal quantification is added in the same syntactic level as the rest of the typing constructs, the system obtained becomes undecidable (Wells, 1994, 1998). In order to have polymorphism keeping decidability, Damas and Milner separate the quantification in a different level (Damas & Milner, 1982), thus having (monomorphic) types in one level, and type schemes (polymorphic types) in another one. The type system is designed in such a way that only declarations made in a **let** construct are allowed to be polymorphic. For that reason, it is sometimes called let-bounded polymorphism. Another solution is to add type annotations into terms (Reynolds, 1998), resulting in the language called *second order lambda calculus*, but that alternative is not described here. The term language is the same, but the type language is changed.

- **Definition 4.1:** *Types for a let-bounded polymorphic language are given by two syntactic categories: monomorphic types, called also* monotypes*, denoted by* τ*, and polymorphic types, called* type schemes*, denoted by* σ*. The grammars defining these categories are the following:*

$$\tau ::= t \mid \mathbf{Int} \mid \mathbf{Bool} \mid (\tau, \ldots, \tau) \mid \tau \to \tau$$
$$\sigma ::= \tau \mid t.\sigma$$

Observe how monotypes can be seen as type schemes, but not the other way round. Once a quantification has been introduced, no other type constructions can be used. This forces the elements of tuples and the arguments and results of functions to be monomorphic (they can be made polymorphic in an outer level, but, for example, no polymorphic function can be passed as argument. See the example at the end of this section). The addition of quantification to types implies that the variables of a type expression are now divided into two groups: those that are quantified (*bound* variables) and those that are not (*free* variables). This notion is exactly the same as the one of free variables in the term language, where is the binder and the s following it, its scope.

The judgements to determine the typing of terms in the polymorphic system are very similar to those of the simply typed system. The only differences are that environments associate variables with type schemes (instead of monotypes as before), and that the resulting type can be a type scheme. That is, the form of judgements is $\Gamma \vdash e : \sigma$, where Γ contains pairs of the form $x : \sigma$. The rules specifying the polymorphic typing are given in Figure 6 (rules (P-INT), (P-NOP), (P-BOOL), and (P-ROP) are omitted because, being monomorphic in nature, they are identical to their counterpart in the simply typed system).

The formulation of this system deserves explanation, because at first glance, it may seem almost identical to the one given in the previous sections. However, the changes make it more powerful. On the one hand, there are two new rules ((P-GEN) and (P-INST)) to allow the quantification of type variables and their removal (by using substitutions). On the other hand, some rules have been changed to incorporate the notion of type scheme ((P-VAR), (P-IF), and, most notably, (P-LET)), while others ((P-TUPLE), (P-PROJ), (P-LAM), (P-APP), and (P-FIX)) remained exactly as before. This fact is represented by the use of

τs and σs in the appropriate places. It is very important that these rules remain unchanged; the reason is that, as functions and tuples cannot handle polymorphic components, their typing must remain monomorphic, and thus the rules are identical to their simply typed counterparts (remember that allowing functions to have polymorphic components turns the system undecidable). The other rules are changed to consider type schemes; the most important of these is (P-LET), because it is the one allowing the actual use of polymorphism. See the following example, considering again the typing of the identity function used twice on different types.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : t_x \vdash x : t_x}\,\textbf{(P-INST)}}{\Gamma^{id} \vdash id : \textbf{Int} \to \textbf{Int}} \quad \Gamma^{id} \vdash 3 : \textbf{Int}}{\Gamma^{id} \vdash id\ 3 : \textbf{Int}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma^{id} \vdash id : \forall t_x.t_x \to t_x}\,\textbf{(P-INST)}}{\Gamma^{id} \vdash id : \textbf{Bool} \to \textbf{Bool}} \quad \Gamma^{id} \vdash \textbf{True} : \textbf{Bool}}{\Gamma^{id} \vdash id\ \textbf{True} : \textbf{Bool}}}{\vdash \lambda x.x : t_x \to t_x}\,\textbf{(P-GEN)}}{\vdash \lambda x.x : \forall t_x.t_x \to t_x} \quad \Gamma'^{id} = id : \forall t_x.t_x \to t_x \vdash (id\ 3, id\ \textbf{True}) : (\textbf{Int}, \textbf{Bool})$$

$$\vdash \textbf{let } id = \lambda x.x \textbf{ in } (id\ 3, id\ \text{True}) : (\textbf{Int}, \textbf{Bool})$$

Observe how the use of the new rules (marked with their names in the derivation) allows the two subderivations to use different types for the identity. This example, that was not typable under the simply typed system, shows that this extension gives more power.

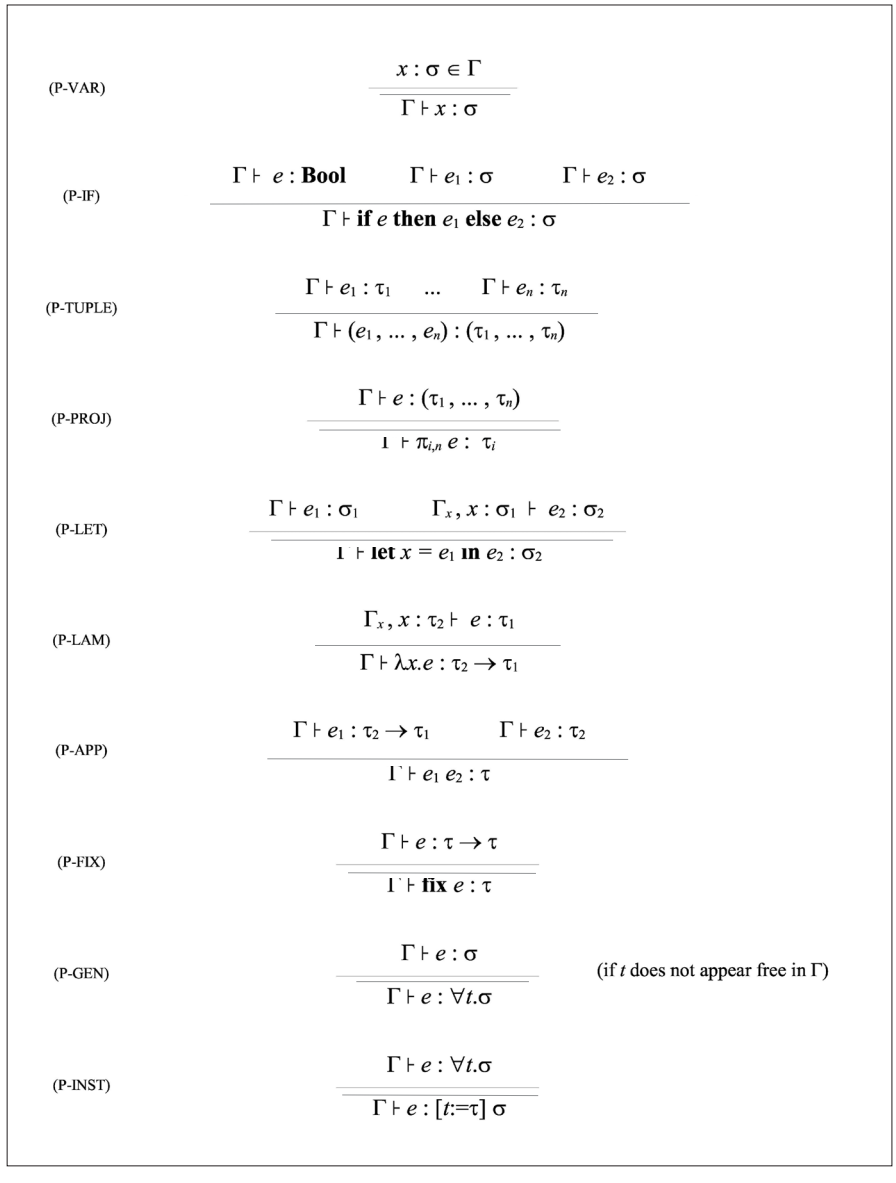
An important thing to observe is that only let-bounded variables are polymorphic. In order to see this, consider the operationally equivalent term $(\lambda id.\ (id\ 3, id\ \textbf{True}))(\lambda x.x)$. This term is not typable on this system. The problem is the use of an anonymous function to bind the variable *id* to the identity function $(\lambda x.x)$; it has to be monomorphic. Observe a possible derivation:

$$\dfrac{\dfrac{\dfrac{}{\textbf{Id}\ (x : t_x) \vdash_{\text{W}} x : t_x}}{\textbf{Id}\ \varnothing \vdash_{\text{W}} \lambda x.x : t_x \to t_x} \quad \dfrac{\dfrac{\textbf{Id}\ \Gamma^{id} \vdash_{\text{W}} id : t_x \to t_x \quad \textbf{Id}\ \Gamma^{id} \vdash_{\text{W}} 3 : \textbf{Int}}{t_x \to t_x \sim^{S=[t_x:=\textbf{Int},\ t_1:=\textbf{Int}]} \textbf{Int} \to t_1}}{S\,\Gamma^{id} \vdash_{\text{W}} id\ 3 : \textbf{Int} = S\,t_1} \quad \dfrac{S\,\Gamma^{id} \vdash_{\text{W}} id : \textbf{Int} \to \textbf{Int} \quad S\,\Gamma^{id} \vdash_{\text{W}} \textbf{True} : \textbf{Bool}}{\textbf{Int} \to \textbf{Int} \not\sim \textbf{Bool} \to t_2\ \textbf{!!!}}}{?? \ (S\,\Gamma^{id}) \not\vdash_{\text{W}} id\ \textbf{True} : ??} \quad (\Gamma^{id} = id : t_x \to t_x) \not\vdash_{\text{W}} (id\ 3, id\ \textbf{True}) : ??$$

$$?? \ \varnothing \not\vdash_{\text{W}} \textbf{let } id = \lambda x.x \textbf{ in } (id\ 3, id\ \text{True}) : ??$$

The implementation of this system requires some care. The problem is that the system is not syntax directed — that is, more than one rule can be applied to type a given term. However, it can be noted that this is the result of the addition of rules (P-GEN) and (P-INST), so, if the use of these two rules can be limited to certain parts of a derivation, they can be incorporated to the other rules. Fortunately, this is the case: The only meaningful uses of (P-GEN) are either before the use of (P-LET), or at the end of a derivation; and the only meaningful uses of (P-INST) are just after the use of (P-VAR). All the other uses can be either removed or changed into one of those mentioned.

Figure 6. Typing system for let-bounded polymorphism



There are two mechanisms to incorporate the uses of these rules to a syntax-directed version: one to capture (P-GEN), and the other for (P-INST). The first one is an operation to quantify exactly those variables that are important.

- **Definition 4.2:** *Let* $A = \{ t_1, ..., t_n \}$ *be the set of free variables of type* τ*, from which the free variables appearing in* Γ *have been removed. Then* $Gen_\Gamma(\tau) = \forall t_1. ... \forall t_n.\tau$. *That is, the operation Gen quantifies all the free variables of a type except those that appear free in the environment.*

The correspondence of this notion with the use of (P-GEN) is given in the following proposition:

- **Proposition 4.3:** *If* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash e : Gen_\Gamma(\tau)$*, and both derivations differ only in the use of rule* (P-GEN).

The second mechanism is a relationship between type schemes, according to how general they are. In order to define it, the notion of generic instance of a type scheme has to be defined first.

- **Definition 4.4:** *A type* τ' *is a* generic instance *of a type scheme* $\sigma = \forall t_1. ... \forall t_n.\tau$ *if there are types* $\tau_1, ..., \tau_n$ *such that* $[t_1:=\tau_1, ..., t_n:=\tau_n]\ \tau = \tau'$.

For example, $\mathbf{Int} \to \mathbf{Int}$ is a generic instance of $\forall t.t \to t$, and $(t, t') \to \mathbf{Bool}$ is a generic instance of $\forall t_1.\forall t_2.t_1 \to t_2$.

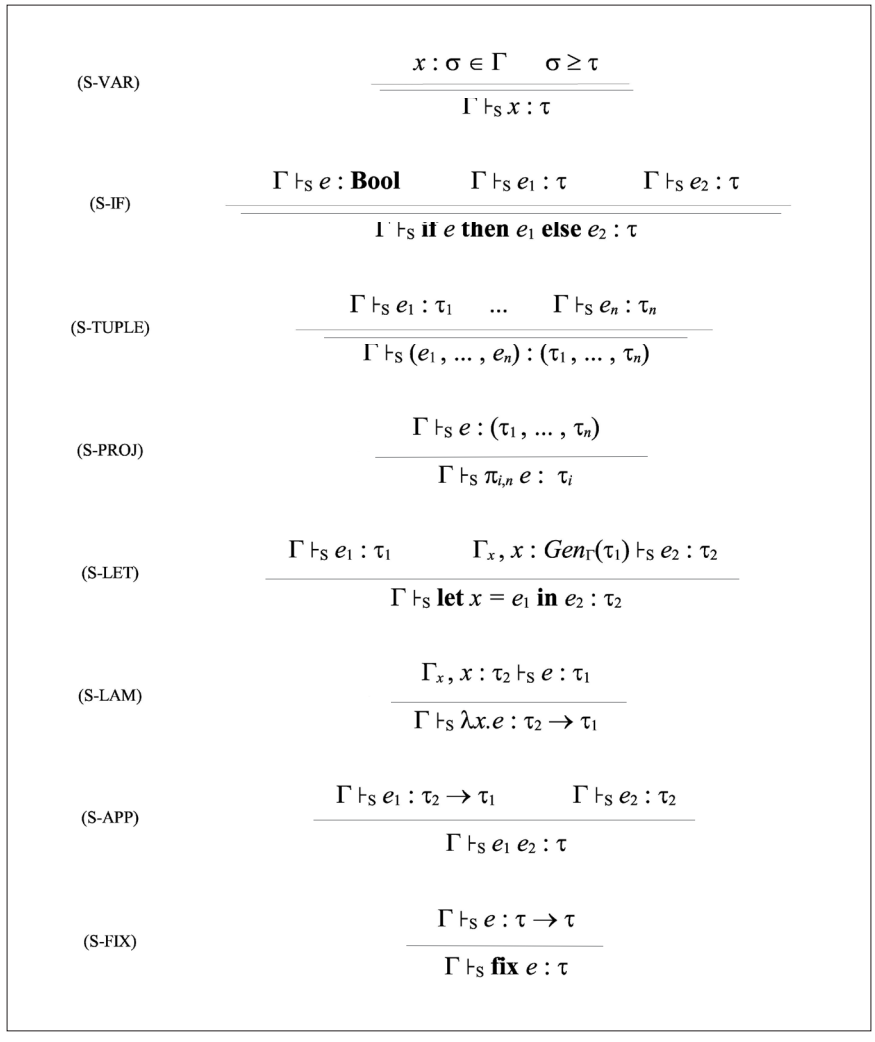
- **Definition 4.5:** *A type scheme* σ *is said to be* more general *than a type scheme* σ'*, written* $\sigma \geq \sigma'$*, if every generic instance of* σ' *is a generic instance of* σ.

For example, $\forall t_1.\ \ t_2.t_1 \to t_2$ is more general than $\forall t.\forall t'.\forall t''.(t, t') \to \mathbf{Bool}$. Observe that there can be any number of quantifiers on the less general type scheme, as long as the variables do not clash with those in the more general one.

This definition of *"more general"* does not provide an easy way to calculate when two types are related. The following proposition provides a characterization giving a syntactic way to do that:

- **Proposition 4.6:** *Let* $\sigma = \forall t_1. ... \forall t_n\tau$*, and* $\sigma' = \forall t'_1. ... \forall t'_m\tau'$*. and suppose none of the* t'_i *appears free in* σ*. Then* $\sigma \geq \sigma'$ *if, and only if, there are types* $\tau_1, ..., \tau_n$ *such that* $[t_1:=\tau_1, ..., t_n:=\tau_n]\ \tau = \tau'$.

Figure 7. Syntax directed system for let-bounded polymorphism



By means of Proposition 4.6, the relationship between two type schemes can be calculated by pattern matching — a restricted form of unification — of τ against τ'.

With all these elements, a syntax-directed version of the polymorphic type system can be introduced. The judgements have the form $\Gamma \vdash_S e : \tau$, where Γ contains pairs of the form $x : \sigma$, but the result is a monotype. The rules are given in Figure 7.

Observe the use of the operation *Gen* in the rule (S-LET), and the use of $\geq$ in the rule (S-VAR). They capture the uses of rules (GEN) and (INST). Correctness and completeness of the syntax-directed version with respect to the specification are given in the following two theorems. Completeness is a bit more complicated because the two

systems return elements of different syntactic categories; the specification returns a type scheme, and the syntax directed version, a monotype. However, they can be related by using *"more general."*

- **Theorem 4.7:** *If* $\Gamma \vdash_S e : \tau$, *then* $\Gamma \vdash e : \tau$. *That is, the type returned by the syntax-directed version is a type for e according to the specification.*

- **Theorem 4.8:** *If* $\Gamma \vdash : \sigma$, *then* $\Gamma \vdash_S e : \tau$ *for some* τ, *and* $Gen_\Gamma(\tau) \geq \sigma$. *In other words, any type scheme for e can be obtained by instantiation of the generalization of a type given by the syntax-directed system.*

Consider a derivation of a type of the identity function using the syntax directed system:

$$\frac{x : t_x \vdash_S x : t_x}{\vdash_S \lambda x.x : t_x \to t_x}$$

Its generalization is $\forall t_x.t_x \to t_x$, and any other type scheme obtainable for the identity function (e.g., $\forall t.(t,t) \to (t,t)$, $\forall t'.\forall t''.(t' \to t'') \to (t' \to t'')$, $\mathbf{Int} \to \mathbf{Int}$, etc.) can be obtained from the former by instantiation.

The algorithm to calculate a type scheme for a given term in the polymorphic system is very similar to the one given for the simply typed system. The most interesting rule is that of variables, because it has to implement the instantiation prescribed by the "more general" relation. This is obtained by substituting the quantified variables for new ones on *every use* of the rule, thus having the desired effect. All the remaining rules are easily deducible.

$$\text{(WP-VAR)} \qquad \frac{x : \forall t_1. \ldots \forall t_n.\tau \in \Gamma}{\mathbf{Id}\ \Gamma \vdash_{WP} x : [t_1 := t'_1, \ldots, t_n := t'_n]\ \tau} \qquad (t'_1, \ldots, t'_n \text{ new variables})$$

The coding of this algorithm is an easy extension of the one presented in Section "Coding Type Inference with Objects." The only possible complication is the representation of bound variables because free and bound variables can be mixed, and thus can cause confusion. A good solution is to have two different representations for free and bound variables and code the algorithms for generalization and instantiation accordingly.

Other Extensions to the Basic Type System

Type systems have been designed for almost all kinds of paradigms in programming, and for almost all features — for example, records, arrays, variant types, not to mention extensions to the language, overloading (Jones, 1994a, 1996; Odersky, Wadler, & Wehr, 1995), recursive types, type systems for imperative features (Hoang, Mitchell, & Viswanathan, 1993; Wright, 1992, 1995), object oriented languages (Abadi & Cardelli, 1996; Nordlander, 1999; Nordlander, 2000; von Oheimb & Nipkow, 1999; Syme, 1999; Wand, 1987), inheritance (Breazu-Tannen, Coquand, Gunter, & Scedrov, 1991; Wand, 1991, 1994), persistency (Atkinson, Bailey, Chisholm, Cockshott, & Morrison, 1983; Atkinson & Morrison, 1988; Dearle, 1988; Morrison, Connor, Cutts, Kirby, & Stemple, 1993), abstract datatypes (Mitchell & Plotkin, 1988), reactive systems (Hughes, Pareto, & Sabry, 1996; Nordlander, 1999), mobile systems (Freund & Mitchell, 1998; Igarashi & Kobayashi, 2001; Knabe, 1995), and others (Pottier, 2000; Shields & Meijer, 2001) in the field of language features.

There are also other ways to present type inference systems, the main alternative being the use of constraints (Aiken, 1999; Sulzmann, Odersky, & Wehr, 1997). In these systems, there are two phases for type inference: the first one, when the term is traversed and constraints on type variables are collected, and a second phase of constraint solving, where a solution for all the constraints gathered is calculated. This form of presentation allows much more flexibility for the use of heuristics during constraint solving, and it is more suitable when typing extensions to the basic system presented here. The theory of qualified types for overloading and the framework of type specialization discussed below use this variation.

In this section, some extensions will be presented. In the first part, basic extensions to the language are considered. In the second part, extensions to the type system to consider advanced features are covered. And in the last one, different type-based analyses are discussed.

Extending the Basic Language

Two important extensions a good language has to consider are the ability to manage records and variant types (or sum types).

Records

Most languages provide records instead of tuples. Records are a variation of tuples with named components. Instead of referencing the elements by their position, there are labels that can be used to retrieve them. The projections $\pi_{i,n}\, e$ are then replaced by the accessing operation $e.l_i$.

- **Definition 5.1:** *The term language is extended with* records *in the following way:*

  e ::= ... | $\{l{=}e, \ldots, l{=}e\}$ | $e.l$ | **with** $\{l = x, \ldots, l = x\} = e$ **in** e

  where l is an element of a set of identifiers called labels.

Examples of record types are $\{x=0, y=0\}$, to represent the origin vector in a two dimension space, $\{$*name* = "Juan Pérez", *age* = 55, *address* = "C.Pellegrini 2010"$\}$ to represent a person, and $\{$*subst* = *idSubst*, *seed* = 0$\}$ to represent a global state (with *idSubst* a proper representation of the identity substitution). Different languages have different rules regarding the scope of labels and whether records can be "incomplete" or not (that is, having some labels with an undefined value or not). For example, in Pascal, any record can contain any label and records may be incomplete, while in O'Haskell (Nordlander, 1999), labels are global constants — thus contained by at most one record type. O'Haskell uses subtyping, thus alleviating the problems that this "global policy" may produce (Nordlander, 2000) — and records have to determine uniquely a record type already declared.

In order to type records, the language of types has to be extended. There are two common choices for doing this: to have named records, and using the name as the type, or to have anonymous records, using the structure as the type. Here, the second option is taken, because it is more illustrative and the other version can be easily reconstructed.

- **Definition 5.2:** *The extension of the type language to consider records is the following:*

  τ ::= ... | **Struct**$\{l : \tau, \ldots, l : \tau\}$

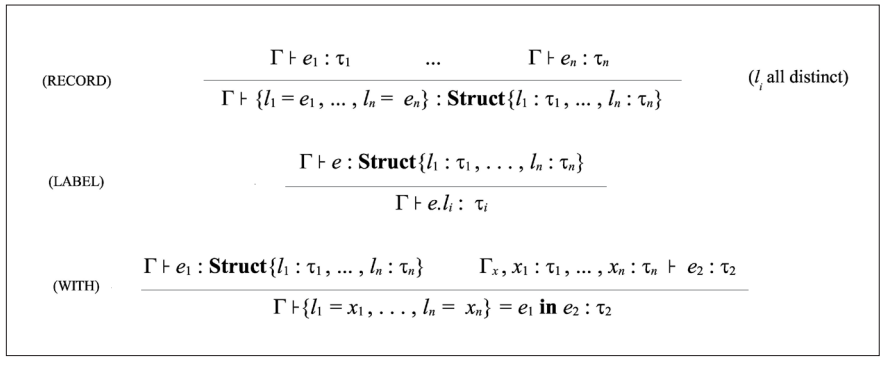
Examples of record types are **Struct**$\{x$: **Int**, y : **Int**$\}$, to represent a vector of two integers, **Struct**$\{$*name* : **String**, *age* : **Int**, *address* : **String**$\}$ to represent the type of a person, and **Struct**$\{$*subst* : **Int** $\rightarrow$ **Type**, *seed* : **Int**$\}$ to represent the type of a global state (for a suitable representation of **Type**s). A common abbreviation used to simplify the writing of record types is to group several labels with the same type by putting them separated by commas before the type, as in **Struct**$\{x, y$: **Int**$\}$, but this is just syntactic sugar.

The extension of the simply typed system to type records is given in Figure 8.

In addition to the accessing operation using a label name, some languages provide a **with** construct, to bind all the elements of a record simultaneously. This construct is similar to the **let** used before. In some languages, the binding of variables to label is left implicit — for example, Pascal — with the consequence that scoping depends thus on typechecking, causing a difficulty to find errors due to hidden variable clashes.

The implementation of type inference for this extension is an easy exercise.

Figure 8. Typing system (records)



The combination of records and higher-order functions has been used to provide objects without classes in the language O'Haskell (Nordlander, 1999). This is an example of the power that a good discipline of types providing neat combinations of different elements can achieve.

Sum Types (Unions, Variants, or Tagged Types)

Sum types are important types for expressiveness, although they are less known than the rest of the type constructs. They consist of the union of several types, discriminating from which addend a given value comes from; the usual way to discriminate is by means of *tags*, and for that reason they also receive the name of *tagged types*. Another way in which they appear in programming languages is as variant records, although that is only a weak implementation of the real notion of sums. The main purpose of variant records is to save memory and not to add expressiveness.

There are several ways to present sum types. The most simple one consists of allowing several named tags — called *constructors* — with a single argument each; constructors with more arguments can be expressed using tuples, and a special type called **Unit** or **Void** needed to have constant constructors. Other versions allow for constructors with multiple arguments, to use unnamed tags (using indexes instead of constructors), to have only binary sums (multiple sums are obtained by combination), and so on.

To start the presentation, the **Unit** type is given first. This type contains a single value, usually called **unit**, and it can be identified with a tuple with zero elements (so, following the notation used for tuples, both the type and its single element can be denoted by a pair of parenthesis — **()**). Another common notation for this type is to call it **void**, and its single element also **void** (for example, in C). The purpose of **unit** is to indicate that some uninteresting or unnecessary argument or result is being used.

- **Definition 5.3:** *The term language is extended with* **unit** *in the following way:*

  e ::= ... | ()

- **Definition 5.4:** *The extension of the type language to consider the* **Unit** *type is the following:*

  t ::= ... | ()

There is only one typing rule for **unit**, presented in Figure 9, with the rules for sum types.

As a simple example of the use of unit, consider its use as filler:

let $fst = \lambda p.\ \pi_{1,2}\ p$
in $(fst\ (2,()), fst\ (\mathbf{True},())$

Here, the type of the first instance of *fst* used in the body of the local definition is (**Int**, ()) → **Int**, and that of the second one is (**Bool**, ()) → **Bool**. More involved examples are given in conjunction with sums.

Sum types are built with constructors — a special kind of function — and they are inspected with a construct called **case**; they can be thought of as a generalization of the notion of booleans, with constructors in place of **True** and **False**, and **case** in place of **if-then-else**. The syntax used here for sum types is given as follows:

- **Definition 5.5:** *The term language is extended with* sums *in the following way:*

  e ::= ... | $C\ e$ | **case** e **of** $C\ x\ .\ e$; ...; $C\ x\ .\ e$

  where C is an element of a set of identifiers called constructors.

The simplest example of sum types is an enumerative type. For example, to model colors, constructors with the names of the colors can be used — as in **Red** (), **Blue** (), and so on. Constructors need an argument, but as colors are supposed to be constants, they are given the meaningless unit value.

Sum types used to type expressions using constructors respect the following syntax:

- **Definition 5.6:** *The extension of the type language to consider sum types is the following:*

  τ ::= ... | **Variant**$\{C : \tau, ..., C : \tau\}$

This presentation provides anonymous sums, that is, they are represented by the possible constructors used to build elements of the type. Different languages made different decisions regarding this choice. The most used one is to assign a name to a sum type, and then use the name when referring to it. The most important difference between these alternatives is how the equivalence of two given types is calculated: They can be considered equal if they have the same set of constructors, regardless of their order — in one extreme — to being considered equal only if they have the same name — in the other one.

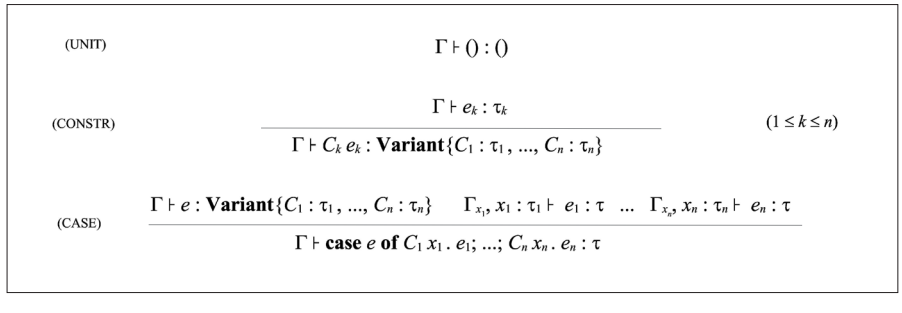
In the case of the modelling of colors, their type can be, for example, **Variant**{**Red**:(), **Yellow**:(), **Blue**:()}, if only primary colors are modelled, or **Variant**{**Red**:(), **Yellow**:(), **Blue**:(), **Orange**:(), **Green**:(), **Purple**:()}, if both primary and secondary colors are modelled.

A slightly more complex example is that of basic geometric shapes. They can be represented as the following type: **Variant**{**Circle** : **Int**, **Rectangle** : (**Int**,**Int**)}, and some of its elements can be given as **Circle** 10, **Rectangle** (3,5), and so on.

The typing rules for expressions involving constructors are given in Figure 9. In the case of rule (CONSTR), there is some freedom in the choice of the constructors not mentioned in the term; some variants regarding the possible accepted constructors there are possible, and the choices are given by the restrictions in the equivalence of types, as described. In the rule (CASE), this presentation forces the construct **case** to have one branch for every constructor appearing in the type; however, other choices are also possible, allowing less constructors to be used. Again, this issue is related with type equivalence between different sums.

By using the case construct, functions over sums can be constructed. For example, the following function calculates the area of a given shape (although in the given definition of the language there are only integers, this example uses operations on floating point numbers, as the decimal number 3.14 as an approximation to number p, and the division by two):

Figure 9. Typing system (unit and sum types)



let *area* = λ*s*. **case** *s* **of Circle** *x*. 3.14*(*x*^2);
Rectangle *xy*. ($\pi_{1,2}$ *xy* * $\pi_{2,2}$ *xy*)/2
in (*area* (**Circle** 5), *area* (**Rectangle** (2,4)))

The type of function *area* in this example is **Variant**{**Circle** : **Int**, **Rectangle** : (**Int**,**Int**)} → **Int**.

Sums can be combined with type variables, to obtain very flexible types. A classic example is that of a variant type used to model the possibility of failure; several names for constructors can be used. Here, they are called **Error** and **Ok**. Thus, the type is **Variant**{ **Error** : **String**, **Ok** : τ}, for some τ. This type can be used to model the result of a function calculating the minimum of a sequence of values, where **Error** "Empty sequence" can be used to model when there are no values, and **Ok** *v* to model that *v* is the minimum.

The real power of sums can be seen when combined with other types, as the case of recursive types, presented in the following section.

Extending the Type System

Some important extensions are better considered extensions to the type system, rather than extensions to the language. Among those, recursive and qualified types are considered here.

Recursive Types

Recursive types are a powerful addition to a type system. Several different ways to introduce recursion with types exist, being the most common one to use recursive definition of types (in combination with named types). However, the formal presentation of recursive types using names and equations is more complex than the anonymous one, thus, as the goal here is to keep simplicity, the latter have been chosen.

The introduction of recursive types usually do not introduce new expressions on the term language, although the presentation is more simple introducing two constructs as follows:

- **Definition 5.7:** *The term language is extended with two new constructs in the following way:*

  e ::= ... | **in** *e* | **out** *e*

These two constructs, **in** and **out**, are usually implicitly assumed in practical languages. Their purpose is to map a recursive type into its expansion, and back.

The syntax of recursive types is obtained with an operator similar to **fix**: it maps a type function into its fixpoint. For historical purposes, this operator is denoted with the greek letter μ. The syntax of types is thus extended as follows:

- **Definition 5.8:** *The extension of the type language to consider recursive types is the following (in addition to type variables, as described in Section "Designing an Algorithm for Type Inference"):*

$$\tau ::= \ldots \mid \mu t.\tau$$

The intended meaning of the type expression $\mu t.\tau$ is the solution of the equation $t = \tau$, where variable t may occur in τ. A simple way to obtain that is to make $\mu t.\tau$ and $[t:=\mu t.\tau]\tau$ isomorphic (i.e., having the same structure); the result of this is that, assuming that $\mu t.\tau$ represents the solution of the equation, replacing on every occurrence of t in τ must give the same result. The term constructs **in** and **out** are functions from $[t:=\mu t.\tau]\tau$ to $\mu t.\tau$, and from $\mu t.\tau$ to $[t:=\mu t.\tau]\tau$, respectively. This is captured by the typing rules given in Figure 10.

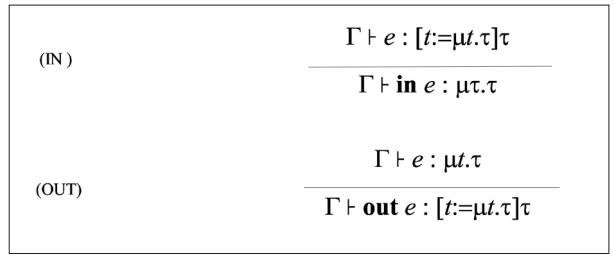
Recursive types provide infinite expansion of type constructs. This feature allows, in particular, extending the solutions to unification. When unifying a type variable t with a type containing it, say $t \rightarrow \mathbf{Int}$, the type $\mu t.\ t \rightarrow \mathbf{Int}$ is a solution for it in this system; observe that the expansion of $\mu t.\ t \rightarrow \mathbf{Int}$ is isomorphic to it, being that the condition needed.

The real power of recursive types comes from the combination with other type constructs, as sums and tuples. A classic example is that of lists of a type t_x; they can be represented as the type $\mu t_l.\mathbf{Variant}\{\ \mathbf{Nil} : (), \mathbf{Cons} : (t_x, t_l)\ \}$. For example, the elements of this type when t_x is **Int** are $(\mathbf{in}\ (\mathbf{Nil}\ ()))$, $(\mathbf{in}\ (\mathbf{Cons}\ (1, \mathbf{in}\ (\mathbf{Nil}\ ()))))$, $(\mathbf{in}\ (\mathbf{Cons}\ (2, \mathbf{in}\ (\mathbf{Cons}\ (1, \mathbf{in}\ (\mathbf{Nil}\ ()))))))$, etc., as the following derivation shows:

$$\dfrac{\dfrac{\dfrac{\vdash () : ()}{\vdash \mathbf{Nil}\ () : \mathbf{Variant}\{\ \mathbf{Nil} : (), \mathbf{Cons} : (\mathbf{Int}, \mu t_l.\ \mathbf{Variant}\{\ \mathbf{Nil} : (), \mathbf{Cons} : (\mathbf{Int}, t_l)\ \})\ \}}}{\vdash \mathbf{in}\ (\mathbf{Nil}\ ()) : \mu t_l.\mathbf{Variant}\{\ \mathbf{Nil} : (), \mathbf{Cons} : (\mathbf{Int}, t_l)\ \}}}{\vdash \mathbf{Cons}\ (1, \mathbf{in}\ (\mathbf{Nil}\ ())) : \mathbf{Variant}\{\ \mathbf{Nil} : (), \mathbf{Cons} : (\mathbf{Int}, \mu t_l.\mathbf{Variant}\{\ \mathbf{Nil} : (), \mathbf{Cons} : (\mathbf{Int}, t_l)\ \})\ \}}$$

$$\dfrac{\dfrac{\vdash 1 : \mathbf{Int} \qquad \dfrac{\text{(as before)}}{\vdash \mathbf{in}\ (\mathbf{Nil}\ ()) : \mu t_l.\mathbf{Variant}\{\ \mathbf{Nil} : (), \mathbf{Cons} : (\mathbf{Int}, t_l)\ \}}}{\vdash \mathbf{Cons}\ (1, \mathbf{in}\ (\mathbf{Nil}\ ())) : \mathbf{Variant}\{\ \mathbf{Nil} : (), \mathbf{Cons} : (\mathbf{Int}, \mu t_l.\mathbf{Variant}\{\ \mathbf{Nil} : (), \mathbf{Cons} : (\mathbf{Int}, t_l)\ \})\ \}}}{\vdash \mathbf{in}\ (\mathbf{Cons}\ (1, \mathbf{in}\ (\mathbf{Nil}\ ()))) : \mu t_l.\mathbf{Variant}\{\ \mathbf{Nil} : (), \mathbf{Cons} : (\mathrm{Int}, t_l)\ \}}$$

Figure 10. Typing system (recursive types)



Observe the use of **in** to indicate when an expansion has to be injected into the recursive type.

Another important example is that of functions over recursive types, or returning a recursive type. In the first case, consider the function counting the number of elements in a list:

let *len* = λl. **case** (**out** *l*) **of** **Nil** *x*. 0;
Cons *p*. 1 + *len* ($\pi_{2,2}$ *p*)
in *len* (**in** (**Cons** (3, **in** (**Cons** (4, **in** (**Nil** ()))))))

The type of *len* in the previous expression is (μt_l.**Variant**{ **Nil** : (), **Cons** : (**Int**, t_l) }) → **Int**. If the abbreviation **List_of** t_x is used for (μt_l.**Variant**{ **Nil** : (), **Cons** : (t_x,t_l) }), then the type of *len* is **List_of Int** → **Int**. For an example of a function returning a recursive type, consider the function generating a list of repeated elements:

let *repeat* = **fix** (λrep. λx. λn. **if** (*n*==0) **then in** (**Nil** ())
else in (**Cons** (*x*, *rep x* (*n*-1))))
in *repeat* **True** 3

The type of the function *repeat* used is **Bool** → **Int** → (μt_l.**Variant**{ **Nil** : (), **Cons** : (**Bool**,t_l) }), which, using the abbreviation **List_of** t_x can be written as **Bool** → **Int** → **List_of Bool**. Observe the use of recursion on integers to generate a finite list of the indicated length.

An important family of recursive types is that of trees. A *tree* is any recursive type where one of the alternatives has two or more uses of the recursively bound variable. Examples of tree types are μt. **Variant**{ **Leaf** : t_x, **Node** : (t_x,*t*,*t*) }, μt. **Variant**{ **Null** : (), **Bin** : (*t*,t_x,*t*) }, μt.**Variant**{ **Tip** : t_x, **Branch** : (*t*,*t*) }, and so on. Going deeper into these examples is beyond the scope of this chapter. However, a lot of literature exists on it (Bird, 1998; Okasaki, 1998).

A final curiosity about the power of recursive types is that in a language with recursive types, the operator **fix** can be constructed. Thus, there is no need to provide it as a primitive. A possible way to construct it is:

let *prefix* = $\lambda m.\, \lambda f.\, f$ (**out** *m m f*)
in *prefix* (**in** *prefix*)

The type of *prefix* is $(\mu t.\, t \to (t_x \to t_x) \to t_x) \to (t_x \to t_x) \to t_x$, and thus the type of (**in** *prefix*) is $\mu t.\, t \to (t_x \to t_x) \to t_x$ obtaining the required type, $(t_x \to t_x) \to t_x$, for **fix**. Verifying that this term has the desired behaviour is an easy exercise.

Qualified Types for Overloading

The theory of qualified types (Jones, 1994a) is a framework that allows the development of constrained type systems in an intermediate level between monomorphic and polymorphic type disciplines. Qualified types can be seen in two ways: either as a restricted form of polymorphism, or as an extension of the use of monotypes (commonly described as *overloading*, in which a function may have different interpretations according to the types of its arguments).

Predicates are used to restrict the use of type variables, or, using the second point of view, to express several possible different instances with one single type (but without the full generality of parametric polymorphism). The theory explains how to enrich types with predicates, how to perform type inference using the enriched types, and which are the minimal properties predicates must satisfy in order for the resulting type system to have similar properties as in the Hindley-Milner one. In particular, it has been shown that any well-typed program has a *principal type* that can be calculated by an extended version of Milner's algorithm.

As we have seen, polymorphism is the ability to treat some terms as having many different types, and a polymorphic type can be expressed by means of a *type scheme*, using universal quantification to abstract those parts of a type that may vary. The idea of qualified types is to consider a form of restricted quantification. If *P(t)* is a predicate on types and *f(t)* a type possibly containing variable *t*, the type scheme $\forall\, t.P(t) \Rightarrow f(t)$ can be used to represent the set of types:

{ $f(\tau)$ s.t. τ is a type such that $P(\tau)$ holds. }

and accurately reflect the desired types for a given term.

Thus, a key feature in the theory is the use of a language of *predicates* to describe sets of types (or, more generally, relations between types). The exact set of predicates may vary from one instance of the framework to another, but the theory effectively captures the minimum required properties by using an entailment relation ($\Vdash$) between (finite) sets

of predicates satisfying a few simple laws. If Δ is a set of predicates, then $\Delta \Vdash \delta$ indicates that the predicate δ can be inferred from the predicates in Δ.

The language of types is stratified in a similar way as in the Hindley-Milner system, where the most important restriction is that qualified or polymorphic types cannot be an argument of functions. That is, types are defined by a grammar with at least the productions:

$$\tau ::= t \mid \tau \to \tau$$

and, on top of them, are constructed qualified types of the form:

$$\rho ::= \tau \mid \delta \Rightarrow \rho$$

and then type schemes of the form

$$\sigma ::= \rho \quad | \quad t.\sigma$$

Type inference uses judgements extended within a context of predicates $\Delta \mid \Gamma \vdash e : \sigma$, representing the fact that when the predicates in Δ are satisfied and the types of the free variables of e are as specified by Γ, then the term e has type σ. Valid typing judgements can be derived using a system of rules that is an extension of the basic one presented here.

As mentioned, one of the uses of the theory of qualified types is to express overloading of functions (Jones, 1996), as it has been done for the language Haskell. Consider the Haskell declaration:

```
member x [] = False
member x (y:ys) = x == y || member x ys
```

The following typing judgement will hold for it:

$$\varnothing \mid \varnothing \vdash \text{member} : \quad a.\mathbf{Eq}(a) \Rightarrow a \to [a] \to \mathbf{Bool}$$

Observe how the use of the overloaded function (==) in the body of the function member is reflected in the predicate qualifying the variable a in the resulting type; this predicate states precisely that a type implements its own version of the overloaded function.

The implementation of a qualified type system follows the line of the presentation given in this chapter, with additions to keep track of predicates, and to simplify the set of predicates obtained for a type (Jones, 1994b).

Type-Based Analysis

The influence of the type discipline on program analysis, however, does not stop there. Typing techniques have been the inspiration for several developments in the fields of program analysis and validation and verification techniques. Typing techniques have been modified to be applied to the calculation of different kinds of effects like type and effects systems (Talpin & Jouvelot, 1992; Jouvelot & Gifford, 1991) — for example, typing references (Wright, 1992), register allocation (Agat, 1997), and so on—control flow analysis (Mossin, 1996; Volpano, Smith, & Irvine, 1996), compilation (Wand, 1997), program transformation (Danvy, 1998; Hannan & Hicks, 1988; Hughes, 1996b; Romanenko, 1990), code splicing (Thiemann, 1999), security (Volpano & Smith, 1997), and other forms of program analysis (O'Callahan, 1998).

In the rest of this section, some of these developments are discussed.

Types and Effects

Types and effects are based on the idea of considering *effects*, which are an abstraction extending types to express the consequence of imperative operations on certain *regions* — the abstraction of possible aliased memory locations. A basic example of effects is that of reading and writing a persistent memory — a possible syntax, where ε denotes effects and ρ regions, is given by:

$$\varepsilon ::= \varnothing \mid \varsigma \quad \mid \textbf{\textit{init}}(\rho,\tau) \mid \textbf{\textit{read}}(\rho) \mid \textbf{\textit{write}}(\rho) \mid \varepsilon \cup \varepsilon$$

where ∅ denotes the absence of effects, ς denotes effect variables, the operators ***init***, ***read***, and ***write*** approximate effects on regions, and operator ∪ gathers different effects together. Types are extended to express effects by adding an effect expression to the function type: $\tau \rightarrow^{\varepsilon} \tau$, and a special type $\textbf{\textit{ref}}_{\rho}(\tau)$ to express a reference to an element of type τ in region ρ. Typing judgements are also extended with effects, becoming $\Gamma \vdash e : \tau,\varepsilon$; the rules of the formal system are extended as well. Finally, basic operations on memory locations are given types with effects. For example, the operator := for assignment receives the type $\forall t.\forall \rho.\forall \varsigma.\forall \varsigma'.\textbf{\textit{ref}}_{\rho}(t) \rightarrow^{\varsigma} t \rightarrow^{\varsigma' \cup \textbf{\textit{write}}(\rho)} \mathbf{()}$, expressing that the assignment takes a reference to a location and a value, and returns a command (of type **()**) whose effect is writing in the given region; other operations on locations are treated similarly.

Types for Register Allocation

Register allocation is the process of deciding where, in which register or in the memory, to store each value computed by a program being compiled. In modern architectures, the difference in access time between a register and a memory cell can be as much as 4 to 10

times (or much more, if a cache miss occurs) (Henessy & Patterson, 1990). Moreover, typical RISC arithmetic can handle only register operands, and thus the difference in the number of required operations to add two numbers stored in registers against two stored in memory is of three instructions (two loads and a store of difference). For that reason, an optimal process of register allocation is crucial to obtain efficient programs.

By its very nature, register allocation must be done late in the compilation process, when the order of evaluation of every subterm has been decided. Typical algorithms to perform register allocation operate on assembly-code level, when all structure and type information of the compiler's intermediate language is lost. This means that calls to statically unknown functions must be made with some rigid calling convention, thus having the risk of requiring much more expensive operations due to misuse of registers. That is, if the register behaviour of the callee is known, the code produced to invoke the function can be optimized to use the best register allocation.

Types and effects, as described in the previous section, can be used to provide a type system describing the behaviour of register usage of all objects in a program (Agat, 1997, 1998). Types are enriched with register usage information, and the type system can be extended to perform optimal interprocedural register allocation, even when the function invoked is statically unknown. The main idea, then, is to move the register allocation phase from assembly-code to intermediate code, thus having all the power of types at hand.

Each type τ is annotated where the value of that type is, or is expected to be, stored, thus becoming τ_r. Function types are modified to be of the form $\tau_r \rightarrow^{k;d} \tau_r$, where $\boldsymbol{k}$, called the *kill-set*, is a description of the registers that might be modified when the function is applied, and $\boldsymbol{d}$, called the *displacement*, describes which registers contain values that might then be referenced. Examples of such types are:

- $\mathbf{Int}_{R3}$ **:** Integer value stored in register 3.
- $\mathbf{Int}_{S8}$ **:** Integer value stored in slot 8 of the current stack frame.
- $(\mathbf{Int}_{R2} \rightarrow^{\{R6,R4\};\{R3\}} \mathbf{Int}_{R4})_{R5}$ **:** Function accepting an integer argument in register 2 and returning an integer result in register 4. While computing the result, registers 6 and 4 might be altered, and the value currently in register 3 might be needed in the computation. The pointer to the function closure is stored in register 5.

There are several benefits to using such a typed system expressing the register assignments made by a register allocator. Since all values have a type, and types describe register behaviour, specialized calling conventions can be used for *all* applications, even those whose functions are not known statically; it is the type of the argument function that tells how to call it. Additionally, a clean and simple setting for reasoning about the correctness of the register-allocation algorithm is gained. Finally, several optimizations can be performed — for example, keeping values required by commonly used functions in registers as much as possible.

Sized Types

In the domain of safety-critical control systems (Levenson, 1986), a correct program is a program not causing any (unintentional) harm — like injury, damage, or death — to its environment. Correctness properties like these require system-level reasoning. A particular way in which one of these systems may cause harm is when some component exceeds its memory limits, thus causing unintentional state changes, or even damage, to the whole system. Errors of this kind are easy to introduce — a small change can do it, without notice — but are completely unacceptable. A formal technique allowing to reason about the memory limits of programs is thus a must.

The theory of sized types (Pareto, 1998) is a framework composed by a language aimed at control systems programming, accompanied by tools asserting program correctness from the memory consumption point of view. The main tool is a variation of a type system accepting only programs whose memory behaviour can be predicted. Some principles of design underlying the theory of sized types guide the development of programs. Basic principles are that types may exclude bottom (a particular theoretical value that represent nonterminating or erroneous computations) — that is, certain types can only be used for type terminating programs — that types may distinguish finite, infinite, and partial values. For example, the availability of data from a given source can be assured by typing it with an infinite type with no partial values, and types may bound their values by size parameters in several dimensions. So, an upper bound in the number of elements of a list, or a lower bound on the number of elements produced by a data source can be determined (and those elements can also be bounded in size).

The particular way to achieve the principles is by using data declarations with additional information. For example, consider the following declarations:

```
data Bool = False | True

idata List ω a = Nil | Cons a (List ω a)

codata Stream ω a = Make a (Stream ω a)

data BS k = BS (Stream k Bool)
```

The first declaration establishes the type Bool as a sum type, but excludes explicitly the possibility of undefined or nonterminating expressions of this type. Thus, if a program can be typed with Bool, it can be asserted that it terminates and produces a value either False or True.

The second declaration defines recursive sum types for lists, but adding an extra parameter ω. This new parameter is a bound on the size of the elements of the type, and the keyword **idata** establishes that several new types are introduced, all finite: one for lists of size less than k for each possible natural number k, and the classical type for lists of any (finite) size.

The third declaration uses the keyword **codata** to define that the elements of the declared type are infinite. It also has a bound parameter, but in this case it is used to determine a lower bound on the size of the streams. So, again, several new types are introduced: one for each natural number k of streams of size bigger than k, and one for streams that will always produce new data on demand.

The last declaration has a parameter that fixes the size of the lower bound for the stream of Booleans declared.

A carefully defined syntax for programs, and its corresponding semantics are needed to assure that all the desired principles are valid. A type system assigning sized types to programs is defined, and also an algorithm calculating the assignment. In this way, the type system can be used to assure the property of bounded size on typable programs (Hughes, Pareto, & Sabry, 1996).

Type Specialization

Program specialization is a form of a automatic program generation taking a given program as input to produce one or more particular versions of it as output, each specialized to particular data. The program used as input is called the *source program*, and those produced as output are called the *residual programs*.

Type Specialization is an approach of program specialization introduced by John Hughes in 1996 (Hughes, 1996a, 1996b, 1998) for typed languages. The main idea of type specialization is to specialize *both* the source program and its type to a residual program and residual type. In order to do this, instead of a generalized form of evaluation — as the one used in partial evaluation, the most common form of program specialization — type specialization uses a generalized form of type inference.

The key question behind type specialization is:

How can the static information provided by the type of an expression be improved?

An obvious first step is to have a more powerful typesystem. But, it is also desirable to *remove* the static information expressed by this new type from the code in order to obtain a simpler and (hopefully) more efficient code. So, type specialization works with two typed languages: the source language, in which the programs to be specialized are coded, and the residual language, in which the result of specialization is expressed. The source language, whose elements are denoted by e, is a two-level language where every construct is marked as either static or dynamic. Static information is supposed to be moved to the type, while a dynamic one is supposed to be kept in the residual term; source types, denoted by τ, reflect the static or dynamic nature of expressions, and thus they also contain the marks. The residual language, whose terms and types are denoted by e' and τ' respectively, has constructs and types corresponding to all the dynamic constructs and types in the source language, plus additional ones used to express the result of specializing static constructs.

In order to express the result of the specialization procedure, Hughes introduced a new kind of judgement and a formal system of rules to infer valid judgements. These judgements, similar to typing judgements in the source language, make use of environments to determine the specialization of free variables. The new judgements have the form $\Gamma \vdash e : \tau \rightsquigarrow e' : \tau'$ which can be read as "if the free variables in e specialize to the expressions indicated in Γ, then source expression e of source type τ specializes to residual expression e' and residual type τ'". The valid judgements are obtained by using a system of rules specifying how to specialize a given typed term; this system follows the same ideas of typing systems.

A very interesting example of type specialization is to specialize an interpreter for a language to a given term: given an interpreter *int* for a language and a program *p* written in it, the specialization of *int* to *p* is a program equivalent to *p*. This is called a Futamura projection (Futamura, 1971), and it compiles by specializing the interpreter. When this way of compiling programs can be done optimally for all the programs in the language, then it can be assured that the specialization method can tackle all the constructs in it. Type specialization was the first approach obtaining optimality for compilation of typed languages by specialization.

The implementation of a type specialization algorithm is rather tricky, and although John Hughes has provided a prototype implementation (Hughes, 1997), it is difficult to understand, and properties about it are difficult to prove. However, Martínez López has given a systematic way to implement type specialization following the lines presented in this chapter (Martínez López & Hughes, 2002; Martínez López, 2005), together with a proof of a very important property of the system: the existence of a principal specialization for every source term. Current research on type specialization involves the addition of sum types to the principal specialization system, and the addition of dynamic recursion.

Conclusion

Type-based analyses are an important way to structure a system for verification or validation of programs. However, the design of a type-based analysis usually does not directly suggest a good way to implement it. This chapter has addressed the problem of going from a specification of a type system to its implementation, showing several of the common subtleties that are a hindrance for designers and implementers. Additionally, the correctness and completeness of the implementation's design were stated as theorems; their proofs are easy examples of induction on derivations, including several lemmas and additional propositions. The case presented here is the most basic one, but it is useful in order to cast light on the techniques needed.

In addition, several extensions to the basic system were discussed, and several type-based analyses were mentioned and referenced. Some of the most important among them are register allocation, sized types, and type specialization; other approaches, such as control-flow analysis and code splicing, that have not been considered are also important.

Acknowledgments

I am very grateful to my colleague Eduardo Bonelli for his careful reading and his many suggestions to improve the material presented here. I am also in debt to Jerónimo Irazábal because he agreed to translate my functional code to C++, even when the task was not as easy as it seemed at first. Two more people deserving my gratitude are Andrés Fortier, who has given his time to help me with the subtleties of the object oriented paradigm, and Alejandro Russo, who reviewed the early drafts. Finally, the anonymous referees, because they have given many suggestions on how to improve the chapter.

References

Abadi, M., & Cardelli, L. (1996). *A theory of objects*. Springer-Verlag.

Aiken, A. (1999). Introduction to set constraint-based program analysis. *Science of Computer Programming, 35*, 79-111.

Agat, J. (1997). Types for register allocation. In *IFL'97* (LNCS 1467).

Agat, J. (1998). *A typed functional language for expressing register usage*. Licenciate Dissertation. Chalmers University of Technology.

Altenkirch, T., Gaspes, V., Nordström, B., & von Sydow, B. (1994). *A user's guide to ALF* (Technical report). Sweden: Chalmers University of Technology.

Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P., & Morrison, R. (1983). An approach to persistent programming . *Computer Journal, 26*(4), 360-365.

Atkinson, M. P., & Morrison, R. (1988). Types, bindings and parameters in a persistent environment. In M.P. Atkinson, O. P. Buneman, & R. Morrison (Eds.), *Data types and persistence*. Springer-Verlag.

Bertot, Y., & Castéran, P. (2004). Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions. In *Texts in theoretical computer science. An EATCS Series*. Springer-Verlag.

Bird, R. (1998). *An introduction to functional programming using Haskell*. Prentice-Hall.

Breazu-Tannen, V., Coquand, T., Gunter, C. A., & Scedrov, A. (1991). Inheritance as implicit coercion. *Information and Computation, 93*, 172-221.

Cardelli, L. (1997). Type systems. In *CRC Handbook of computer science ad engineering* (2nd ed.). CRC Press.

Cardelli, L., & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *Computing Surveys, 17*(4), 471-522.

Clément, D., Despeyroux, J., Despeyroux, T., & Kahn, G. (1986). A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (pp. 13-27). Cambridge, MA: ACM Press.

The Coq proof assistant (2004). Retrieved from http://coq.inria.fr/ ECurry, H. B., & Feys, R. (1958). *Combinatory logic*. Amsterdam, The Netherlands: North Holland.

Coquand, T., & Huet, G. (1988). The calculus of constructions. *Information and Computation, 76*(2/3), 95-120.

Damas, L., & Milner, R. (1982). Principal type-schemes for functional languages. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages,* Albuquerque, NM (pp. 207-212).

Danvy, O. (1998). Type-directed partial evaluation. In J. Hatcliff, T. E. Mogensen, & P. Thiemann (Eds.), *Partial evaluation — Practice and theory* (LNCS 1706, pp. 367-411). Copenhagen, Denmark: Springer-Verlag.

Dearle, A. (1988). *On the construction of persistent programming environments.* PhD thesis, University of St. Andrews.

Freund, S. N., & Mitchell, J. C. (1998). A type system for object initialization in the Java bytecode language. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'98)* (pp. 310-328).

Futamra, Y. (1971). Partial evaluation and computation process — an approach to a compiler-compiler. *Computer, Systems, Controls, 2*(5), 45-50.

Gamma, R., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison Wesley.

Gaster, B. R., & Jones, M. P. (1996). *A polymorphic type system for extensible records and variants* (Tech. Rep. No. NOTTCS-TR-96-3). Department of Computer Science, University of Nottingham.

Girard, J. Y. (1989). *Proofs and types*. Cambridge University Press.

Hannan, J. J., & Hicks, P. (1988). Higher-order arity raising. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'98)* (pp. 27-38). Baltimore: ACM.

Henglein, F., & Rehof, J. (1997). The complexity of subtype entailment for simple types. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science* (pp. 352-361). Warsaw, Poland.

Hennessy, J. L. & Patterson, D. A. (1990). *Computer architecture: a quantitative approach*. Palo Alto: Morgan Kaufmann Publishers.

Hindley, J. R. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society, 146*, 29-60.

Hindley, J. R. (1995). *Basic simpletTypetTheory*. Cambridge University Press.

Hoang, M., Mitchell, J. C., & Viswanathan, R. (1993). Standard ML: NJ weak polymorphism and imperative constructs. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science (LICS)*. Journal version appeared in (Mitchell & Viswanathan, 1996).

Hughes, J. (1996a). An introduction to program specialisation by type inference. In *Functional Programming*. Scotland: Glasgow University. Published electronically.

Hughes, J. (1996b). Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.), *Selected papers of the International Seminar "Partial Evaluation"* (LNCS 1110, pp. 183-215). Springer-Verlag.

Hughes, J. (1997). *Type specialiser prototype*. Retrieved from http://www.cs.chalmers.se/~rjmh/TypeSpec2/

Hughes, J. (1998). Type specialisation. In *ACM Computing Surveys, 30*. ACM Press. Article 14. Special issue: electronic supplement to the September 1998 issue.

Hughes, J., Pareto, L., & Sabry, A. (1996). Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. FL: ACM Press.

Igarashi, A., & Kobayashi, N. (2001). A generic type system for the Pi-calculus. *ACM SIGPLAN Notices, 36*(3), 128-141.

Jim, T. (1996). What are principal typings and what are they good for? In *Proceedings of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 42-53). St. Petersburg Beach, FL: ACM Press.

Jones, M. P. (1994a). Qualified types: Theory and practice. In *Distinguished dissertations in computer science*. Cambridge University Press.

Jones, M. P. (1994b). *Simplifying and improving qualified types* (Tech. Rep. No. YALE/DCS/RR-989). Yale University.

Jones, M. P. (1996). Overloading and higher order polymorphism. In E. Meijer & J. Jeuring (Eds.), *Advanced functional programming* (LNCS 925, pp. 97-136). Springer-Verlag.

Jouvelot, P., & Gifford, D. (1991). Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*. Orlando, FL: ACM Press.

Knabe, F. (1995). *Language support for mobile agents*. PhD thesis. Pittsburgh, PA: Carnegie Mellon University.

Leveson, N. G. (1986). Software safety: Why, what, and how. *ACM Computing Surveys*, *18*(2), 25-69.

Martínez López, P. E. (2005). *The notion of principality in type specialization*. PhD thesis, Argentina: University of Buenos Aires. Retrieved from http://www-lifia.info.unlp.edu.ar/~fidel/Work/Thesis.tgz

Martínez López, P. E. (in press). Static type systems: A functional implementation (Technical report). LIFIA, Facultad de Informática, Universidad Nacional de La Plata.

Martínez López, P. E., & Hughes, J. (2002). Principal type specialisation. In W. N. Chin (Ed.), *Proceedings of the 2002 ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-based Program Manipulation* (pp. 94-105). ACM Press.

McCarthy, J. (1962). Towards a mathematical science of computation. In C. Popplewell (Ed.), *Information processing 1962: Proceedings of IFIP Congress 62*. Amsterdam, The Netherlands: North-Holland.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences, 17*(3).

Mitchell, J. C. (1991). Type inference with simple subtypes. *Journal of Functional Programming, 1*(3), 245-285.

Mitchell, J. C. (1996). *Foundations for programming languages*. MIT Press.

Mitchell, J. C., & Plotkin, G. D. (1988). Abstract types have existential type. *ACM Transactions on Programming Languages and Systems, 10*(3), 470-502.

Mitchell, J. C., & Viswanathan, R. (1996). Standard ML-NJ weak polymorphism and imperative constructs. Information and Computation*, 127*(2), 102-116.

Morrison, R., Connor, R. C. H., Cutts, Q. I., Kirby, G. N. C., & Stemple, D. (1993). Mechanisms for controlling evolution in persistent object systems. *Journal of Microprocessors and Microprogramming, 17*(3), 173-181.

Mossin, C. (1996). *Flow analysis of typed higher-order programs*. PhD thesis. Denmark: DIKU, University of Copenhagen.

Nordlander, J. (1999). *Reactive objects and functional programming*. PhD thesis. Chalmers University of Technology.

Nordlander, J. (2000). Polymorphic subtyping in O'Haskell. In *Proceedings of the APPSEM Workshop on Subtyping and Dependent Types in Programming*.

Nordström, B., Petersson, K., & Smith, J. M. (1990). *Programming in Martin-Löf's Type Theory*. Oxford University Press.

O'Callahan, R. (1998). *Scalable program analysis and understanding based on type inference*. PhD thesis proposal. Pittsburgh, PA: School of Computer Science, Carnegie Mellon University.

Object Management Group (2004). *Unified Modelling Language specification, version 2.0*. Retrieved from http://www.uml.org/#UML2.0

Odersky, M., Wadler, P., & Wehr, M. (1995). A second look at overloading. In *Proceedings of ACM Conference on Functional Programming and Computer Architecture.* ACM Press.

Okasaki, C. (1998). *Purely functional data structures*. Cambridge University Press.

Palsberg, J., Wand, M., & O'Keefe, P. (1997). Type inference with non-structural subtyping. *Formal Aspects of Computer Science, 9*, 49-67.

Pareto, L. (1998). *Sized types.* PhD thesis. Chalmers University of Technology.

Peyton Jones, S., & Hughes, J. (Eds.). (1999). *Haskell 98: A non-strict, purely functional language.* Retrieved from http://haskell.org

Pierce, B. (2002). *Types and Programming Languages*. MIT Press.

Pottier, F. (2000). A 3-part type inference engine. In G. Smolka (Ed.), *Proceedings of the 2000 European Symposium on Programming (ESOP '00)* (LNCS 1782, pp. 320-335). Springer-Verlag.

Rèmy, D. (1989). Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages* Austin, TX.

Rèmy, D., & Pottier, F. (2004). The essence of ML type inference. In B. Pierce (Ed.), *Advanced topics in types and programming languages*. MIT Press.

Reynolds, J. C. (1974). Towards a theory of type structure. In *Proceedings of the Symposium on Programming* (LNCS 19, pp. 408-423). Springer-Verlag.

Reynolds, J. C. (1983). Types, abstraction, and parametric polymorphism. In R. E. A. Mason (Ed.), *Information Processing'83. Proceedings of the IFIP 9th World Computer Congress* (pp. 513-523). Amsterdam, The Netherlands: North-Holland.

Reynolds, J. C. (1998). *Theories of programming languages*. Cambridge University Press.

Romanenko, S. (1990). Arity raising and its use in program specialisation. In N. D. Jones (Ed.), *Proceedings of 3rd European Symposium on Programming (ESOP'90)* (LNCS 432, pp. 341-360). Copenhagen, Denmark: Springer-Verlag.

Shields, M., & Meijer, E. (2001). Type-indexed rows. In *Proceedings of the 28th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL '01),* London (pp. 261-275).

Stroustrup, B. (2000). *The C++ programming language* (special 3rd ed.). Addison-Wesley.

Sulzmann, M., Odersky, M., & Wehr, M. (1997). Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*. Journal version appeared in (Sulzmann, Odersky, & Wehr, 1999).

Sulzmann, M., Odersky, M., & Wehr, M. (1999). Type inference with constrained types. *Theory and Practice of Object Systems, 5*(1), 33-55.

Syme, D. (1999). Proving Java type soundness. In J. Alves-Foss (Ed.), *Formal syntax and semantics of Java* (LNCS 1523, pp. 119-156). Springer-Verlag.

Talpin, J. P., & Jouvelot, P. (1992). The type and effect discipline. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*. Santa Cruz, CA: IEEE Computer Society Press.

Thatte, S. R. (1992). *Typechecking with ad-hoc polymorphism* (Technical report preliminary manuscript). Potsdan, NY: Clarkson University, Department of Mathematics and Computer Science.

Thiemann, P. (1999). Higher-order code splicing. In S. D. Swierstra (Ed.), *Proceedings of the Eighth European Symposium on Programming* (LNCS 1576, pp. 243-257). Springer-Verlag.

Tiuryn, J., & Wand, M. (1993). Type reconstruction with recursive types and atomic subtyping. In *CAAP '93: 18th Colloquium on Trees in Algebra and Programming*.

von Oheimb, D. & Nipkow, T. (1999). Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss (Ed.), *Formal syntax and semantics of Java* (LNCS 1523, pp. 119-156). Springer-Verlag.

Volpano, D., & Smith, G. (1997). A type-based approach to program security. In *Proceedings of TAPSOFT'97, Colloqium on Formal Approaches in Software Engineering*.

Volpano, D., Smith, G., & Irvine, C. (1996). A sound type system for secure flow analysis. *Journal of Computer Security, 4*(3), 1-21.

Wand, M. (1987). Complete type inference for simple objects. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science* (pp. 37-44).

Wand, M. (1991). Type inference for record concatenation and multiple inheritance. *Information and Computation, 93*, 1-15. Preliminary version appeared in *Proc. 4th IEEE Symposium on Logic in Computer Science* (1989), 92-97.

Wand, M. (1994). Type inference for objects with instance variables and inheritance. In C. Gunter & J. C. Mitchell (Eds.), *Theoretical aspects of object-oriented programming* (pp. 97-120). MIT Press. Originally appeared as Northeastern University College of Computer Science (Tech. Rep. No. NU-CCS-89-2), February 1989.

Wand, M. (1997). Types in compilation: Scenes from an invited lecture. In *Workshop on Types in Compilation (invited talk), held in conjunction with ICFP97*.

Wells, J. B. (1994). Typability and typechecking in second-order lambda calculus are equivalent and undecidable. In *Proceedings of the 9th Annual IEEE Symposium Logic in Computer Science.* Superseded by (Wells, 1998).

Wells, J. B. (1998). Typability and typechecking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic, 98*(1-3), 111-156.

Wright, A. K. (1992). Typing references by effect inference. In B. Krieg-Bruckner (Ed.), *Proceedings of ESOP'92, 4th European Symposium on Programming* (LNCS 582, pp. 473-491). Springer-Verlag.

Wright, A. K. (1995). Simple imperative polymorphism. *Lisp and Symbolic Computation*, *8*(4), 343-355.