UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA

Departamento de Computación

Proxecto de Fin de Carreira de Enxeñaría Informática

A Damas-Milner Type System Extended with Predicate Subtypes for Correct Software Development

Author: Iago Abal Rivas

Advisor: Gilberto Pérez Vega

A Coruña, September 20, 2012

Especificación

Título: Un sistema de tipos Damas-Milner extendido con predicate subtypes para o desenvolvemento de software correcto

Clase: Investigación e Desenvolvemento

Autor: Iago Abal Rivas

Director: Gilberto Pérez Vega

Tribunal:

Fecha de lectura:

Calificación:

To my family and friends, especially my loving parents Isabel and Jesus.

Acknowledgments

Appart from our efforts, the success of any project depends largely on the encouragement and guidelines of many others. I take this opportunity to express my gratitude to the people who have been instrumental in the successful completion of this project.

I would like to thank my advisor, Dr. Gilberto Pérez Vega, for the help and guidance that he has given me over the past two years. I also owe a special debt of gratitude to David Castro Pérez for his encouragement, feedback, and many fruitful discussions.

I am very grateful to those who introduced me to functional programming and formal methods, for changing the way I think about software. I especially thank the staff of *Formal Methods in Software Engineering* at University of Minho, for their excellent work in teaching and the valuable background that their course gave me.

Most importantly, I would like to thank my parents for all their love and support over the years, for teaching me the value of work and the important things in life, and for helping me develop into the person that I became.

A more mathematical approach is inevitable. Professional software development will become more like a true engineering discipline, applying mathematical techniques. I don't know how long this evolution will take, but it will happen.

Bertrand Meyer

Abstract

Software errors are increasingly common, hard to detect, and very expensive for both software companies and end-users. As the complexity of software systems continues to grow, this situation can only get worse. The adoption of rigorous development processes with emphasis in V&V activities, has been demonstrated to reduce the rate of errors considerably.

Yet, this is not enough. Along with the development process, the programming language used to build a software system also has a great impact on the number of errors in it. In particular, functional programming languages with strong static typing have been shown to be far superior in this respect. Programming by contract can be seen as an extension of the type system to enrich its specification capabilities, and it is becoming increasingly popular in both mainstream and safety-critical settings. Contract checking, together with automated static analysis techniques, allows a wide range of program errors to be detected at compile time.

This thesis proposes an approach to introduce contracts as types into mainstream statically-typed functional programming languages. For this purpose, we have extended a Damas-Milner type system with predicate subtypes in the spirit of PVS, and developed a proof-of-concept programming language —$\mathcal{H}_{\text{SPEC}}$— based on this system. In this way, the concepts of pre/postcondition and type invariant are built into the type system as *contract types*, and integrate smoothly with type polymorphism.

Keywords

Predicate subtypes, Refinement types, Programming by contract, Subset coercions, Damas-Milner, Hindley-Milner, Contract checking, Automated random testing.

Contents

List of Figures

List of Tables

List of Definitions

Chapter 1

Introduction

Software is important in our society, to the extent that it is present in every aspect of our daily lives. Our dependency on software goes far beyond things like our cell phones, TV sets and cars; all of our infrastructures depend on software, including energy, communication, transport, finances and healthcare. Software defects have been the cause of patient's death, spaceflight accidents and stock market crashes, just to name a few. Software malfunction affects different people, in different ways, and to different degrees. In economic terms, the costs of faulty software in the U.S. economy have been estimated to range from $22 to $60 billion per year, and over half of these costs are borne by end users [1]. The correctness of software must be taken seriously.

Unfortunately, there is no magic bullet for developing correct (error free) software. Indeed, the ideal of software correctness should be taken with a grain of salt. Zero-defect software is, in the words of Tony Hoare, "an impossible dream" for many reasons. As far as software engineering is concerned, correctness is a means to attain reliable software. Software reliability, a key factor in software quality [2, 3], is defined as *the probability of failure-free software operation for a specified period of time in a specified environment* (ANSI/IEEE 729-1991). For instance, the levels of reliability required in the avionics industry vary from a 10^{-5} to a 10^{-9} failure rate per (flight) hour. In such safety-critical settings, those high levels of reliability are achieved through the application of a rigorous development process with strong emphasis in verification and validation (V&V) activities.

There are plenty of ways to approach software reliability, but many of them —the

so called heavyweight formal methods— are overkilling and cost-prohibitive for mainstream settings. The paradigm of *programming by contract* has been demonstrated to be a cost-effective way to integrate program specification and verification into the traditional programming workflow [4]. It prescribes the (formal) specification of programs with logical assertions, in the form of pre- and postcondition. Depending on the methods employed for *contract checking*, this paradigm may be used in developing more reliable mainstream software [5, 4] or high-assurance software [6, 7] for safety-critical systems.

Contract programming and *static contract checking* have been widely studied in procedural and object-oriented languages. Recent advances in automated verification have brought contracts to mainstream programming [5, 6, 8], and it is receiving significant attention from industry. On the functional programming side, however, researchers (mostly from academia) seem more interested in other approaches to software correctness. Still, important research has been done in this direction [9, 10, 11, 12, 13, 14]. The present work is yet another attempt to bring the attention back to contract checking in functional programming. We believe that this programming paradigm, due to its strong mathematical foundations, is particularly well-suited for contract-based verification.

In this project, we focus on the approach of *types as contracts*, where the type system allows types to be refined by predicates. A *subset type* such as $\{x\,\texttt{:}\,\texttt{Int} \mid x \geq 0\}$ denotes the set of terms x of type `Int` that satisfy the predicate $x \geq 0$ —that is, the set of natural numbers. This notion of "refined types" has been used with great success in the PVS system [15], through a type feature known as *predicate subtypes* [16]. Later, Matthie Sozeau adapted the *predicate subtyping* mechanism of PVS to the case of a programming language —Russell— based on dependent type theory, leading to the concept of *subset coercions* [17].

This thesis proposes an extension to incorporate contracts as types into mainstream functional programming languages, as a practical realization of the ideas introduced in Russell. Our approach integrates smoothly with a ML-like type system, and supports polymorphic contract types as well as contract inference. What is more, we argue that a programming language based on our proposal naturally embodies a lightweight and incremental ("pay as you go") verification philosophy.

1.1 A motivating example

Contract types significantly extend the expressive power of the type system, allowing types to capture function pre- and postconditions, and data type invariants. Consider the (original) Euclidian division algorithm —implemented in a Haskell-like language:

```
euclid x y | x < y = (0,x)
           | else  = let (q,r) = euclid (x-y) y
                     in (q+1,r)
```

A traditional type system would assign `euclid` the type `Int -> Int -> (Int,Int)`. Ideally, the type of a function should specify what it does, and the typechecker uses types to rule out bad programs. However, `Int -> Int -> (Int,Int)` is a rather weak specification of `euclid`: it simply says that `euclid` takes two integers and returns another two. Actually, the above algorithm only works for natural numbers. Moreover, if the divisor is non-positive then the algorithm diverges.

In a language with contract types, as we propose in this thesis, it is possible to express these restrictions in the form of subset types. One can define type synonyms for natural and positive numbers,

```
type Nat = {n:Int|n>=0}
type Pos = {n:Int|n>0}
```

and use them to annotate `euclid` with a more precise type:

```
euclid : Nat -> Pos -> (Nat,Nat)
```

In this way we have converted `euclid` into a total function, since it is (well-)defined for every input of the specified types. The type system can now rule out bad programs like `euclid 1 0`. What is more, the typechecker can take advantage of this to detect subtle implementation errors. Suppose we mistyped `| y < x = ...` instead of `x < y`, then `else` would stand for `x <= y` and thus `euclid (x-y) y` would become ill-typed, because in this context `x-y` may not be a natural number as required by the type of `euclid` —a trivial counterexample is `x` $\mapsto$ `0` and `y` $\mapsto$ `1`.

Yet, `Nat -> Pos -> (Nat,Nat)` does not describe a division algorithm precisely. There are plenty of functions that meet such specification; `euclid x y = (0,0)` does, for instance. Refining this specification requires to state a relation between the input

and the output of our function. We do this through dependent function types, a generalization of the function space by allowing the range to depend on the domain. The following type unambiguously characterizes a division function for natural numbers:

```
euclid : {a:Nat} -> {b:Pos} -> (q:Nat,r:Nat|a == b*q+r && r < b)
```

Compared with other approaches where types and contracts are orthogonal, this promotion of types to specifications resembles languages based on dependent type theory and, in our view, feels more natural to the (functional) programmer.

1.2 Contributions

In this work we develop a new functional programming language called $\mathcal{H}_{\text{SPEC}}$, based on the syntax and semantics of Haskell, and with built-in support for programming by contract. The type system of $\mathcal{H}_{\text{SPEC}}$ is an extension of the Damas-Milner[1] type system [18] with predicate (sub)types and (limited forms of) dependent types in the spirit of PVS. Our major contribution is a type inference algorithm for this extended system, derived from the original Algorithm $\mathcal{W}$ proposed by Damas and Milner, exploiting the coercion mechanism of Russell. Additionally, we make the following contributions:

1. We introduce a syntactic extension of subset types, which we refer to as *pattern types*, that allows types to be refined by pattern matching. A pattern type like {`(x::xs):[T]`} denotes the set of non-empty lists of type `T`.

2. We prototype a typechecker for the $\mathcal{H}_{\text{SPEC}}$ language in Haskell. Our implementation constitutes an original application of advanced type extensions, recently added to Haskell, in compiler construction.

3. We develop a proof-of-concept contract checker, providing automated testing and automated theorem proving back-ends.

[1] Also known as Hindley-Milner system, it is the basis of the type systems of most modern functional programming languages, such as ML and Haskell.

1.3 Reading guide

This dissertation is organized into eight chapters. The first three chapters present the motivation for this work, introduce the necessary background in type systems 2, and depict the state-of-the-art of correct software development 3. The next three chapters describe the syntax and semantics of the $\mathcal{H}_{\text{SPEC}}$ language 4, its type system and type inference algorithm 5, and the implementation of a reference typechecker 6. Finally, we analyze the practical usefulness of $\mathcal{H}_{\text{SPEC}}$ for developing mainstream software 7, and conclude with a discussion of the contributions and directions for future work 8.

- Chapter 2 aims to provide a broad introduction to type systems. The concepts introduced here are instrumental to understand the theoretical development of this work. It should be possible, however, to get an insight about this theoretical work without such background. The reader familiar with typed lambda calculus and Damas-Milner type systems may skip most of the chapter. A quick reading is highly recommended though, especially of section § 2.3, because of the notations and conventions that are introduced within the chapter.

- Chapter 3 presents an overview of the state of the art approaches to correct software development. This chapter is only intended to contextualize this work, and to provide further references for the interested reader.

- Chapter 4 describes the syntax and semantics of the $\mathcal{H}_{\text{SPEC}}$ programming language. Syntax is precisely described using EBNF notation, while the semantics are informally described through a set of rules to translate the high-level constructs of $\mathcal{H}_{\text{SPEC}}$ into more basic and standard constructs of lambda calculus.

- Chapter 5 formulates the $\mathcal{H}_{\text{SPEC}}$'s type system as an extension of the Damas-Milner system. We provide both a declarative and an algorithmic presentation of the type system, as well as a detailed type inference algorithm. A careful read of section § 2.3, to become familiar with the Damas-Milner system and the Algorithm $\mathcal{W}$, is required for a deep understanding of this chapter. Nonetheless, the declarative presentation of the type system should be readable without special background.

- Chapter 6 presents the HSC $\mathcal{H}_{\text{SPEC}}$ compiler, a reference implementation of the type inference algorithm that we developed for $\mathcal{H}_{\text{SPEC}}$ in the previous chapter. Most of the chapter is dedicated to describe the compilation process and the software architecture of the compiler, thus no theoretical background is required, except for basic notions of compiler technology.

- Chapter 7 offers a preliminary analysis of the practical usefulness of $\mathcal{H}_{\text{SPEC}}$ for developing reliable software in mainstream settings. We use $\mathcal{H}_{\text{SPEC}}$ to implement a number of representative case studies, and then we evaluate the burden associated with the use of contract types and the bug-finding capabilities of our checking toolset.

- Finally, chapter 8 closes this dissertation with a discussion of the most important conclusions and contributions of this work, and also outlines some interesting directions for future work.

Chapter 2

Type Systems

The *lambda calculus* (also written as λ-calculus) is a formal system for expressing computation, which is the basis of *functional programming*. It was first formulated by the mathematician Alonzo Church in the 1930s [19, 20] as a way to formalize mathematics through the notion of functions. Alonzo Church first described what is nowadays known as the *untyped lambda calculus*, which can be understood as a computational system or as a logical system. Stephen Kleene and J. B. Rosser showed in 1935 that the *untyped lambda calculus*, as a logical system, is inconsistent. This discovery motivated Alonzo Church to introduce in 1940 a computationally weaker, but logically consistent, version of his original system known as the *simply typed lambda calculus*. This new system, which is often denoted as $\lambda^{\rightarrow}$, is the basis of modern type systems.

The limited expressiveness of $\lambda^{\rightarrow}$ motivated the development of a family of typed lambda calculus. Broadly, these extensions define richer type expressions covering a wider range of well-typed programs. A very important typed lambda calculus is the so-called *polymorphic lambda calculus*, denoted as λ^2, which formalizes the notion of parametric type polymorphism. This system, also known as System F, constitutes the theoretical basis of modern functional programming languages such as Haskell and ML, and also of the language subject of this thesis, $\mathcal{H}_{\text{SPEC}}$.

A richer type system provides more expressiveness but it can also make programming more cumbersome. Most typed lambda calculus, including λ^2, require a number of type annotations in order to get a decidable type inference algorithm. In practice, a type system should require a small amount of type annotations, otherwise programs

become too verbose to write. Traditionally, functional language designers have preferred type systems that allow for type inference while requiring no type annotations at all. This is not possible for λ^2, and for this reason programming languages such as Haskell and ML do not support arbitrary polymorphic types. Instead, these languages offer a restricted class of polymorphism for which complete type inference is decidable.

This restriction of System F is widely known as the Damas-Milner (or Hindley-Milner) system, independently discovered by J. Roger Hindley and Robin Milner, and later formalized by Luis Damas. Damas and Milner proposed the first algorithm to perform type inference for this system, known as the Algorithm $\mathcal{W}$ [18]. Despite Damas-Milner type systems significantly restrict the universe of polymorphic terms accepted by System F, it turns out that this calculus is expressive enough for most practical purposes.

This chapter is intended to offer a broad introduction to the theoretical foundations of type systems. The concepts introduced here are instrumental to understand the development of $\mathcal{H}_{\text{SPEC}}$ through the core chapters of this dissertation, especially the extended Damas-Milner system described in chapter 5. First, we introduce basic type systems through the simply-typed lambda calculus in § 2.1; then § 2.2 presents System F, to introduce the notion of type polymorphism; and next in § 2.3 we describe the Damas-Milner system and the Algorithm $\mathcal{W}$. We would like to point the interested reader to *Types and Programming Languages* [21], which is an excellent text and reference book for this field.

2.1 Simply typed lambda calculus

The *simply typed lambda calculus*, denoted $\lambda^{\rightarrow}$, is a typed interpretation of the lambda calculus and it lies at the basis of all type systems. This system considers a set of *base types* $\mathcal{B}$, also called *atomic types* or *type constants*, and only one type constructor ($\rightarrow$) that builds function types. It is the canonical and simplest example of a typed lambda calculus.

Definition 2.1 (Types ($\lambda^{\rightarrow}$)). Let $\mathcal{B}$ be a countable set whose members will be called atomic types. The set T of types of $\lambda^{\rightarrow}$ is defined as:

$$T ::= \mathcal{B} \mid T \rightarrow T$$

We use $\sigma, \tau, \ldots$ to denote arbitrary types. The type constructor $\to$ is defined to be right associative, and consequently outermost parenthesis can be omitted without ambiguity: e.g. $\sigma \to (\tau \to \varphi)$ can be written as $\sigma \to \tau \to \varphi$.

Example 2.1 ($\lambda^{\to}$ types)
For $\mathcal{B} = \{\mathsf{int}, \mathsf{bool}\}$ *the following are valid* $\lambda^{\to}$ *types:* $\mathsf{int}, \mathsf{bool}, \mathsf{int} \to \mathsf{int}, \mathsf{bool} \to \mathsf{bool}, \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}, \mathsf{int} \to \mathsf{bool}, \mathsf{int} \to \mathsf{int} \to \mathsf{bool}, (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}, \ldots$

Definition 2.2 (Terms ($\lambda^{\to}$)). Let $\mathcal{V}$ be a countable infinite set whose members will be called *term variables*. The set E of terms of $\lambda^{\to}$ is defined as:

$$E ::= \mathcal{V} \mid \lambda \mathcal{V} : T.\ E \mid E\ E$$

We use $x, y, \ldots$ to denote arbitrary term variables, and $M, N, \ldots$ to denote arbitrary terms. A term of the form $\lambda x : \sigma.\ M$ is called a λ-abstraction, this construct denotes an unnamed function and abstracts terms over terms. The λ (abstraction) operator binds any occurrence of a variable within the body of the abstraction, and it extends as far right as possible. For clarity we may group λ-abstractions to write $\lambda x_1 : \sigma_1.\ \ldots\ \lambda x_n : \sigma_n.\ M$ as $\lambda (x_1 : \sigma_1) \ldots (x_n : \sigma_n).\ M$. A term of the form $M\ N$ is called an *application* of M to N. Application has the highest precedence and it is defined to be left associative, hence we will write $((M\ N_1)\ N_2)\ \ldots\ N_k$ as $M\ N_1\ N_2\ \ldots\ N_k$.

Example 2.2 ($\lambda^{\to}$ terms)
For $\mathcal{B} = \{\mathsf{int}, \mathsf{bool}\}$ *and assuming the existence of special function symbols* $+, =, \geq, \ldots$ *and constants* $\mathsf{false}, \mathsf{true}, 0, 1, \ldots$ *the following are (syntactically) valid* $\lambda^{\to}$ *terms:* $0, \mathsf{true}, \lambda x : \mathsf{int}.\ x, \lambda x : \mathsf{bool}.\ x, \lambda (a\ b : \mathsf{int}).\ a > b, (\lambda x : \mathsf{int}.\ y)\ z, 0\ 1, \mathsf{true}\ \mathsf{false}, \mathsf{true}\ 1, f\ 1, g\ \mathsf{true}, f\ x\ y, f\ (g\ x), (\lambda x : \mathsf{int}.\ x + 1)\ 0, (\lambda x : \mathsf{int}.\ x > 0)\ \mathsf{true}, \ldots$

Definitions 2.1 and 2.2 respectively provide the syntactic rules to generate types and terms of $\lambda^{\to}$. Any type produced in this way is a valid $\lambda^{\to}$ type; however, there are some syntactically valid terms that do not make too much sense. For instance, one may wonder what is the intended meaning of 0 1 or $\lambda x : \mathsf{bool}.\ x + 1$. This question will be discussed next in § 2.1.1.

Definition 2.3 (Free and bound variables). Occurrences of a variable within the scope of a λ-abstraction that binds such variable are said to be *bound*. The *free* occurrences

of a variable in a λ-term are those which are not bound by any λ-abstraction. The set of *free variables* (more precisely, the set of variables occurring free) of a λ-term, M, is denoted as $\mathsf{FV}(M)$ and inductively defined as:

$$\begin{aligned} \mathsf{FV}(x) &= x \\ \mathsf{FV}(\lambda x.\ M) &= \mathsf{FV}(M) - \{x\} \\ \mathsf{FV}(M\ N) &= \mathsf{FV}(M) \cup \mathsf{FV}(N) \end{aligned}$$

We will use the standard mathematical notation $M[x \mapsto N]$ to denote the substitution of N for the free variable x in M. In order to avoid the undesired capture of free variables as in $(\lambda y : \tau.\ x)[x \mapsto y] \equiv \lambda y : \tau.\ y$ —notice that y were free and became bound, we will assume that every λ-abstraction in a term binds a different variable and that all the bound variables are different from the free ones[1]. Any λ-term can be transformed into this form, which is known as the Barendregt convention, by renaming all its abstractions with unique names.

Definition 2.4 (α-conversion). Also called α-renaming, consists of renaming bound variables in a term. The basic rule of α-conversion is $\lambda x : \tau.\ M \to_\alpha \lambda x_0 : \tau.\ M[x \mapsto x_0]$, where x_0 is a fresh name: e.g. $\lambda x : \mathsf{int}.\ 1 + x \to_\alpha \lambda y : \mathsf{int}.\ 1 + y$. We say that two terms M, N are α-equivalent, written as $M \equiv_\alpha N$, if and only if they can be shown syntactically equal modulo α-renaming.

Definition 2.5 (β-reduction). The β-reduction relation ($\to_\beta$) maps terms of the form $(\lambda x.\ M)N$ to $M[x \mapsto N]$, where a term $(\lambda x.\ M)N$ is called a *redex* (reducible expression). Thus $M \to_\beta M'$ holds if and only if M' is the result of reducing any of the redexes in M. A term M is said to be in *normal form*, denoted as $M\!\downarrow$, if it contains no redex. We use $\twoheadrightarrow_\beta$ to denote the reflexive and transitive closure of $\to_\beta$; and $\equiv_\beta$ to denote its reflexive, transitive and symmetric closure. β-reduction captures the idea of function application and evaluation.

2.1.1 Typing $\lambda^\to$ terms

A typing judgment is of the form $\Gamma \vdash M : \sigma$, meaning that M *has type* σ in the context Γ. We say that M is *typable* if and only if there exist Γ and σ such that $\Gamma \vdash M : \sigma$. We will also abbreviate $\forall\Gamma.\ \Gamma \vdash M : \sigma$ as $\vdash M : \sigma$. The set of typable terms is a proper

[1] Other presentations of λ-calculus prefer to define a *capture-avoiding* substitution algorithm.

subset of all $\lambda^{\to}$ terms. The goal of typing is to impose restrictions to terms in order to rule out specific classes of incorrect programs.

Definition 2.6 (Contexts ($\lambda^{\to}$)). The set of *typing contexts* C is a set of sets of the form

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n$$

with $x_1, \ldots, x_n \in \mathcal{V}$ different variables and $\sigma_1, \ldots, \sigma_n \in T$.

In the following, we will use Γ to denote arbitrary contexts, and assume that these are always well-formed to reduce the number of side-conditions in typing rules.

Definition 2.7 (Typability ($\lambda^{\to}$)). The $\lambda^{\to}$ typability relation $\vdash$ on $C \times E \times T$ is defined by the following typing rules:

$$\textsc{Var}\ \frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \textsc{Abs}\ \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.\ M : \sigma \to \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}$$

These rules basically guarantee that a function is always applied to arguments of the right type. For example, $(\lambda x : \mathsf{int}.\ 1 + x)\ \mathsf{true}$ is not a typable term, because true has type bool whereas $\lambda x : \mathsf{int}.\ 1 + x$ is a function that expects an argument of type int.

Definition 2.8 (Type Checking). For a given context Γ, term M and type σ the type-checking problem is to determine whether $\Gamma \vdash M : \sigma$ holds.

Definition 2.9 (Type Inference). For a given context Γ and term M the type-inference problem consist of finding a type σ such that $\Gamma \vdash M : \sigma$ holds.

Both type-checking and type-inference are decidable problems for $\lambda^{\to}$. Actually, it is straightforward to derive such algorithms from the typing rules provided above. Regarding our practical issue of allowing the programmer to omit type annotations, it turns out that both problems are also decidable for terms with no annotations at all — that is, where all abstractions are of the form $\lambda x.\ M$. The omission of type annotations in λ-abstractions constitutes a flavor known as *Curry-style* λ-calculus.

2.1.2 A $\lambda^{\to}$ type system

A type system based on $\lambda^{\to}$ defines a concrete set $\mathcal{B}$ of atomic types and usually extends the sort of terms with constants and built-in expressions. It is also possible to extend

the sort of types with extra built-in type constructors or even to add other kinds of orthogonal extensions such as algebraic data types. The specification language of the PVS system [22] is an example of how it is possible to define a powerful language on top of a simple formalism like $\lambda^{\to}$. For the sake of simplicity we shall discuss this class of type systems using a simple representative language with built-in integer and boolean expressions. Let int and bool, the type of integers and booleans respectively, be the only types in $\mathcal{B}$:

$$T ::= \mathsf{int} \mid \mathsf{bool} \mid T \to T$$

Then we may extend the set of terms with integer and boolean constants, a few convenient operators and *if-then-else* expressions; as in:

$$E ::= \mathcal{V} \mid \mathbb{Z} \mid \mathsf{true} \mid \mathsf{false} \mid (+) \mid (\leq) \mid \mathsf{if}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E \mid \lambda \mathcal{V} : T.\ E \mid E\ E$$

where $(+)$ and $(\leq)$ are the *plus* and *less-than* functions for integers. For convenience we will resort to infix notation to write terms like $(+)\ x\ y$ as $x + y$. These extensions also introduce the following typing rules:

$$\textsc{Int}\ \frac{}{\vdash n : \mathsf{int}}\ n \in \mathbb{Z} \qquad \textsc{Bool}\ \frac{}{\vdash b : \mathsf{bool}}\ b \in \{\mathsf{true}, \mathsf{false}\} \qquad \textsc{Plus}\ \frac{}{\vdash (+) : \mathsf{int} \to \mathsf{int} \to \mathsf{int}}$$

$$\textsc{Le}\ \frac{}{\vdash (\leq) : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}} \qquad \textsc{If}\ \frac{\Gamma \vdash C : \mathsf{bool} \qquad \Gamma \vdash N : \sigma \qquad \Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{if}\ C\ \mathsf{then}\ M\ \mathsf{else}\ N : \sigma}$$

This simple language is enough to write some programs of interest as shown below. Note however that it is not possible to define recursive functions: the type system ban them to force program termination, which makes $\lambda^{\to}$ logically consistent. Even though there are ways to support specific recursion patterns preserving program termination, most programming languages introduce built-in recursive definitions breaking this property. This question will be left for detailed discussion in a later section.

Example 2.3 (Well-typed $\lambda^{\to}$ terms)

The identity function for int: $\vdash (\lambda x : \mathsf{int}.\ x) : \mathsf{int} \to \mathsf{int}$

The double function: $\vdash (\lambda x : \mathsf{int}.\ x + x) : \mathsf{int} \to \mathsf{int}$

The sign function: $\vdash (\lambda x : \mathsf{int}.\ \mathsf{if}\ 0 \leq x\ \mathsf{then}\ (\mathsf{if}\ 1 \leq x\ \mathsf{then}\ 1\ \mathsf{else}\ 0)\ \mathsf{else}\ -1) : \mathsf{int} \to \mathsf{int}$

The not function: $\vdash (\lambda b : \mathsf{bool}.\ \mathsf{if}\ b\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}) : \mathsf{bool} \to \mathsf{bool}$

The and function: $\vdash (\lambda p : \mathsf{bool}.\ \lambda q : \mathsf{bool}.\ \mathsf{if}\ p\ \mathsf{then}\ q\ \mathsf{else}\ \mathsf{false}) : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$

A type system enforces a set of rules on the terms of the language, in order to satisfy a notion of correctness. In the case of a pure $\lambda^{\to}$ type system, the evaluation of a well-typed term is guaranteed to be error-free and terminating. The typechecker is a program that implements the typing rules of the type system and rejects any term for which there is no typing derivation —i.e. it is not typable.

Definition 2.10 (Soundness). A type system is said to be *sound* if every typable term is correct.

Definition 2.11 (Completeness). A type system is said to be *complete* if any correct term is also typable.

A type system is expected to be sound (it is not very useful otherwise), but they are usually incomplete. Soundness guarantees that well-typed programs *cannot go wrong* because of an error that the type system is supposed to detect. In this respect *types are specifications*, and the typing rules allow to check whether a term conforms to its specification. On the other hand, incompleteness means that there are correct programs that are however discarded by the type system, usually because the type system is not expressive enough. The following are examples of erroneous terms that would be discarded by the typechecker:

Example 2.4 (Ill-typed $\lambda^{\to}$ terms)

$\mathsf{true}\ 1$	$\nrightarrow$	`true is not a function.`
$(\lambda x : int.\ x + 1)\ \mathsf{true}$	$\nrightarrow$	λ-abstraction `expected argument of type int`
$\lambda x : int.\ \mathsf{if}\ x\ \mathsf{then}\ x + 1\ \mathsf{else}\ x + 2$	$\nrightarrow$	`'x' is used as bool but has type int`

2.1.3 Limitations

The main limitation of $\lambda^{\to}$ is its inability to handle polymorphic functions. Polymorphic functions are those that can be applied to values of different types. The prime

example of a polymorphic function is the identity function, which takes a term as argument and simply returns that term. Note that the behavior of the identity function does not depend on the concrete type of its argument, hence its polymorphic nature. Unfortunately, defining a function like that in our language is impossible; our best tentative definition would be:

$$\texttt{id} \;:=\; \lambda x : \sigma.\; x$$

but since we have no way to abstract terms over types, we are forced to substitute σ by picking a single type from T. We could define a family of identity functions for a series of types ($\texttt{id}_{\mathsf{int}}$, $\texttt{id}_{\mathsf{int}\to\mathsf{bool}}$, ...), but we have no way to define a single identity function for all types.

2.2 Polymorphic lambda calculus

The *polymorphic lambda calculus*, also known as System F and denoted as λ^2, introduces the notion of abstraction of types out of terms making possible the definition of *polymorphic* functions. More precisely, System F formalizes the notion of *parametric polymorphism*, where the behavior of a polymorphic function is completely independent of the concrete types that may take its type parameters. Polymorphism is a very important concept in software development since it enables software reuse.

Definition 2.12 (Types (λ^2)). Let $\mathcal{B}$ be a countable set of atomic types, and $\mathcal{U}$ be a countable infinite set whose members will be called *type variables*. The set T of types of λ^2 is defined as:

$$T ::= \mathcal{B} \mid \mathcal{U} \mid T \to T \mid \forall U.\; T$$

where the type-level $\forall$ constructs polymorphic types, also known as *universal types*. We use $\alpha, \beta, \gamma, \ldots$ to denote arbitrary type variables, and $\sigma, \tau, \ldots$ to denote arbitrary types. The $\forall$ operator binds any occurrence of a type variable within the body of the universal type, and it extends as far right as possible. For presentation purposes universal quantifiers can be group together, so $\forall\alpha_1.\ \ldots\ \forall\alpha_n.\ \tau$ can be written as $\forall\alpha_1\ \ldots\ \alpha_n.\ \tau$.

Example 2.5 (λ^2 types)
For $\mathcal{B} = \{\mathsf{int}, \mathsf{bool}\}$ *the following are (syntactically) valid* λ^2 *types:* $\mathsf{int}, \alpha, \mathsf{int} \to \mathsf{bool}, \alpha \to \alpha, \alpha \to \beta, \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \alpha \to \mathsf{int}, \forall\alpha.\ \alpha, \forall\alpha.\ \alpha \to \alpha, \forall\alpha.\ \alpha \to \alpha \to \mathsf{bool}, \forall\alpha\ \beta.\ \alpha \to \mathsf{int} \to \beta, \forall\alpha\ \beta.\ \alpha \to \beta \to \alpha, \ldots$

Definition 2.13 (Terms (λ^2)). Let $\mathcal{V}$ be a countable infinite set of term variables. The set E of terms of λ^2 is defined as:

$$E ::= \mathcal{V} \mid \lambda\mathcal{V} : T.\ E \mid E\ E \mid \Lambda\mathcal{U}.\ E \mid E\ T$$

Again, we will use $x, y, \ldots$ to denote arbitrary term variables, and $M, N, \ldots$ to denote arbitrary terms. A term of the form $\Lambda\alpha.\ M$ is called a *type abstraction*: it is a polymorphic term parametrized by an arbitrary type α, where α may occur within the term M as any other type does. As we do for λ-abstractions, we may also group consecutive Λ-abstractions together. A term of the form $M\ \sigma$ is called a *type application* (often *type instantiation*), and it allows the instantiation of a polymorphic term M with a type σ.

Example 2.6 (λ^2 terms)
For $\mathcal{B} = \{\mathsf{int}, \mathsf{bool}\}$ *and assuming the existence of special function symbols* $+, =, \geq, \ldots$ *and constants* $\mathsf{false}, \mathsf{true}, 0, 1, \ldots$ *the following are (syntactically) valid* λ^2 *terms:* $0, \mathsf{true}, \lambda x : \mathsf{int}.\ x, \Lambda\alpha.\ \lambda x : \alpha.\ x, \lambda x : \beta.\ \Lambda\alpha.\ x, \mathsf{true}\ 1, f\ \alpha\ x, g\ \mathsf{int}\ 1\ \mathsf{bool}\ \mathsf{false}, \ldots$

Definition 2.14 (Free and bound variables (λ^2)). The definition of free and bound variables for $\lambda^{\to}$ is naturally extended to λ^2. Analogously, we define the concepts of free and bound occurrences for type variables, which are bound by Λ-abstractions. The set of type variables occurring free within a term M is denoted $\mathsf{FTV}(M)$.

For the sake of simplicity, we assume that both λ- and Λ-abstractions follow the Barendregt convention, and that every type variable is bound by an outer $\forall$ or Λ. We write $M[\alpha \mapsto \tau]$ to mean the substitution of τ for the free type variable α in M.

Definition 2.15 (β-reduction (λ^2)). In λ^2 the relation $\to_\beta$ is extended to map terms of the form $(\Lambda\alpha.\ M)\ \sigma$ to $M[\alpha \mapsto \sigma]$. This new rule captures the instantiation of polymorphic terms, e.g. $(\Lambda\alpha.\ \lambda x : \alpha.\ x)\ \mathsf{int} \to_\beta \lambda x : \mathsf{int}.\ x$.

2.2.1 Typing λ^2 terms

A typing judgment is of the form $\Gamma \vdash M : \sigma$, meaning that M *has type* σ in the context Γ. Typing serves the same purpose in λ^2 as in $\lambda^{\to}$, but the language of types is more expressive and thus more terms are typable.

Definition 2.16 (Contexts (λ^2)). The notion of typing context is naturally extended to keep track of type variables. A context Γ is now a (possibly empty) set of the form:

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n, \alpha_1, \ldots, \alpha_m$$

with $x_1, \ldots, x_n \in \mathcal{V}$ different variables; $\sigma_1, \ldots, \sigma_n \in T$; and $\alpha_1, \ldots, \alpha_m \in \mathcal{U}$ different type variables.

In the same way as for $\lambda^{\to}$, we assume that every context Γ is well-formed to reduce the number of side-conditions in typing rules.

Definition 2.17 (Typability (λ^2)). Typing rules for λ^2 are those for $\lambda^{\to}$ extended with two new rules to handle polymorphic terms:

$$\textsc{Var}\ \frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \textsc{Abs}\ \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.\ M : \sigma \to \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}$$

$$\textsc{TyAbs}\ \frac{\Gamma, \alpha \vdash M : \sigma}{\Gamma \vdash \Lambda\alpha.\ M : \forall\alpha.\ \sigma} \qquad \textsc{TyApp}\ \frac{\Gamma \vdash M : \forall\alpha.\ \sigma}{\Gamma \vdash M\ \tau : \sigma[\alpha \mapsto \tau]}$$

Example 2.7 (Well-typed λ^2 terms)

The identity function: $\vdash (\Lambda\alpha.\ \lambda x : \alpha.\ x) : \forall\alpha.\ \alpha \to \alpha$

The identity function for σ*:* $\vdash (\Lambda\alpha.\ \lambda x : \alpha.\ x)\ \sigma : \sigma \to \sigma$

The constant function: $\vdash (\Lambda\alpha\beta.\ \lambda x : \alpha.\ \lambda y : \beta.\ x) : \forall\alpha\beta.\ \alpha \to \beta \to \alpha$

The composition function:

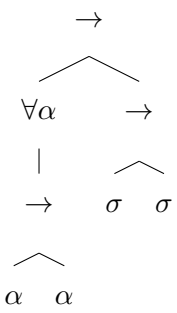
$\vdash (\Lambda\alpha\beta\gamma.\ \lambda f : \alpha \to \beta.\ \lambda g : \beta \to \gamma.\ \lambda x : \alpha.\ g(fx)) : \forall\alpha\beta\gamma.\ (\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma$

Both type-checking and type-inference are decidable problems for λ^2. However, for Curry-style λ^2, type-checking is decidable but type-inference is not.

Definition 2.18 (Rank of a type). For some fixed value k, a type has rank-k if in the tree representation of the type a $\forall$ quantifier does not appear to the left of more than $k-1$ arrows.

It is also possible to provide an inductive definition for the rank of a type [23], but we believe that the above fits better our needs. In other words, it says that a type has rank $k+1$ only if the domain of every arrow ($\rightarrow$) contained in it has rank k.

Example 2.8 (Rank-2 type)
The type $(\forall\alpha.\ \alpha \rightarrow \alpha) \rightarrow \sigma \rightarrow \sigma$ *has rank-2 since in its tree representation* $\forall\alpha$ *appears to the left of exactly one arrow.*



Definition 2.19 (Impredicative polymorphism). Also called first-class polymorphism, allows the instantiation of a type variable α with any type, including polymorphic types. System F is an impredicative system.

Example 2.9 (Impredicativity)
The identity (id) function, $\Lambda\alpha.\ \lambda x : \alpha.\ x$*, has type* $\forall\alpha.\ \alpha \rightarrow \alpha$*. Since* λ^2 *is impredicative, we can instantiate id with its own type: id* $(\forall\alpha.\ \alpha \rightarrow \alpha)$ *has type* $(\forall\alpha.\ \alpha \rightarrow \alpha) \rightarrow (\forall\alpha.\ \alpha \rightarrow \alpha)$*. Thus, we can apply id to itself: id* $(\forall\alpha.\ \alpha \rightarrow \alpha)$ *id, which reduces to id.*

Definition 2.20 (Predicative polymorphism). A predicate polymorphic system restricts type variables to range over monomorphic types only.

2.3 Damas-Milner type system

A Damas-Milner type system is a restriction of System F that syntactically restricts the set of types to be in *prenex normal form* (PNF). A type in PNF is of the form $\forall\overline{\alpha}.\ \tau$, consisting of a (possibly empty) string of quantified type variables ($\overline{\alpha}$) followed by a quantifier-free type (τ). In Damas-Milner, types of this form are also called *type schemes*, and they actually constitute a canonical representation of rank-1 polymorphic

types. Which made Damas-Milner type systems so popular was the possibility to infer the most general type (principal type) of a term with no type annotation at all. What is more, type inference can be performed almost in linear time, making it practically usable to type large programs.

2.3.1 Language

This system is going to be described using a representative functional language, which is still very close to the λ-calculus described in § 2.1. The language of types, which is shown in Figure 2.1, is similar to the one of System F. However, in order to enforce the restriction of types being in PNF, it makes a syntactic separation between monomorphic types (*monotypes*) and type schemes (*polytypes*). For simplicity we assume that int is the only basic type in our language. As usual, we use α, β, γ to range over type variables and $\overline{\alpha}$ to denote a string of zero or more type variables. Since we distinguish two classes of types, τ and υ will be used to range over monotypes, and σ to range over polytypes.

$$
\begin{array}{rrcl}
\text{Type variables } (\mathcal{U}) & \alpha, \beta, \gamma & & \\
\text{Basic types } (\mathcal{B}) & & ::= & \mathsf{int} \\
\text{Monotypes } (\mathcal{T}) & \tau, \upsilon & ::= & \mathsf{int} \mid \alpha \mid \tau_1 \to \tau_2 \\
\text{Polytypes } (\mathcal{S}) & \sigma & ::= & \forall \overline{\alpha}.\ \tau
\end{array}
$$

Figure 2.1: Syntax of types.

Note that a polytype is not necessarily an universally quantified type, since the string $\overline{\alpha}$ can be empty. Polytypes contain both monotypes and rank-1 polymorphic types in PNF. For convenience, we will sometimes abbreviate types of the form $\forall.\ \tau$ as τ, if there is no ambiguity.

The language of terms, which is shown in Figure 2.2, is a λ-calculus augmented with local (*let*) bindings. These let-bindings serve to associate a name with a term, allowing to introduce intermediate definitions (e.g. $\mathsf{let}\ x = y + y\ \mathsf{in}\ x \times x$), usually with the purpose of enabling *sharing*. In fact, let-bindings play a more important role in Damas-Milner system, since it is the only construct that allows polymorphic terms to be defined —as will be explained later in § 2.3.2.

It is worth noting that, even when Damas-Milner is a polymorphic type system, its

Term variables ($\mathcal{V}$)	x, y, z			
Integers ($\mathbb{Z}$)	i			
Terms (E)	t, u	$::=$	i	Literal
		$\mid$	x	Variable
		$\mid$	$\lambda x.\ t$	λ-abstraction
		$\mid$	$t\ u$	Application
		$\mid$	$\mathsf{let}\ x = u\ \mathsf{in}\ t$	Let-binding

Figure 2.2: Syntax of terms.

term language lacks System F's Λ-abstraction and type application. These constructs are implicit in Damas-Milner systems, leading to a term language that resembles the simply typed λ-calculus.

2.3.2 Type system

A typing judgment is of the form $\Gamma \vdash M : \sigma$, meaning that M *has type* σ in the context Γ. We will however abuse the notation writing $\Gamma \vdash M : \tau$ instead of $\Gamma \vdash M : \forall.\ \tau$. A typing context Γ keeps track of in-scope variables and their types:

$$\text{Type contexts } (C) \quad \Gamma \quad ::= \quad \varepsilon \mid \Gamma, x : \sigma$$

Recall from the previous presentation of System F that $\mathsf{FTV}(\sigma)$ stands for the *free type variables* of σ. The FTV function is extended to typing contexts in the obvious way: $\mathsf{FTV}(\Gamma) = \bigcup_{(x:\sigma)\in\Gamma} \mathsf{FTV}(\sigma)$. We also use the notation $\tau[\overline{\alpha \mapsto \upsilon}]$ to mean the substitution of type variables $\overline{\alpha}$ by monotypes $\overline{\upsilon}$ in the type τ.

Figure 2.3 presents the Damas-Milner type system. The very basic typing rules are not different from those of the simply typed λ-calculus. Notice that only variables introduced by let-bindings are allowed to have a polymorphic type, which is the reason why Damas-Milner polymorphism is often called *Let-polymorphism*. The interesting rules of this system are the Gen and Inst ones. Rule Gen allows the type of a term to be generalized by quantifying over type variables, thus playing the role of TyAbs in System F. The side condition $\overline{\alpha} \notin \mathsf{FTV}(\Gamma)$ prevents undesired capturing of type variables that were already quantified. The purpose of rule Inst is to instantiate a polymorphic term, thus playing the role of TyApp in System F. This rule makes

$$\boxed{\Gamma \vdash t : \sigma}$$

$$\textsc{Int}\ \frac{}{\Gamma \vdash i : \mathsf{int}}\ i \in \mathbb{Z} \qquad \textsc{Var}\ \frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\ t : \tau_1 \to \tau_2}$$

$$\textsc{App}\ \frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2} \qquad \textsc{Let}\ \frac{\Gamma \vdash u : \sigma \qquad \Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ t : \tau}$$

$$\textsc{Gen}\ \frac{\overline{\alpha} \notin \mathsf{FTV}(\Gamma) \qquad \Gamma \vdash t : \tau}{\Gamma \vdash t : \forall \overline{\alpha}.\ \tau} \qquad \textsc{Inst}\ \frac{\Gamma \vdash t : \forall \overline{\alpha}.\ \tau}{\Gamma \vdash t : \tau[\overline{\alpha \mapsto \upsilon}]}$$

Figure 2.3: Declarative Damas-Milner type system.

clear that Damas-Milner is a predicate system, since terms are instantiated by monotypes. We could think of these rules as implicitly introducing a Λ-abstraction or a type application, respectively.

Example 2.10 (Damas-Milner polymorphism)

$$\dfrac{\dfrac{\dfrac{x : \alpha \vdash x : \alpha}{\vdash \lambda x.\ x : \alpha \to \alpha}\ \textsc{Lam}}{\vdash \lambda x.\ x : \forall \alpha.\ \alpha \to \alpha}\ \textsc{Gen} \qquad \dfrac{\dfrac{\dfrac{}{id : \forall \alpha.\ \alpha \to \alpha \vdash id : \forall \alpha.\ \alpha \to \alpha}\ \textsc{Var}}{id : \forall \alpha.\ \alpha \to \alpha \vdash id : \mathit{int} \to \mathit{int}}\ \textsc{Inst} \qquad \dfrac{}{\vdash 1 : \mathit{int}}\ \textsc{Int}}{id : \forall \alpha.\ \alpha \to \alpha \vdash id\ 1 : \mathit{int}}\ \textsc{App}}{\vdash \mathit{let}\ id = \lambda x.\ x\ \mathit{in}\ id\ 1 : \mathit{int}}\ \textsc{Let}$$

We first presented the typing rules of the Damas-Milner system in a declarative way. Such presentation is more elegant and succinct, but it is difficult to derive an inference algorithm from it. For instance, the Gen and Inst rules have the same syntactic form in their premise as in their conclusion, so they can be applied virtually anywhere. Thus, there is no clear or unique way to derive a typing judgment, and typing becomes a process of searching for a valid derivation. Because of this, it is a good practice to provide both a declarative and a *syntax-directed* version of a type system.

An alternative, syntax-directed, form of the Damas-Milner type system is shown in

$$\boxed{\Gamma \vdash t : \tau}$$

$$\textsc{Int}\ \frac{}{\Gamma \vdash i : \mathsf{int}}\ i \in \mathbb{Z} \qquad \textsc{Var}\ \frac{\vdash^{\mathrm{inst}} \sigma \leq \tau}{\Gamma, x : \sigma \vdash x : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\ t : \tau_1 \to \tau_2}$$

$$\textsc{App}\ \frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2} \qquad \textsc{Let}\ \frac{\Gamma \vdash^{\mathrm{poly}} u : \sigma \qquad \Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ t : \tau}$$

$$\boxed{\Gamma \vdash^{\mathrm{poly}} t : \sigma} \qquad \boxed{\vdash^{\mathrm{inst}} \sigma \leq \tau}$$

$$\textsc{Gen}\ \frac{\overline{\alpha} = \mathsf{FTV}(\tau) - \mathsf{FTV}(\Gamma) \qquad \Gamma \vdash t : \tau}{\Gamma \vdash^{\mathrm{poly}} t : \forall \overline{\alpha}.\ \tau} \qquad \textsc{Inst}\ \frac{}{\vdash^{\mathrm{inst}} \forall \overline{\alpha}.\ \tau \leq \tau[\overline{\alpha \mapsto v}]}$$

Figure 2.4: Syntax-directed Damas-Milner type system.

Figure 2.4. We say that it is syntax-directed because each rule has a distinct syntactic form in its conclusion. Therefore, a typing derivation can be driven by the syntax of the term, instead of having to search for a valid derivation. The main judgment now takes the form $\Gamma \vdash t : \tau$, and generalization and instantiation of types take place when specified by the syntax of the term.

We introduce two new judgments to handle generalization and instantiation. The judgment $\Gamma \vdash^{\mathrm{poly}} t : \sigma$ means that a polytype σ can be inferred for t in Γ. The rule Gen specifies how to generalize the type τ of a term t, by quantifying over its free type variables not occurring in Γ (to avoid premature generalization). The only place where generalization is allowed to occur is in the right hand side of a let-binding. On the other hand, the judgment $\vdash^{\mathrm{inst}} \sigma \leq \tau$ means that the quantified type variables of σ can be instantiated to give τ. Any occurrence of a variable is implicitly instantiated as specified by rule Var.

2.3.3 Inference algorithm

The syntax-directed form of the Damas-Milner type system could be regarded as an algorithm, but it still leaves a few things unspecified. Some rules refer to types that appear out of nowhere, for instance, it is totally unspecified where the τ_1 comes from in rule Abs. We found the same situation in rule Inst, where a σ-type is instantiated with $\overline{\upsilon}$ types that are also unknown. Such types have to be guessed in some way, while satisfying the conditions imposed by applications: for $t\ u$, t must have a function type whose domain matches the type of u. For example, given $\Gamma \vdash^{\text{poly}} t : \forall\alpha.\ \alpha \to \mathsf{int}$ and $\Gamma \vdash u : \mathsf{int}$, t has to be instantiated as $\mathsf{int} \to \mathsf{int}$ so that it can be applied to u. Any other instantiation of t would not satisfy the premise of rule App.

The way to guess such types effectively is to resort on *unification* [24]. Very generally speaking, unification is an algorithmic process that finds the most general solution for a set of equations. For example, the solution of $\{f(x,y) = f(a,y),\ g(y) = g(b)\}$ is $\{x \mapsto a,\ y \mapsto b\}$; where x, y are variables and a, b are constants. In our case, we will handle equations that specify constraints about types. For this purpose, we introduce a new kind of variable, named *meta type variables* and written as $\mathring{\tau}$. A meta type variable represents a temporary place-holder for an as-yet-unknown monotype. Whenever a type has to be guessed we will introduce a fresh $\mathring{\tau}$-variable, and the concrete value of that type will be later determined according to the restrictions imposed to that type, namely by the applications occurring within the program.

To summarize, the inference algorithm that it is going to be presented here distinguishes two classes of type variables: *a*) *bound type variables* (α, β, γ), that are used to build polymorphic types, such as $\forall\alpha.\ \alpha \to \alpha$; and *b*) *meta type variables* $(\mathring{\tau})$, that represent temporary place-holders for unknown monotypes. Meta type variables cannot be quantified and they are not part of the language of types, they are just an internal concept of the type inference method itself.

We define $\mathsf{MTV}(\sigma)$ as the set of all meta type variables that occur in σ, and this definition is extended to contexts Γ in the obvious way —as for FTV. We will use the notation $\tau_1 \sim \tau_2$ to mean the unification of types τ_1 and τ_2. How exactly this unification takes place is not important now, one could think of $\sim$ as recording equations that are later solved by an underlying unification process.

$$
\begin{array}{rcl}
\mathsf{infer}_\sigma(\Gamma, t) & := & gen(\Gamma, \tau) \\
& \mathsf{where} & \tau := \mathsf{infer}_\tau(\Gamma, t)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{infer}_\tau(\Gamma, i) & := & \mathsf{int} \\
\mathsf{infer}_\tau(\Gamma \cup \{x : \sigma\}, x) & := & inst(\sigma) \\
\mathsf{infer}_\tau(\Gamma, \lambda x.\ t) & := & \mathring{\tau}_1 \to \tau_2 \\
& \mathsf{where} & \mathring{\tau}_1 \ \mathit{fresh} \\
& & \tau_2 := \mathsf{infer}_\tau(\Gamma \cup \{x : \mathring{\tau}_1\}, t) \\
\mathsf{infer}_\tau(\Gamma, t\ u) & := & \mathring{\tau}_2 \\
& \mathsf{where} & \tau' := \mathsf{infer}_\tau(\Gamma, t) \\
& & \tau_1 := \mathsf{infer}_\tau(\Gamma, u) \\
& & \mathring{\tau}_2 \ \mathit{fresh} \\
& & \tau' \sim \tau_1 \to \mathring{\tau}_2 \\
\mathsf{infer}_\tau(\Gamma, \mathsf{let}\ x = u\ \mathsf{in}\ t) & := & \mathsf{infer}_\tau(\Gamma \cup \{x : \sigma\}, t) \\
& \mathsf{where} & \sigma := \mathsf{infer}_\sigma(\Gamma, u)
\end{array}
$$

$$
\begin{array}{rcll}
gen(\Gamma, \tau) & := & \forall \overline{\alpha}.\ \tau[\overline{\mathring{\tau} \mapsto \alpha}] & \qquad inst(\forall \overline{\alpha}.\ \tau) := \tau[\overline{\alpha \mapsto \mathring{\tau}}] \ \mathsf{where}\ \mathring{\tau}\ \mathit{fresh} \\
& \mathsf{where} & \overline{\mathring{\tau}} := \mathsf{MTV}(\tau) - \mathsf{MTV}(\Gamma) & \\
& & \overline{\alpha}\ \mathit{fresh} &
\end{array}
$$

Figure 2.5: Damas-Milner type inference algorithm.

The Damas-Milner type inference algorithm, originally proposed by Damas and Milner and known as Algorithm $\mathcal{W}$, is described in Figure 2.5[2]. The entry point of the inference method is the function infer_σ which plays the role of $\vdash^{\mathrm{poly}}$. This function simply generalizes the τ-type inferred for a term by applying the rule Gen, which became the function *gen*. Notice that the function infer_τ, which implements the rules for $\vdash$, must not be called directly since it return types containing $\mathring{\tau}$-variables.

[2]Our presentation is slightly different, but the algorithm is the same.

Unification

Figure 5.16 shows our reference unification algorithm. In this presentation $\sim$ is a function that, along with the types to be unified, additionally takes a substitution ς as input. The result of $\tau_1 \overset{\varsigma}{\sim} \tau_2$ is either *a*) a new substitution ς' such that $\tau_1\varsigma' \equiv \tau_2$; or *b*) a *failure*, in such a case the types cannot be unified and that situation is reported as a typing error.

$$
\begin{array}{rcll}
\mathsf{int} \overset{\varsigma}{\sim} \mathsf{int} & \mapsto & \varsigma & \\
\tau_1 \to \tau_2 \overset{\varsigma}{\sim} \tau_1' \to \tau_2' & \mapsto & \varsigma \cup \tau_1 \overset{\varsigma}{\sim} \tau_1' \cup \tau_2 \overset{\varsigma}{\sim} \tau_2' & \\
\mathring{\tau} \overset{\varsigma}{\sim} \mathring{\tau} & \mapsto & \varsigma & \\
\mathring{\tau}_1 \overset{\varsigma}{\sim} \tau_2 & \mapsto & \varsigma(\mathring{\tau}_1) \overset{\varsigma}{\sim} \tau_2 & \text{if } \mathring{\tau}_1 \in dom(\varsigma) \\
\mathring{\tau}_1 \overset{\varsigma}{\sim} \tau_2 & \mapsto & \varsigma \cup \{\mathring{\tau}_1 \mapsto \tau_2\} & \text{if } \mathring{\tau}_1 \notin dom(\varsigma) \\
\tau_1 \overset{\varsigma}{\sim} \mathring{\tau}_2 & \mapsto & \mathring{\tau}_2 \overset{\varsigma}{\sim} \tau_1 & \\
\tau_1 \overset{\varsigma}{\sim} \tau_2 & \mapsto & \mathsf{fail} &
\end{array}
$$

Figure 2.6: Unification of types.

This algorithm differs from the traditional presentation of Robinson's unification, which takes a set of equations and returns a substitution. This version can be used to build a substitution incrementally, by feeding it with restrictions as they appear during typing. Both approaches are equivalent, but this presentation turns out to be more convenient for our purposes.

Chapter 3

Approaches to Correct Software Development

This chapter provides an overview of the state of the art approaches to correct software development. By "approach" we mean a paradigm to undertake the challenge of writing specifications and programs that implement them, independently of the particular method that may be used for verifying that a program meets its specification. There is no standard classification of such approaches in the program verification literature, so we are presenting our own.

First, we describe the use of theorem provers to formalize properties about programs in § 3.1. Then we describe the use of programming languages equipped with powerful type systems, that allow to express (and enforce) rich specifications about programs through typing, in § 3.2 and § 3.3. Finally, we introduce the well-known discipline of contract programming in § 3.4. We deliberately start with the more formal but heavier approaches, to end with the more practical and lightweight ones.

3.1 Theorem provers

The use of theorem provers to formalize program specifications is probably the most well-established approach for developing correct software, especially in academic settings. Despite we refer to them as "theorem provers", the systems considered here are targeted for software and hardware verification, and can be described more precisely

as specification systems with theorem proving capabilities.

Actually, there are two fundamental ways to address the development of correct software using theorem provers. First, the specification language of the theorem prover can be used as a programming language for developing (a model of) the target software. In this way the program is coded and verified using the theorem prover. A program developed in this way has to be *extracted* as a program in some programming language of choice in order to be used in production. This approach is of great theoretical interest, but it is not very useful in practice for many reasons.

In practice it is more common to write programs as usual, and then embed into the theorem prover those parts of the program that have to be verified. There are two (major) degrees of embedding: a *shallow embedding* consists of a syntactic translation of the program to the specification language of the theorem prover; whereas in a *deep embedding* the programming language is treated as a data type and its semantics are given by an interpreter function. A shallow embedding is usually easier to verify, but it may be difficult to capture the semantics of the programming language; on the other hand, a deep embedding captures the semantics precisely but often makes verification harder [25].

Next, three theorem provers are presented: PVS, Isabelle/HOL and Coq. PVS and Coq are described here (mainly) because this work is highly inspired in PVS's predicate subtyping and in Coq's Program environment. We decided to present Isabelle/HOL as well because it is the preferred choice for verifying Haskell programs. Nonetheless, the approach described here can be applied to (virtually) any theorem prover.

3.1.1 PVS

PVS [15, 26], which stands for *Prototype Verification System*, is a system for the development and analysis of formal specifications. PVS specifications are not intended to be executable: they are used to describe *what* is to be computed, not necessarily *how*. Even though PVS is not meant to be used for programming, it shares some features with programming languages such as recursive definitions and modules, and it can be used to describe algorithms to some extent. The PVS specification language [22] is (essentially) a classical typed higher-order logic, extended with predicate subtypes and dependent types.

Predicate subtypes [16] are inspired in the interpretation of types as sets of values, and in the set-comprehension notation used in matemathics for describing a set by stating the properties that its members must satisfy. A predicate subtype $\{x : T \mid P\}$ defines the subset of individuals x from T satisfying P. This feature, together with dependent types, enhances the expressiveness of the language, allowing to write very precise types. One of the immediate benefits of predicate subtypes is the possibility of defining partial functions by reflecting domain restrictions in function types. For example, the type of real division can be defined so the denominator is constrained to be nonzero, and then typechecking prevents any division by zero.

Since the predicate used in defining a predicate subtype is arbitrary, typechecking becomes undecidable. In PVS there is firm distinction between conventional typechecking, which is performed algorithmically; and checking the of uses of predicate subtyping, that may lead to proof obligations called *type correctness conditions* (TCCs). Typechecking is considered complete when all TCCs are discharged, which in principle should be done by the user with the assistance of the PVS prover.

PVS owns a versatile and highly automated proof system. It provides an interactive theorem prover, but also a symbolic model checker and a tool for randomized testing [27]. As a proof assistant, PVS combines aspects from both interactive and automatic theorem provers. It supports a rich specification logic and requires human guidance at the higher levels of proof development, but it provides a high degree of automation for the lower levels. PVS proofs are developed in sequent calculus, which is well-suited for automated deduction, and it comes with a number of built-in decision procedures and high-level proof strategies (and the ability to define your own). A large fragment of the PVS specification language is actually execuble. PVS supports the evaluation of executable specifications and even to generate pieces of code from them.

3.1.2 Isabelle/HOL

Isabelle/HOL [28] is another proof assistant for classical typed higher-order logic. It is the most widespread instance of Isabelle [29], a generic framework for interactive theorem proving. Despite Isabelle/HOL was primarily designed for describing abstract models of systems, it is well-suited for writing functional programs and verifying properties about them. Its declarative proof language, called Isar, is versatile and incorporates

a broad range of automated proof methods. Isar offers commands for automatic deduction based on resolution and term rewriting, allows for randomized testing of properties, and also provides built-in connections to first-order provers and SMT solvers. As in PVS, the user is expected to guide the higher level steps of the proof, while low-level steps are handled by automated methods. The programming features of Isabelle/HOL, together with its high degree of automation, have made it a popular choice for developing correct software.

Haskabelle

Haskabelle [30] is a tool, distributed with Isabelle, to convert Haskell source files into Isabelle/HOL theories. Haskabelle is intended to support the verification of properties about Haskell programs through a shallow embedding into Isabelle/HOL. The tool tries a one-to-one translation whenever possible, but there exist many features of both Haskell types and terms that cannot be translated directly (or in any way). In order to guide the translation one can manually specify mappings from Haskell definitions to Isabelle/HOL definitions. This approach may be interesting and of practical usefulness, but it is inevitably restricted to a subset of Haskell, since HOL cannot capture all the semantics of Haskell — for instance, general recursion and laziness.

3.1.3 Coq

The Coq system [31, 32] is especially desgined to write formal specifications, and to develop programs in accordance with them. The specification language of Coq, called Gallina, is a dependently typed functional programming language based on the *Calculus of Inductive Constructions* (CIC), a derivative of the *Calculus of Constructions* (CoC) developed by Thierry Coquand. Using the so-called *Curry-Howard isomorphism* [33], which states the proofs-as-programs and propositions-as-types interpretation; programs, properties and proofs are specified using the same language. Propositions are just types in a *universe* called Prop, and a proof for a proposition P is just a term of type P, thus proofs are constructive. Coq is also an interactive proof assistant where proofs can be build using a language of tactics known as L_{tac}. Tactics are a convenient way to construct proofs more concisely, and provide some degree of

automation.

Russell language

Russell [17] is a language developed by Matthieu Sozeau to support practical programming with dependent types in Coq. Russell allows writing strong specifications (types) like `forall (x:nat) {y:nat|y>0}, {q:nat & {r:nat|x=y*q+r}}` while providing purely algorithmic implementations as in a regular functional programming language. In pure Coq the implementation of the above specification is fairly complicated due to the mixture of both algorithmic and proof code. In Russell, however, the proof parts of the Coq term lead to proof obligations that are discarded separately using Coq proof machinery. The typing technique employed by Russell to produce proof obligations is derived from the *Predicate subtyping* mechanism of PVS.

3.2 Dependently typed programming

Dependent types, or more precisely types depending on terms, bring the expressive power to use types as complete specifications of programs behavior. The usefulness of dependent types for developing reliable software has been widely stated [34], and many argue that programming languages will evolve in this direction. However, the design of dependently-typed languages suitable for mainstream programming is still an open research topic. Roughly, the problem is that terms occurring in types have to be executed as part of typechecking, and in a mainstream language such execution may lead to non-termination, run-time errors, etc.

Here are presented a few representative attempts to introduce dependent typing into mainstream programming. We should distinguish approaches that support full dependent types, like Ynot and Agda; from those that offer a restricted form of dependent types like Guru, DML and ATS.

3.2.1 Ynot

Ynot [35] is a library that extends the Coq proof assistant to support writing, reasoning and extraction of dependently-typed programs with side-effects. Ynot allows to write imperative programs involving features such as non-termination, mutable references

and exceptions. These features are added to Coq in a controlled way, so programs may be impure, while proofs remain pure and logically consistent. This library also includes tactics to automate much of the reasoning required to discharge the proof goals about these imperative constructs.

3.2.2 Agda

Agda [36, 37] is a dependently typed programming language based on a predicative extension of Martin-Löf's Type Theory. It can also be seen as a proof assistant, following the propositions-as-types interpretation it can be used as a system for developing constructive proofs. Agda is intended to be a practical programming language by including features such as general recursion, I/O operations, and a *foreign function interface* (FFI) for calling Haskell functions from Agda. Some of the above features may actually introduce logical inconsistencies, being the user responsible for its careful use. Even though Agda involves proof construction as part of dependently-typed programming, it is not a system well-suited for developing complex proofs. It could fairly be considered a very naive proof assistant, for instance, it does not have any separate (tactic) language for assisting proofs.

3.2.3 Guru

Guru [38] is a functional programming language, largely inspired by Coq, which integrates dependently-typed programming in the presence of partial functions and effectful computations. In Guru, polymorphism and dependent types present some restrictions in comparison with Coq or Agda. It enforces a separation between *specificational* and *non-specificational* data, so the latter cannot depend on the first. Data types may depend on (proven) *well-founded* terms, which are then marked as specificational in the data type constructors. Like Agda, Guru is focused in facilitating proof construction derived from dependently typed programming, however it is not well-suited for complicated theorem proving. It does not follow the Curry-Howard isomorphism but it has separate languages for writing proofs and program terms, as well as for formulas and types. Effectful computations like mutable references or I/O are supported without breaking the language purity thanks to a specific type system extension known as

linear types.

3.2.4 Dependent ML

Dependent ML [39] (DML) is an extension of ML with a restricted form of dependent types, where data types can be indexed by naturals. In contrast with traditional dependent type systems, type index terms belong to a separate language $\mathcal{L}$, and are not regular terms of the language. Type expressions are allowed to quantify over type indexes and to impose linear constraints on them. This allows many interesting program properties, such as memory safety and termination, to be expressed in the type system of DML that enforces them at compile-time. Constraints arising from typechecking are automatically solved using linear integer programming, without any user intervention.

3.2.5 ATS

ATS [40] is a ML-like dependently typed programming language which generalizes the approach followed by DML. ATS distinguishes a language of dynamic (program) terms, that may be divergent or effectful; and a language of static terms, that are well-founded and on which types are allowed to depend. Types for the static language are called *sorts*, and the type of a dynamic term is a static term of the special sort *type*. ATS also defines a language of proof terms, whose types are static terms of sort *prop* called propositions. The programmer is then allowed to define new sorts and (inductive) propositions in order to specify properties of interest within types. In this way ATS provides dependent programming features that are comparable to those offered by Guru. The restricted form of types indexed by integers of DML, for which linear constraints are automatically solved, is also incorporated into ATS through a special sort *int*.

3.3 Lightweight dependently-typed programming

The term *lightweight dependently-typed programming* [41] refers to the (smart) use of the type-level features of a (non-dependently typed) programming language to approximate dependent types. Roughly, one interprets types as propositions and makes use of type parameters to encode constraints over terms, which are thereby enforced during typechecking. In principle, this would be better described as *type hackery* rather

than as a specific paradigm to approach correct software development. However, during the past few years a number of extensions to the Damas-Milner type system have been developed with the primary purpose of increasing its ability to reflect properties about terms within types. This new paradigm, which is argued as the future of functional programming by many authors, is exemplified by the Ωmega [42] programming language.

The use of type parameters with the only purpose of constraining a type to statically enforce programming invariants is the basis of lightweight dependently-typed programming. As an illustrative example, let's consider a data type `XMLDoc s` representing a XML document, where the parameter `s` does not occur in the domain of any of the data constructors of `XMLDoc`. We say that `s` is a *phantom* type [43, 44] because it distinguishes terms that share the same representation. The parameter `s` is intended to classify documents by its validation status, for which we define two dummy types `Ok` and `Valid` that will act as values for `s`. A well-formed document will be assigned the type `XMLDoc Ok`, which when validated will be converted into a document of type `XMLDoc Valid`. If a function requires a valid document as argument then it simply specifies that such argument must have type `XMLDoc Valid`. On the other hand, functions that work for any kind of document can be specified as polymorphic functions.

3.3.1 Haskell

The Haskell functional programming language seems to be evolving towards this paradigm of programming, and many features that were previously sketched in Ωmega are now included in Haskell as experimental type extensions. These new features have been exploited in many different and impressive ways [45, 46]. A very illustrative and highly practical example of this approach to program correctness can be found in the Darcs version control system (VCS), where lightweight dependent types are used to enforce non-trivial invariants during patch manipulation [47].

3.4 Contract programming

Contract programming, also known as *programming by contract*, is an approach to software development in which programs are annotated with preconditions, postconditions

and invariants that constitute precise functional specifications. These specifications are referred to as *contracts*, in accordance with a conceptual metaphor with the conditions and obligations of business contracts. Contract programming is a natural application of the *Design by Contract* approach to software design [48].

From the functional programming perspective, preconditions and postconditions apply to functions while invariants apply to data types. A precondition states a predicate that the arguments of a function must hold, and a postcondition states a relation between the arguments and the value of the function for those arguments. The pair of pre- and postconditions of a function constitutes its contract or specification. For instance, a function *div* that computes the real division of two arguments a (numerator) and b (denominator) can be fully specified by a precondition $b \neq 0$ and a postcondition $a = b \times div(a, b)$. A data type invariant is a predicate that every inhabitant of the data type must satisfy.

Contract programming naturally leads to *proof obligations* that guarantee that the contracts are respected: e.g. every application $f(t)$ requires to prove that the term t satisfies f's precondition. The validity of the proof obligations entails the correctness of the annotated program. At the time this approach was introduced (1991), it was unfeasible to perform static checking of contracts, and proof obligations were (optionally) translated into run-time checks. However, recent advances in automated reasoning made possible to analyze programs statically to detect contract violations.

3.4.1 Spec#

Spec# [49] is a programming system for developing reliable object-oriented software through the discipline of programming by contract. The Spec# programming language is an extension of Microsoft's C# with non-null types, checked exceptions and contract annotations. The compiler statically enforces non-null types and can convert contracts into run-time checks. The system also includes a program verifier that can be used to find contract violations or, in some cases, to prove the absence of errors. Spec# is a research project being developed at Microsoft Research, and it is bringing new features to the .NET platform, such as the non-null types of C# 3.0.

3.4.2 Contracts for functional programming

Contract programming has had relative success within the imperative/object-oriented community. There exist (thirty party) tools or libraries that add support for this programming discipline to any of the most popular imperative/OO languages including Ada, C, C++, C#, Java, JavaScript, Python, etc. On the functional programming side, however, there is simply no tool for any of the mainstream functional programming languages, namely Haskell and Caml. One can only mention a few research prototypes that have tried to fill this gap.

Hoare logic for functional programming Pangolin is a ML-like programming language with contracts where proof obligations are derived using a variant of Hoare logic for call-by-value functional programming [12]. The language itself is just a proof-of-concept for demonstration purposes.

Static contract checking for Haskell On the Haskell side there was an experiment to add support for contract programming to the GHC Haskell compiler [14]. This approach differs substantially from traditional implementations of contracts. First, functions and data types are annotated with a language of contracts that reminds a language of dependent types, even though contracts are not treated like types. Second, there is no generation of proof obligations, but instead contracts are used to instrument the program that is then analyzed using symbolic execution. Unfortunately, for some reason this work was never taken seriously enough to make it part of GHC.

Chapter 4

The $\mathcal{H}_{\text{spec}}$ Programming Language

$\mathcal{H}_{\text{SPEC}}$ is a functional programming language based on the syntax and semantics of Haskell, and equipped with a type system that is also a rich specification language. The type system of $\mathcal{H}_{\text{SPEC}}$ naturally accomodates the paradigm of programming by contract through predicate subtypes and dependent function types, in the spirit of PVS. $\mathcal{H}_{\text{SPEC}}$ is a language designed to support development of reliable software in mainstream settings. This chapter describes the syntax and informal sematantics of $\mathcal{H}_{\text{SPEC}}$, and discusses the most important features and design choices. The style of our presentation is deliberately close to the one of the Haskell 98's language report [50].

A formal presentation of $\mathcal{H}_{\text{SPEC}}$ semantics could be very complicated and it is definitely out of the scope of this work — that could be a thesis by itself. Instead, we will provide an informal description of $\mathcal{H}_{\text{SPEC}}$ semantics by translating its high-level constructs into simpler, and mostly standard, constructs of λ-calculus. Such basic constructs form a small subset of $\mathcal{H}_{\text{SPEC}}$ that we call the $\mathcal{H}_{\text{SPEC}}$ *kernel*. This kernel is not formally specified, but it is a slightly sugared polymorphic λ-calculus extended with predicate subtypes. The translation rules to $\mathcal{H}_{\text{SPEC}}$ kernel, together with textual explanations and reference typechecker and interpreter implementations, describe the semantics precisely enough for most practical purposes.

4.1 Notation

The $\mathcal{H}_{\text{SPEC}}$ grammar presented here follows much the same notations and structure that were used to describe Haskell 98's grammar [50]. This is a natural choice given that $\mathcal{H}_{\text{SPEC}}$ is essentially a subset of Haskell extended with predicate types *à la* PVS. Among other benefits, it becomes fairly easy to compare the syntax of both languages.

Haskell's syntax is rather complicated and it is tricky to express using standard BNF notation. Thus, a few extra notational conventions, mostly standard EBNF [51, 52], are used for presenting syntax —see Table 4.1.

$[pattern]$	optional
$\{pattern\}$	zero or more repetitions
$pattern^{+}$	one or more repetitions
$(pattern)$	grouping
$pat_1 \mid pat_2$	choice
$pat_{<pat'>}$	difference, i.e. $[\![pat]\!] - [\![pat']\!]$
`'symbol'`	terminal syntax (in typewriter font)

Table 4.1: Extra notational conventions.

Moreover, we will introduce families of non-terminals indexed by precedence levels; where precedence is written as a superscript, e.g. $\langle exp^0 \rangle$. We will write i for a precedence-level variable ranging from 0 to 9, such that a pattern like $\langle exp^i \rangle$ stands for the *choice* between the 10 patterns resulting of the 10 substitutions for i. In addition to a precedence level, non-terminals referring to operators have another superscript to indicate associativity: l, r, or n for left-, right-, and non-associativity respectively. We will write a for an associativity variable varying over $\{l, r, n\}$.

4.2 Lexical syntax

For convenience the lexical structure is also specified using EBNF notation. As in Haskell, the rule of *maximum munch* is used for disambiguation, so the lexical analysis always takes the longest possible match. For example, the input "==" will always be read as a `'=='` token instead of two `'='`. Any *whitespace* character (including spaces,

tabulators and newlines) is considered a lexeme delimiter. In contrast with Haskell, only ASCII characters are allowed.

4.2.1 Lexemes

⟨*small*⟩ ::= `'a'` | ... | `'z'` | `'_'`

⟨*large*⟩ ::= `'A'` | ... | `'Z'`

⟨*digit*⟩ ::= `'0'` | ... | `'9'`

Literals The only numeric literals are natural numbers given in decimal notation.

⟨*natural*⟩ ::= ⟨*digit*⟩$^+$

Identifiers An identifier consists of a letter followed by zero or more letters, digits, underscores, and single quotes. Identifiers are lexically distinguished into two namespaces: *variable identifiers*, that start with a lower-case letter; and *constructor identifiers*, that start with an upper-case letter.

⟨*varid*⟩ ::= (⟨*small*⟩ {⟨*small*⟩ | ⟨*large*⟩ | ⟨*digit*⟩ | `''`}) $_{\langle reservedid \rangle}$

⟨*conid*⟩ ::= ⟨*large*⟩ {⟨*small*⟩ | ⟨*large*⟩ | ⟨*digit*⟩ | `''`}

⟨*reservedid*⟩ ::= `'case'` | `'data'` | `'else'` | `'exists'` | `'forall'` | `'if'`
| `'in'` | `'lemma'` | `'let'` | `'module'` | `'of'`
| `'then'` | `'theorem'` | `'type'` | `'where'` | `'_'`

There are five kinds of names in $\mathcal{H}_{\text{SPEC}}$: those for *variables* and *type variables*, those for *constructors* and *type constructors*, and *module names*. The first two kinds are *variable identifiers* while the other three are *constructor identifiers*.

⟨*var*⟩ ::= ⟨*varid*⟩

⟨*tyvar*⟩ ::= ⟨*varid*⟩

⟨*con*⟩ ::= ⟨*conid*⟩

⟨*tycon*⟩ ::= ⟨*conid*⟩

⟨*modid*⟩ ::= ⟨*conid*⟩

Operators For simplicity we defined a considerable amount of built-in operators, in opposition to the Haskell approach which is to define as few as possible. This design choice simplifies the implementation of a compiler, and the potential benefits of avoiding built-in operators are not really important here.

⟨*reservedop*⟩ ::= ‘.’ | ‘..’ | ‘:’ | ‘::’ | ‘=’ | ‘\’ | ‘|’ | ‘->’ | ‘@’
| ‘~’ | ‘||’ | ‘&&’ | ‘==>’ | ‘<=>’
| ‘+’ | ‘-’ | ‘/’ | ‘%’ | ‘^’
| ‘==’ | ‘/=’ | ‘>=’ | ‘>’ | ‘<=’ | ‘<’

Comments A single-line comment starts with a sequence of two or more consecutive dashes, e.g. ‘--’, and extends to the following line. A multi-line comment begins with ‘{-’ and ends with ‘-}’, the comment itself is not lexically analyzed but nested multi-line comments are allowed.

4.2.2 Layout

As Haskell, $\mathcal{H}_{\text{SPEC}}$ also permits the omission of braces and semicolons by using *layout* to establish a separation between declarations. Both layout-sensitive and layout-insensitive styles can be used alone or freely mixed. The interested reader is invited to consult the *Haskell 98 Language Report* [50] for a detailed description of these layout rules.

4.3 Types

Types classify terms and serve to avoid programming mistakes. A term can be rejected by the typechecker either because it is ill-typed, or because its actual type does not match the expected type for the term in the context where it is being used. The paradigm of programming by contract is integrated into the type system of $\mathcal{H}_{\text{SPEC}}$ through a family of *contract types*: predicate subtypes, dependent function types, and dependent tuple types. These new type constructs allow types to express rich functional specifications. This section describes the grammar and informal semantics of $\mathcal{H}_{\text{SPEC}}$ types.

⟨*polytype*⟩ ::= `'forall'` ⟨*tyvar*⟩$^+$ `'.'` ⟨*type*⟩ (type scheme)
| ⟨*type*⟩

⟨*type*⟩ ::= ⟨*btype*⟩ [`'->'` ⟨*type*⟩] (function type)

⟨*btype*⟩ ::= [⟨*btype*⟩] ⟨*atype*⟩ (type application)

⟨*atype*⟩ ::= ⟨*gtycon*⟩ (global type constructor)
| ⟨*tyvar*⟩ (type variable)
| `'('` ⟨*tupdom*⟩$_1$ `','` ... `','` ⟨*tupdom*⟩$_k$ `')'` $k \geq 2$ (tuple type)
| `'['` ⟨*type*⟩ `']'` (list type)
| `'{'` ⟨*apat*⟩ `':'` ⟨*type*⟩ [`'|'` ⟨*exp*⟩] `'}'` (predicate subtype)
| `'('` ⟨*type*⟩ `')'` (parenthesized type)

⟨*tupdom*⟩ ::= ⟨*type*⟩
| ⟨*apat*⟩ `':'` ⟨*type*⟩ [`'|'` ⟨*exp*⟩]

⟨*gtycon*⟩ ::= ⟨*tycon*⟩ (type constructor)
| `'()'` (unit type)

4.3.1 Type constructors

Just as data values are built using data constructors, types are built from type constructors. Unlike Haskell, every occurrence of a type constructor must be saturated. Regular type constructors are identifiers (beginning with an uppercase letter) introduced by `type` and `data` declarations, although $\mathcal{H}_{\text{SPEC}}$ provides special syntax for some built-in ones. Namely, $\mathcal{H}_{\text{SPEC}}$ has the following built-in type constructors:

- `Bool` (booleans), defined as `data Bool = False | True`.
- `Int` (integers), is a special built-in type.
- `Nat` (naturals), defined as `type Nat = {n:Int | n >= 0}`.
- `()` (unit), could be thought as `data () = ()`.

4.3.2 Type schemes

$\mathcal{H}_{\text{SPEC}}$ supports classical Damas-Milner predicative polymorphism—see § 2.3, like most typed functional programming languages. An universally quantified type has the form `forall` a_1 ... a_n`.` *type*, where a_1 ... a_n are distinct type variables scoping over *type*.

Unlike standard presentations of Damas-Milner polymorphism, we introduce the `'forall'` keyword and make the universal quantifier $\forall$ explicit. The type $\forall\alpha.\ \alpha \to \alpha$ is written in Haskell as `a -> a`, but as `forall a. a -> a` in $\mathcal{H}_{\text{SPEC}}$. Besides we believe that this is a good practice, this decision was also motivated by future plans to support arbitrary-rank polymorphism.

4.3.3 Predicate subtypes

Predicate (sub)types correspond to the concept of *subset type* in type theory. This construct allows a type to be refined by means of a predicate, which can be interpreted as a subtyping mechanism. In the traditional presentation, a type $\{x\texttt{:}A\ \texttt{|}\ P\}$ denotes the set of values v of type A such that $P[x \mapsto v]$ holds.

$\mathcal{H}_{\text{SPEC}}$ offers a sugared version of predicate subtypes combined with pattern-matching, which also subsumes the classical form. A predicate subtype has the general form $\{pat\texttt{:}A\ \texttt{|}\ P\}$, denoting the set of values v of type A matching the pattern *pat* such that `let` *pat* `=` v `in` P holds. For convenience, the following syntax $\{pat\texttt{:}A\}$ is also allowed, where the proposition P is omitted and assumed `True`. Types of this form are called *pattern types*, and describe the subset of values of a type matching a given pattern.

It is possible to translate predicate types of the form $\{pat\texttt{:}A\ \texttt{|}\ P\}$ to the more standard form without pattern-matching, by resorting to an internal *matches* expression —see § 4.4.10:

$$\{pat\texttt{:}A\ \texttt{|}\ P\} \equiv \{x\texttt{:}A\ \texttt{|}\ x\ \texttt{matches}\ pat\ \texttt{->}\ P\} \quad x\ \textit{fresh}$$

4.3.4 Dependent functions

Dependent function types correspond to the concept of first-order *dependent product type* in type theory. It is a generalization of the function space by allowing the range

to depend on the domain. A dependent function of type $\{x\,\mathtt{:}\,A\}$ `->` B maps arguments v of type A to values of type $B(v)$, we say that $B(x)$ is a type that is indexed by x.

$\mathcal{H}_{\text{SPEC}}$ offers a sugared version of function types that permits the syntax of predicate subtypes for specifying the domain of a dependent function. Thus, a dependent function type has the general form $\{pat\,\mathtt{:}\,A\ \mathtt{|}\ P\}$ `->` B, where variables bound by pat —$\mathcal{V}(pat)$— not only scope over P but also over B. The special forms $\{pat\,\mathtt{:}\,A\}$ `->` B and A `->` B are also allowed, the later describes a non-dependent function type and stands for $\{\_\,\mathtt{:}\,A\}$ `->` B.

The translation of the sugared version of function types provided by $\mathcal{H}_{\text{SPEC}}$, to the classical form used in type theory, requires the use of a special form of substitution $A[pat \overset{\texttt{match}}{\leftarrow} e]$ which binds $\mathcal{V}(x)$ to e by pattern-matching —see § 4.3.6:

$$\{pat\,\mathtt{:}\,A\ \mathtt{|}\ P\}\ \texttt{->}\ B \equiv \{x\,\mathtt{:}\,\{pat\,\mathtt{:}\,A\ \mathtt{|}\ P\}\}\ \texttt{->}\ B[pat \overset{\texttt{match}}{\leftarrow} x] \quad x\ \textit{fresh}$$

4.3.5 Dependent tuples

Dependent tuple types correspond to the concept of first-order *dependent sum types* of type theory. It is a generalization of n-tuple types by allowing the ith component to depend on the previous jth components for $0 \leq j < i$. The classical syntax is $(x_1\,\mathtt{:}\,A_1\,\mathtt{,}\,\ldots\,\mathtt{,}\,x_{n-1}\,\mathtt{:}\,A_{n-1}\,\mathtt{,}\,A_n)$, where each x_i scopes over A_j for $i < j \leq n$. Analogously to dependent function types, each A_i is a typed indexed by all the x_j for $0 \leq j < i$.

$\mathcal{H}_{\text{SPEC}}$ offers a convenient syntactic form that incorporates the syntax of predicate subtypes. The general form of a dependent tuple type is $(pat_1\,\mathtt{:}\,A_1\,\mathtt{|}\,P_1\,\mathtt{,}\,\ldots\,\mathtt{,}\,pat_n\,\mathtt{:}\,A_n\,\mathtt{|}\,P_n)$, where $\mathcal{V}(pat_i)$ scope over A_j for $i < j \leq k$, and over P_j for $i \leq j \leq k$. The special syntactic forms $(pat_1\,\mathtt{:}\,A_1\,\mathtt{,}\,\ldots\,\mathtt{,}\,pat_n\,\mathtt{:}\,A_n)$ and $(A_1\,\mathtt{,}\,\ldots\,\mathtt{,}\,A_n)$ are allowed as well, the later denotes a non-dependent tuple type and stands for $(\_\,\mathtt{:}\,A_1\,\mathtt{,}\,\ldots\,\mathtt{,}\,\_\,\mathtt{:}\,A_n)$.

The translation of the sugared version of tuple types provided by $\mathcal{H}_{\text{SPEC}}$, to the classical form of type theory, is similar to the translation of dependent function types:

$$(pat_1\,\mathtt{:}\,A_1\,\mathtt{|}\,P_1\,\mathtt{,}\,pat_2\,\mathtt{:}\,A_2\,\mathtt{|}\,P_2) \equiv (x\,\mathtt{:}\,\{pat_1\,\mathtt{:}\,A_1\ \mathtt{|}\ P_1\}\,\mathtt{,}\ \{pat_2\,\mathtt{:}\,A_2\ \mathtt{|}\ P_2\}[pat_1 \overset{\texttt{match}}{\leftarrow} x]) \quad x\ \textit{fresh}$$

For the sake of simplicity we present this translation only for the case of 2-tuples. This translation rule can be inductively extended to n-tuples by thinking of the dependent tuple type $(pat_1\,\mathtt{:}\,A_1\,\mathtt{|}\,P_1\,\mathtt{,}\,\ldots\,\mathtt{,}\,pat_n\,\mathtt{:}\,A_n\,\mathtt{|}\,P_n)$ as $(pat_1\,\mathtt{:}\,A_1\,\mathtt{|}\,P_1\,\mathtt{,}\,(pat_2\,\mathtt{:}\,A_2\,\mathtt{|}\,P_2\,\mathtt{,}\,\ldots\,\mathtt{,}\,pat_n\,\mathtt{:}\,A_n\,\mathtt{|}\,P_n))$.

4.3.6 Match-substitution

For a type A indexed by a pattern pat, the *match*-substitution $A[pat \overset{\texttt{match}}{\leftarrow} e]$ binds $\mathcal{V}(pat)$ to e by pattern-matching. It traverses the type A replacing type predicates P within A by: e `matches` pat `->` P.

Definition 4.1 (Match-substitution).

$$
\begin{aligned}
a[pat \overset{\texttt{match}}{\leftarrow} e] &= a \\
(T\ t_1 \ldots t_n)[pat \overset{\texttt{match}}{\leftarrow} e] &= T\ t_1[pat \overset{\texttt{match}}{\leftarrow} e]\ \ldots\ t_n[pat \overset{\texttt{match}}{\leftarrow} e] \\
\{p{:}A\ \texttt{|}\ P\}[pat \overset{\texttt{match}}{\leftarrow} e] &= \{p{:}A[pat \overset{\texttt{match}}{\leftarrow} e]\ \texttt{|}\ e\ \texttt{matches}\ pat\ \texttt{->}\ P\} \\
(\{p{:}A\ \texttt{|}\ P\}\ \texttt{->}\ \mathrm{B})[pat \overset{\texttt{match}}{\leftarrow} e] &= \{p{:}A[pat \overset{\texttt{match}}{\leftarrow} e]\ \texttt{|}\ e\ \texttt{matches}\ pat\ \texttt{->}\ P\}\ \texttt{->}\ B[pat \overset{\texttt{match}}{\leftarrow} e] \\
(\texttt{forall}\ \bar{a}.\ t)[pat \overset{\texttt{match}}{\leftarrow} e] &= \texttt{forall}\ \bar{a}.\ t[pat \overset{\texttt{match}}{\leftarrow} e] \\
(p_1{:}A_1 \texttt{|} P_1\texttt{,}\ \ldots\texttt{,}p_n{:}A_n \texttt{|} P_n)[pat \overset{\texttt{match}}{\leftarrow} e] &= \\
\multicolumn{2}{r}{(p_1{:}A_1[pat \overset{\texttt{match}}{\leftarrow} e] \texttt{|} e\ \texttt{matches}\ pat\ \texttt{->}\ P_1\texttt{,}\ldots\texttt{,}\ pat_n{:}A_n[pat \overset{\texttt{match}}{\leftarrow} e] \texttt{|} e\ \texttt{matches}\ pat\ \texttt{->}\ P_n\texttt{)}}
\end{aligned}
$$

4.4 Expressions

This section describes the grammar and informal semantics of $\mathcal{H}_{\text{SPEC}}$ expressions. It is worth noting that some of the translation rules for expressions may not preserve the type inference properties.

$\langle literal \rangle$ ::= $\langle natural \rangle$

$\langle exp \rangle$	::= $\langle exp^0 \rangle$ `':'` $\langle type \rangle$		(type annotation)
	| $\langle exp^0 \rangle$		
$\langle exp^i \rangle$	::= $\langle exp^{i+1} \rangle$ $[\langle op^{(n,i)} \rangle\ \langle exp^{i+1} \rangle]$		
	| $\langle lexp^i \rangle$		
	| $\langle rexp^i \rangle$		
$\langle exp^i \rangle$	::= $(\langle lexp^i \rangle\ \mid\ \langle exp^{i+1} \rangle)\ \langle op^{(l,i)} \rangle\ \langle exp^{i+1} \rangle$		
$\langle lexp^6 \rangle$	::= `'-'` $\langle exp^7 \rangle$		(arithmetic negation)
	| `'~'` $\langle exp^7 \rangle$		(logical negation)
$\langle rexp^i \rangle$	::= $\langle exp^{i+1} \rangle\ \langle op^{(r,i)} \rangle\ (\langle rexp^i \rangle\ \mid\ \langle exp^{i+1} \rangle)$		
$\langle exp^{10} \rangle$	::= `'\'` $\langle apat \rangle_1 \ldots \langle apat \rangle_n$ `'->'` $\langle exp \rangle$	$n \geq 1$	(lambda abstraction)
	| `'let'` $\langle decls \rangle$ `'in'` $\langle exp \rangle$		(let expression)

		‘`if`’ ⟨*exp*⟩ ‘`then`’ ⟨*exp*⟩ ‘`else`’ ⟨*exp*⟩ (conditional)
		‘`if`’ ⟨*gdalts*⟩ (conditional)
		‘`case`’ ⟨*exp*⟩ ‘`of`’ ‘`{`’ ⟨*alts*⟩ ‘`}`’ (case expression)
		⟨*quant*⟩ ⟨*var*⟩$^+$ ‘`,`’ ⟨*exp*⟩ (quantified proposition)
		⟨*fexp*⟩
⟨*quant*⟩	::=	‘`forall`’
		‘`exists`’
⟨*fexp*⟩	::=	[⟨*fexp*⟩] ⟨*aexp*⟩ (function application)
⟨*aexp*⟩	::=	⟨*var*⟩ (variable)
		⟨*gcon*⟩ (general constructor)
		⟨*literal*⟩
		‘`(`’ ⟨*exp*⟩ ‘`)`’ (parenthesized expression)
		‘`(`’ ⟨*exp*⟩$_1$ ‘`,`’ ... ‘`,`’ ⟨*exp*⟩$_k$ ‘`)`’ $k\geq 2$ (k-tuple)
		‘`[`’ ⟨*exp*⟩$_1$ ‘`,`’ ... ‘`,`’ ⟨*exp*⟩$_k$ ‘`]`’ $k\geq 1$ (list)
		‘`[`’ ⟨*exp*⟩$_1$ [‘`,`’ ⟨*exp*⟩$_2$] ‘`..`’ ⟨*exp*⟩$_3$ ‘`]`’ (arithmetic sequence)
		‘`(`’ ⟨*op*⟩ ‘`)`’ (operator as a function)
		‘`(`’ ⟨exp^{i+1}⟩ ⟨$op^{(a,i)}$⟩ ‘`)`’ (left section)
		‘`(`’ ⟨$lexp^{i}$⟩ ⟨$op^{(l,i)}$⟩ ‘`)`’ (left section)
		‘`(`’ ⟨$op^{(a,i)}$⟩ ⟨exp^{i+1}⟩ ‘`)`’ (right section)
		‘`(`’ ⟨$op^{(r,i)}$⟩ ⟨$lexp^{i}$⟩ ‘`)`’ (right section)

4.4.1 Applications and lambda abtractions

Function application is written $e_1\ e_2$, with e_i arbitrary expressions. Application has the highest precedence and associates to the left, so an expression like $((f\ x)\ y) + 1$ can be written as $f\ x\ y + 1$. Data constructors are treated as regular functions, hence partial applications of data constructors are also allowed.

Lambda abstractions are written $\backslash\ p_1 \ldots p_n$ `->` e, where p_i are *patterns* and e an expression. The set of patterns p_i must be linear, that is, no variable may appear more than once in the set. This sugared form of λ-abstraction combined with pattern-

matching can be reduced to a traditional λ-abstraction by moving pattern-matching to a case expression:

$$\backslash\ \ p_1 \ldots p_n \texttt{ -> } e \equiv \backslash\ x_1 \ldots x_n \texttt{ -> case } (x_1, \ldots, x_n) \texttt{ of } (p_1, \ldots, p_n) \texttt{ -> } e,\ \ x_i \textit{ fresh}$$

4.4.2 Operators

A term of the form $e_1\ op\ e_2$ denotes the infix application of a binary operator op to expressions e_1 and e_2. There are also two special prefix operators: the unary minus `'-'`, and the logical *not* `'~'`. These prefix operators can only be applied in prefix form as $\texttt{-}e$ and $\sim e$ respectively. On the contrary, binary operators have alternative syntactic forms that are very useful in practice.

A binary operator op can be enclosed within parentheses, `(`op`)`, which allows treating op as an ordinary curryfied function.

$$\texttt{(}op\texttt{)} \equiv \backslash\ e_1\ e_2 \texttt{ -> } e_1\ op\ e_2$$

Sections are written as `(`e_1 op`)` or `(`op e_2`)`. Sections are a convenient syntactic form of partial application of binary operators, e.g. `(+1)` is the successor function.

$$\texttt{(}e_1\ \ op\texttt{)} \equiv \backslash\ x \texttt{ -> } e_1\ op\ x$$

$$\texttt{(}op\ \ e_2\texttt{)} \equiv \backslash\ x \texttt{ -> } x\ op\ e_2$$

4.4.3 Conditionals

A conditional expression has the form `if |` g_1 `->` e_1 $\ldots$ `|` g_n `->` e_n where $n \geq 1$, and `|` g_i `->` e_i are called *guarded alternatives* —see § 4.6. We use g_i to mean a *guard expression*, that is either a boolean expression or the reserved keyword `'else'`. The `else` guard can only appear in the last guarded alternative, and in this context it stands for $\bigwedge_{i \in [1,n)} \neg g_i$. Execution semantics for conditional expressions are defined in the natural manner: `if |` g_1 `->` e_1 $\ldots$ `|` g_n `->` e_n evaluates to e_k iff g_k evaluates to `True` and g_i evaluates to `False` for $i \neq k$.

$\mathcal{H}_{\text{SPEC}}$ imposes two additional constraints to the set of guards $g_1, \ldots, g_n$ of a conditional expression. First, these guards must satisfy a *completeness* property, so that $g_1 \vee \ldots \vee g_n$ must hold. This implies that for every possible evaluation of a conditional

expression it is guaranteed to exist a guard g_k that evaluates to `True`. The second constraint to be satisfied is the *disjointness* of the guards, so that $g_i \Rightarrow \bigwedge_j^{i \neq j} \neg g_j$ must hold for every guard g_i. Disjointness implies that there is no overlapping between the guards, and therefore only one guard g_i can evaluate to `True` during a given execution.

The traditional *if-then-else* expressions are also supported by $\mathcal{H}_{\text{SPEC}}$, and they take the form `if` e_1 `then` e_2 `else` e_3. The translation of *if-then-else* expressions to general conditional expressions is straightforward:

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \equiv \texttt{if | } e_1 \texttt{ -> } e_2 \texttt{ | else -> } e_3$$

Note that if-then-else expressions trivially satisfy both completeness and disjointness.

4.4.4 Lists

Lists are inductively defined as being either the empty list `[]`, or a *cons*-list of the form $e_1\texttt{::}e_2$, where e_1 is an expression and es is a list of elements of the same type than e. The data constructors `[]` and `::` are reserved and part of the language syntax. In particular, `::` is a right-associative operator with precedence level 5. Additionally, it is possible to write lists as $\texttt{[}e_1\texttt{,...,}e_n\texttt{]}$ where $n \geq 1$ and every e_i must have the same type.

$$\texttt{[}e_1\texttt{,...,}e_n\texttt{]} \equiv e_1\texttt{::...::}e_n\texttt{::[]}$$

4.4.5 Tuples

Tuples are written $\texttt{(}e_1\texttt{,...,}e_n\texttt{)}$ and may be of arbitrary length $n \geq 2$. Tuples has to be built-in into the language because the $\mathcal{H}_{\text{SPEC}}$'s type system is not expressive enough to define dependent tuple types as regular inductive data types.

4.4.6 Unit

The *unit expression*, written `()`, is the only inhabitant of the *unit type*, written `()` as well.

4.4.7 Parenthesized expressions

The form $\texttt{(}e\texttt{)}$ is simply a parenthesized expression, and is equivalent to e.

$$\texttt{(}e\texttt{)} \equiv e$$

4.4.8 Arithmetic sequences

The *arithmetic sequence* `[`e_1`,`e_2`..`e_3`]` denotes a list of integer values from e_1 up to e_3, with step $e_2 - e_1$. The form `[`e_1`..`e_3`]` is equivalent but assuming a step of one.

$$\texttt{[}e_1\texttt{..}e_3\texttt{]} \equiv \texttt{[}e_1\texttt{,}e_1\texttt{+1..}e_3\texttt{]}$$

4.4.9 Let expressions

A *let expression* has the general form `let` $\{d_1\texttt{;...;}d_n\}$ `in` e and introduces a nested, lexically-scoped list of declarations. Mutually recursive bindings are not allowed in $\mathcal{H}_{\text{SPEC}}$, and therefore a let expression introducing n declarations can be decomposed into n nested simpler let expressions introducing a single declaration.

$$\texttt{let}\ \{d_1\texttt{;...;}d_n\}\ \texttt{in}\ e \equiv \texttt{let}\ d_1\ \texttt{in}\ \ldots\ \texttt{let}\ d_n\ \texttt{in}\ e$$

4.4.10 Case expressions

A case expression has the general form `case` e `of` $\{p_1\ \ \texttt{match}_1\texttt{;...;}p_n\ \ \texttt{match}_n\}$ where p_i are patterns and each $match_i$ has the general form:

```
| g_i1 -> e_i1
...
| g_im_i -> e_im_i
where decls_i
```

Each alternative p_i $match_i$ consists of a pattern p_i and a match, which in turn consists of a sequence of *guarded expressions* followed by a set of bindings $decls_i$. Variables bound by p_i scope over $match_i$, and the (optional) bindings $decls_i$ scope over all the guards and expressions. The guards of a match are subject to the same constraints than in a conditional expression —see § 4.4.3. In fact, these guards and the *where* clause is just syntax sugar and, following translation rules given in § 4.6, any case expression can be reduced to a simpler form: `case` e `of` { p_1 `->` e_1`;...;`p_n `->` e_n }. A case expression is evaluated by matching the expression e against the case alternatives. If e matches the pattern p_k then the whole case expression is reduced to e_k.

The set of patterns of a case expression has to obey analogous constraints than those imposed to the set of guards of conditional expressions. First, the set of patterns

must be exhaustive, which corresponds to *completeness* and guarantees the absence of run-time errors due to non-exhaustive patterns. Second, the set of patterns must be *uniform*, which corresponds to *disjointness* and guarantees that there is no overlapping between the set of terms matched by each of the patterns.

Logical case expressions

We introduce a variation of case expressions, that we call *logical case-expressions*, that are targeted for simplifying the internal handling of logical formulas involving pattern-matching constraints. These expressions have the general form $\texttt{caseP}_b\ e\ \texttt{of}\ pat\ \texttt{->}\ P$ where $b \in \{\texttt{True}, \texttt{False}\}$. If e matches pat then this expression acts as a case expression and evaluates to P, otherwise it evaluates to b. A `caseP` expression is just a mono-alternative case-expression with a default value. Desugaring a logical case-expression into a case expression involves the generation of a set of $\overline{p}$ patterns to complete pattern-matching with respect to pat:

$$\texttt{caseP}_b\ e\ \texttt{of}\ pat\ \texttt{->}\ P \equiv \texttt{case}\ e\ \texttt{of}\ \{\ pat\ \texttt{->}\ P\texttt{;}p_1\ \texttt{->}\ \texttt{b;}\ldots\texttt{;}p_n\ \texttt{->}\ \texttt{b}\ \}$$

For aesthetic reasons we write $\texttt{caseP}_{\texttt{False}}\ e\ \texttt{of}\ pat\ \texttt{->}\ P$ as $e\ \texttt{matches}\ pat\ \texttt{->}\ P$, and $\texttt{caseP}_{\texttt{True}}\ e\ \texttt{of}\ pat\ \texttt{->}\ P$ as $\texttt{if}\ e\ \texttt{matches}\ pat\ \texttt{->}\ P$.

4.4.11 Type signatures

Expressions with *type-signatures* have the form $e\texttt{:}t$, where e is an expression and t is a type. Type signatures are used to type an expression explicitly and are particularly useful to state properties that the typechecker cannot infer by itself. The value of the expression is just that of e.

4.5 Patterns

Patterns are used to specify a subset of the terms of a given type by their shape. Traditionally patterns are used in case expressions, but in $\mathcal{H}_{\text{SPEC}}$ they can also appear in lambda abstractions, binding declarations or even in types. Nevertheless, any use of patterns in $\mathcal{H}_{\text{SPEC}}$ can be reduced to case expressions while preserving dynamic semantics —it is also possible to preserve static semantics by adding explicit type signatures.

$$\langle pat \rangle \ ::= \langle pat^0 \rangle$$

$$\begin{array}{lll}
\langle pat^i \rangle & ::= \langle pat^{i+1} \rangle & \\
& \mid \texttt{`-'}\ \langle natural \rangle & \text{(negative literal)} \\
& \mid \langle rpat^i \rangle &
\end{array}$$

$$\langle rpat^i \rangle ::= \langle pat^{i+1} \rangle\ \texttt{`::'}\ (\langle rpat^i \rangle \mid \langle pat^{i+1} \rangle)$$

$$\begin{array}{lll}
\langle pat^{10} \rangle & ::= \langle apat \rangle & \\
& \mid \langle gcon \rangle\ \langle apat \rangle^+ &
\end{array}$$

$$\begin{array}{lll}
\langle apat \rangle & ::= \langle var \rangle\ [\texttt{`@'}\ \langle apat \rangle] & \text{(\textit{as} pattern)} \\
& \mid \langle gcon \rangle & \\
& \mid \langle literal \rangle & \\
& \mid \texttt{`_'} & \text{(wildcard)} \\
& \mid \texttt{`('}\ \langle pat \rangle\ \texttt{`)'} & \text{(parenthesized pattern)} \\
& \mid \texttt{`('}\ \langle pat \rangle_1\ \texttt{`,'}\ \ldots\ \texttt{`,'}\ \langle pat \rangle_n\ \texttt{`)'} \quad n \geq 2 & \text{(tuple pattern)} \\
& \mid \texttt{`['}\ \langle pat \rangle_1\ \texttt{`,'}\ \ldots\ \texttt{`,'}\ \langle pat \rangle_n\ \texttt{`]'} \quad n \geq 1 & \text{(list pattern)}
\end{array}$$

It is not possible to match against a partially-applied constructor, so the arity of a constructor in a pattern must match the number of sub-patterns associated with it. Moreover, all patterns must be *linear*, that is, no variable can appear more than once in a pattern. The linearity constraint also applies for sequences of patterns $p_1\ \ldots\ p_n$ as in lambda abstractions and bindings declarations.

As-patterns Patterns of the form $x\texttt{@}p$ are called *as-patterns*, and bind the value matched by p to the variable x.

$$\texttt{case}\ e\ \texttt{of}\ \{\ldots;x\texttt{@}p_k\ \texttt{->}\ e_k;\ldots\} \equiv \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in case}\ x\ \texttt{of}\ \{\ldots;p_k\ \texttt{->}\ e_k;\ldots\}$$

Wildcard patterns A pattern of the form _ is a *wildcard*, matching with everything without doing any binding. It is useful when some part of a pattern is not referenced on the right-hand side. A wildcard pattern can always be replaced by a fresh variable.

$$\texttt{\_ ->}\ e_k \equiv x\ \texttt{->}\ e_k \qquad x\ \textit{fresh}$$

Parenthesized patterns A pattern (p) is semantically equivalent to p.

$$(p) \equiv p$$

List patterns A pattern $[p_1, \ldots, p_n]$ matches a list of length n whose elements also match the sub-patterns p_i from left to right.

$$[p_1, \ldots, p_n] \equiv p_1 :: \ldots :: p_n :: []$$

4.5.1 Semantics of pattern matching

Patterns are matched against *values*. Attempting to match a pattern may *fail*; or it may *succeed*, returning a binding for each variable in the pattern. Pattern matching proceeds from left to right, and outside to inside, according to the following equations:

1. Matching the pattern x against a value v always succeeds and binds x to v.

$$match(x, v) = \{x \mapsto v\}$$

2. Matching the wildcard pattern _ against any value always succeeds, and no binding is done.

$$match(\_, v) = \emptyset$$

3. Matching a literal pattern succeeds iff the value is equals to the literal itself.

$$\begin{aligned} match(i, i) &= \emptyset \\ match(i, j) &= fail \quad \text{where } i \neq j \end{aligned}$$

4. Matching the pattern $con\ p_1 \ldots p_n$ against a value $con\ v_1 \ldots v_n$ is performed by matching sub-patterns p_i against values v_i from left to right. If all matches succeed then the overall match succeed; but the first failing matching causes the overall match to fail. If the value is build with a different constructor than *con* then the match fails.

$$\begin{aligned} match(con\ p_1 \ldots p_n, con\ v_1 \ldots v_n) &= \bigcup_{i \in [1,n]} match(p_i, v_i) \\ match(con\ p_1 \ldots p_n, con'\ v_1 \ldots v_m) &= fail \quad \text{where } con \neq con' \end{aligned}$$

5. Matching a tuple pattern $(p_1, \ldots, p_n)$ against a tuple of the same length $(v_1, \ldots, v_n)$ is performed by matching sub-patterns p_i against values v_i from left to right. If all matches succeed then the overall match succeed; but the first failing matching causes the overall match to fail.

$$match((p_1, \ldots, p_n), (v_1, \ldots, v_n)) = \bigcup_{i \in [1,n]} match(p_i, v_i)$$

6. Matching an as-pattern $x\texttt{@}p$ against a value v is the result of matching p against v, augmented with the binding of x to v. If the match of p against v fails then so does the overall match.

$$match(x\texttt{@}p, v) = match(p, v) \cup \{x \mapsto v\}$$

4.6 Alternatives

Alternatives are the basic blocks for control flow constructs: *case* and *if* expressions.

$\langle alts \rangle$ ::= $\langle alt \rangle_1$ ‘;’ ... ‘;’ $\langle alt \rangle_n$ $n \geq 1$

$\langle alt \rangle$ ::= $\langle pat \rangle$ $\langle match \rangle$

$\langle match \rangle$ ::= ‘->’ $\langle exp \rangle$ [‘where’ $\langle decls \rangle$]
| $\langle gdalts \rangle$ [‘where’ $\langle decls \rangle$]

$\langle gdalts \rangle$ ::= $\{\langle gdalt \rangle\}^+$ [$\langle egdalt \rangle$]
| $\{\langle gdalt \rangle\}$ $\langle egdalt \rangle$

$\langle gdalt \rangle$ ::= $\langle gd \rangle$ ‘->’ $\langle exp \rangle$

$\langle gd \rangle$ ::= ‘|’ $\langle exp^0 \rangle$

$\langle egdalt \rangle$ ::= ‘|’ ‘else’ ‘->’ $\langle exp \rangle$

A case expression consists of a set of alternatives specifying the different control flow paths. Each alternative alt_i has the general form:

p_i | g_{i1} -> e_{i1}
...
| g_{im} -> e_{im}
where $decls_i$

The list of guarded alternatives `|` g_{ij} `->` e_{ij} can be translated into a conditional expression and moved to the right-hand side, leading to a non-guarded alternative:

p_i `->` `if` `|` g_{i1} `->` e_{i1}
`...`
`|` g_{im} `->` e_{im}
`where` $decls_i$

Finally, given a non-guarded alternative of the form:

p_i `->` e_i
`where` $decls_i$

It is possible to move the set of declarations introduced by the where-clause to the right-hand side of the alternative as a let-expression.

$$p_i \texttt{ -> let } decls_i \texttt{ in } e_i$$

Therefore any case expression can be translated into a more basic form in which every alternative has the simpler form p_i `->` e_i.

4.7 Declarations and bindings

Declarations serve to introduce new types, value bindings, and also to declare logical facts such as theorems and lemmas.

⟨*topdecl*⟩	::=	`'type'` ⟨*simpletype*⟩ `'='` ⟨*type*⟩	(type synonym)
	|	`'data'` ⟨*simpletype*⟩ `'='` ⟨*constrs*⟩	(algebraic type)
	|	⟨*goaltype*⟩ ⟨*goalid*⟩ `'='` ⟨*exp*⟩	(logical goal)
	|	⟨*decl*⟩	
⟨*simpletype*⟩	::=	⟨*tycon*⟩ {⟨*tyvar*⟩}	
⟨*goaltype*⟩	::=	`'theorem'` | `'lemma'`	
⟨*decls*⟩	::=	`'{'` ⟨*decl*⟩$_1$ `';'` ... `';'` ⟨*decl*⟩$_n$ `'}'` $n \geq 0$	
⟨*decl*⟩	::=	[⟨*sigdecl*⟩ `';'`] ⟨*funlhs*⟩ ⟨*rhs*⟩	

$$
\begin{array}{lll}
 & | & \langle pat^0 \rangle) \ \langle rhs \rangle \\
\langle sigdecl \rangle & ::= & \langle var \rangle \ \texttt{`:'} \ \langle type \rangle \\
\langle funlhs \rangle & ::= & \langle var \rangle \ \{\langle apat \rangle\}^+ \\
\langle rhs \rangle & ::= & \texttt{`='} \ \langle exp \rangle \ [\texttt{`where'} \ \langle decls \rangle] \\
 & | & \langle gdrhss \rangle \ [\texttt{`where'} \ \langle decls \rangle] \\
\langle gdrhss \rangle & ::= & \langle gdrhs \rangle^+ \ [\langle egdrhs \rangle] \\
 & | & \{\langle gdrhs \rangle\} \ \langle egdrhs \rangle \\
\langle gdrhs \rangle & ::= & \langle gd \rangle \ \texttt{`='} \ \langle exp \rangle \\
\langle gd \rangle & ::= & \texttt{`|'} \ \langle exp^0 \rangle \\
\langle egdrhs \rangle & ::= & \texttt{`|'} \ \texttt{`else'} \ \texttt{`='} \ \langle exp \rangle \\
\langle constrs \rangle & ::= & \langle constr \rangle_1 \ \texttt{`|'} \ \ldots \ \texttt{`|'} \ \langle constr \rangle_n \quad n \geq 1 \\
\langle constr \rangle & ::= & \langle con \rangle \ \{\langle atype \rangle\}
\end{array}
$$

There is a syntactic separation between declarations that are only allowed at the top level of a $\mathcal{H}_{\text{SPEC}}$ module, *topdecls*; and those that may be used either at the top level or in nested scopes, within a *let* or *where* construct. Currently $\mathcal{H}_{\text{SPEC}}$ allows recursive declarations but not mutually recursive declarations.

4.7.1 Type synonyms

A type synonym declaration `type` $T \ a_1 \ \ldots \ a_k$ `=` t introduces a new type constructor T. The type $T \ u_1 \ \ldots \ u_k$ is equivalent to the type $t[\overline{a \mapsto u}]$. The type variables a_i must be distinct and are scoped over the righ-hand side of the definition, t. Type constructors introduced by type synonyms declarations, T, cannot be partially applied —applications must be saturated. Type synonyms declarations cannot be recursive.

Most languages use type synonyms as a convenient mechanism to introduce *unchecked* documentation and make type signatures more readable. For instance, in Haskell we may declare a new type `Nat`, defined as `type Nat = Int`, to document that some values are expected to be non-negative. In $\mathcal{H}_{\text{SPEC}}$ type synonyms are often used to name predicate subtypes, thus constituting *checked* documentation. Actually, the type `Nat` in $\mathcal{H}_{\text{SPEC}}$ is defined as `type Nat = {n:Int | n >= 0}`.

4.7.2 Algebraic datatypes

An algebraic datatype declaration has the form:

`data` T a_1 ... a_k `=` K_1 t_{11} ... t_{1k_1} `|` ... `|` K_n t_{n1} ... t_{nk_n}

and introduces a new type constructor T with one or more (data) constructors $K_1, \ldots, K_n$. Type variables a_i must be distinct and scope over all t_{ij}; no other type variable can appear on the right-hand side of the datatype declaration. Any use of the type constructor T must be saturated. The type of data constructor K_i is given by:

K_i `:` `forall` a_1 ... a_k`.` t_{i1} `->` ... `->` t_{ik_i} `->` T a_1 ... a_k

4.7.3 Function bindings

A function binding binds a variable to a function value, but it provides a more appealing syntax than ordinary lambda abstractions. The general form of a function binding (for a variable `f`) is:

`f` $p_{11} \ldots p_{1k}$ $match_1$

...

`f` $p_{n1} \ldots p_{nk}$ $match_n$

where $n \geq 1$, p_{ij} are patterns, and each $match_i$ is of the general form:

`|` g_{i1} `=` e_{i1}

...

`|` g_{im_i} `=` e_{im_i}

`where` $decls_i$

All the equations defining a function must be contiguous, and the number of patterns in each equation must be the same. As usual, the set of patterns corresponding to each match must be linear.

A type signature may be provided to specify the type of the variable being defined. This type signature is optional but, if given, it must be declared immediately before the set of defining function equations.

`f :` *type*

`f` $p_{11} \ldots p_{1k}$ $match_1$

```
...
f p_n1 ... p_nk match_n
```

If a type signature is given for a variable `f`, then that will be the visible type of `f` within the module. The declared type must be compatible with the inferred type for `f`.

The general form of a function binding can be translated into a binding of the variable `f` to a lambda abstraction whose arguments are scrutinized by a case expression:

```
f = \ x_1 ... x_k -> case (x_1, ... ,x_k) of
                        (p_11, ... ,p_1k) match_1
                        ...
                        (p_n1, ... ,p_nk) match_n
```

where $\mathtt{x}_1 \ldots \mathtt{x}_k$ are fresh identifiers.

4.7.4 Pattern bindings

Pattern bindings perform irrefutable pattern matching of an expression to some pattern p, bringing the variables of p into scope. A pattern binding (for a pattern p) has the general form:

```
p | g_1 = e_1
  ...
  | g_m = e_m
  where decls
```

Every e_i must be proven to match the pattern p, because of the completeness restriction of pattern matching. The pattern binding above can be translated into a simple pattern binding $p = e$ by following the translation rules given in § 4.6.

Pattern bindings, once transformed to the simple form $p = e$, can be translated into a set of core function bindings, by introducing a function binding for each variable in p that uses a case expression to match e against p and return the binding for that variable. For example, a pattern binding `(x,y) =` e can be translated to:

```
x = case e of (a,_) -> a
y = case e of (_,b) -> b
```

4.7.5 Logical goals

A logical goal is a high-order logic sentence that must be proved valid. A goal declaration has the form $G\ g$ `=` $Q\ x_1\ \ldots\ x_n$`.` exp where G is either `theorem` or `lemma`, Q is either `forall` or `exists`; and $x_1\ \ldots\ x_n$ are distinct variables scoping over exp, that must be a boolean expression. A logical goal may be polymorphic, but such polymorphism cannot be specified explicitly and has to be inferred by the typechecker.

4.8 Modules

A module defines a collection of types, values and logical goals. $\mathcal{H}_{\text{SPEC}}$ does not own a proper module system, it is very naive and not intended for practical use. The features introduced by $\mathcal{H}_{\text{SPEC}}$ open a new design space, allowing features such as parametrized modules in the spirit of PVS, and this task is out of the scope of this work. Hence, it was preferred to keep the module system pretty much undefined than to stick with plain Haskell modules, since the latter would complicate the implementation of a typechecker without constituting a valuable contribution.

$$\begin{aligned}
\langle module \rangle \ &::= \texttt{`module'}\ \langle modid \rangle\ \texttt{`where'}\ \langle body \rangle \\
&\ \ \mid\ \langle body \rangle
\end{aligned}$$

$$\langle body \rangle \ ::= \texttt{`\{'}\ \langle topdecls \rangle\ \texttt{`\}'}$$

$$\langle topdecls \rangle \ ::= \langle topdecl \rangle_1\ \texttt{`;'}\ \ldots\ \texttt{`;'}\ \langle topdecl \rangle_n \quad n \geq 1$$

4.9 Key design choices

$\mathcal{H}_{\text{SPEC}}$ was carefully designed to be a practical programming language suitable for developing reliable software. In contrast with most academic approaches, we want to avoid complicated type theories as well as the burden of proof usually associated with dependently-typed programming languages. This section discusses the key design choices that, in our view, make $\mathcal{H}_{\text{SPEC}}$ a language to effectively approach error-free programming.

4.9.1 Proving is optional

Strictly speaking $\mathcal{H}_{\text{SPEC}}$ is a language with a type system that was designed to guarantee error-free programs. This is not for free and as a consequence typechecking is undecidable. For good reasons, undecidability of typechecking is generally not considered acceptable for a programming language. What makes $\mathcal{H}_{\text{SPEC}}$ different is that, as PVS, there is a clear separation between the decidable and undecidable parts of typechecking.

The process of typechecking for $\mathcal{H}_{\text{SPEC}}$ logically consists of two phases. First, it is performed typechecking with respect to plain-old types only —which is decidable. Any type error that would be detected by a plain Damas-Milner type system is also going to be detected at this phase, causing the whole process to fail. Then, during the second phase, the typechecker deals with coercions between types and other side conditions. The typechecker will generate a proof obligation for any condition that cannot be trivially discarded. To be precise typechecking is not considered complete until all proof obligations are discarded, which could be view as a third phase of the typechecker that requires user intervention.

The ultimate goal of $\mathcal{H}_{\text{SPEC}}$ is to be a practical programming language, therefore a high degree of flexibility is allowed, specially in comparison with more formal approaches. First of all, any $\mathcal{H}_{\text{SPEC}}$ implementation must allow compilation once typechecking with respect to plain types was successfully completed. It is actually easier for a programmer to be willing to provide annotations using all the expressiveness of $\mathcal{H}_{\text{SPEC}}$ types, once he is released of the burden of proving verification conditions. Moreover, the $\mathcal{H}_{\text{SPEC}}$ toolset must provide multiple back-ends for verification, supporting at least automated proving and randomized testing based methods for discharging proof obligations. Hence, the use of rich type annotations is highly recommended, as they constitute valuable documentation that can be later checked using automated methods or just simply tested.

4.9.2 Propositions are boolean expressions

In $\mathcal{H}_{\text{SPEC}}$ logical propositions are plain boolean expressions augmented with universal and existential quantification. As a consequence a mechanism to avoid quantifiers from appearing on ordinary functions is required, since quantified formulas are not generally

executable expressions. The way this problem is addressed in $\mathcal{H}_{\text{SPEC}}$ is rather simple and can be formalized as an *effect system*: quantifiers are banned outside of purely logical contexts, that is, predicate types and logical goals. This approach does not further complicate the type system which, in our experience, is both welcomed by implementors and programmers. There is a deeper consequence of this choice, it means that $\mathcal{H}_{\text{SPEC}}$ propositions are sentences of *classical* high-order logic. In comparison with intuitionistic logic, classical logic is better suited for automated reasoning; in particular first-order logic is semi-decidable and there exist many interesting decidable fragments of it.

4.9.3 Pattern types

$\mathcal{H}_{\text{SPEC}}$ generalized the concept of predicate type by allowing to filter the set of elements of the base type by pattern matching. For instance, the type {`(x::xs):[`t`]`} stands for all non-empty lists of type t. Predicate types with patterns can be translated into standard predicate types, but such support for pattern matching turned out to be very convenient.

Both types {`(_::_):[`t`]`} `->` t and {`l:[`t`] | not (null l)`} `->` t are (isomorphic) valid types for the function `head`, which returns the head element of a non-empty list. However the former is more concise, and it can be easily inferred by the typechecker given `head (x::_) = x`. Moreover, many kinds of coercions involving pattern types can be easily discharged automatically, like {`(_::_::_):[Int]`} $\preceq$ {`(_::_):[Int]`} —lists of 2 or more elements are (obviously) non-empty lists. Finally, patterns are extremely useful for test-case generation: given a type {`(_::_):[Int]`} it is straightforward to generate values of such type (non-empty lists of integers) by taken the pattern into account.

Pattern types can also be view as convenient syntactic sugar that allow to write down succinct types that would look nasty otherwise. The Haskell's `divMod` function is a good example, since it can be described in $\mathcal{H}_{\text{SPEC}}$ by the type:

```
{x:Int} -> {y:Int | y /= 0} -> {(q,r):(Int,Int) | r <= y && x == q*y + r}
```

4.9.4 Control flow constraints

$\mathcal{H}_{\text{SPEC}}$ imposes *completeness* and *disjointness* constraints to every control flow construct. Completeness essentially ensures that every possible path is being considered, thus avoiding potential execution-time errors. The purpose of disjointness is threefold: *a*) to facilitate compilation and TCC generation, especially for case expressions; and *b*) to improve readability of code by forcing the programmer to identify non-overlapping execution paths, which usually leads to *c*) simpler and more concise TCCs .

Probably the most distinguishing feature of $\mathcal{H}_{\text{SPEC}}$ is the choice of *uniform patterns* to allow nested patterns while statically guaranteeing disjointness. Other languages like PVS that follow the same philosophy support simple patterns only, which can be considered too restrictive.

Chapter 5

The $\mathcal{H}_{\text{spec}}$ Type System

The type system of $\mathcal{H}_{\text{SPEC}}$ is a traditional Damas-Milner type system extended to support predicate subtypes and limited forms of dependent types, what we refer to as *contract types*. Damas-Milner is the basis of today's functional programming languages due to its expressiveness, and a fairly simple and efficient type inference algorithm: the so-called Algorithm $\mathcal{W}$ presented in § 2.3.3. The major contribution of this work is to provide a formal presentation of this extension to Damas-Milner, including a detailed type inference algorithm.

Describing a type system for $\mathcal{H}_{\text{SPEC}}$ would be cumbersome due to all its syntactic constructs. Instead, we will make use of the $\mathcal{H}_{\text{SPEC}}$ kernel language that was depicted in the previos chapter, and that is formally described in § 5.1. First, we introduce some preliminary definitions and notations in § 5.2. The declarative and syntax-directed formulations of this type system are presented in § 5.3. Next in § 5.4, we show how to extend the Algorithm $\mathcal{M}$ —a popular variation of Algorithm $\mathcal{W}$— to perform type inference for contract types. Finally, we discuss the introduction of pattern types into this system in § 5.5.

5.1 Language

The $\mathcal{H}_{\text{SPEC}}$ kernel was informally described previously in chapter 4 by the translation rules provided to transform $\mathcal{H}_{\text{SPEC}}$ constructs into their most fundamental form. Figure 5.1 describes the language of types. We use T to range over built-in type constructors as well as any user-defined datatype. Built-in type constructors include `()`,

`Bool` and `Int`. The type of lists, whose type constructor is written `[]`, has special syntactic support so the type `[]`τ can be written as `[`τ`]`. Moreover, any application of a type constructor T must be saturated. A function type $\{x{:}\tau_1\}$ `->` τ_2 where x does not appear in τ_2 may be written as τ_1 `->` τ_2. In the same way, we may omit any variable bound within a tuple type if the other fields of the tuple do not depend on it.

Type variables ($\mathcal{U}$)	a, b, c			
Type constructors	T	$\supseteq$	$\{$`()`, `Bool`, `Int`, `[]`$\}$	
Monotypes ($\mathcal{T}$)	τ, υ	::=	a	Type variable
		|	$T\ \tau_1\ \ldots\ \tau_n$	Type constructor
		|	$\{x{:}\tau \mid P\}$	Predicate subtype
		|	`(`$x_1{:}\tau_1$`,` $\ldots$ `,`$x_k{:}\tau_k$`)`	k-tuples
		|	$\{x{:}\tau_1\}$ `->` τ_2	Dependent function
Polytypes ($\mathcal{S}$)	σ	::=	`forall` $\overline{a}$`.` τ	Type scheme

Figure 5.1: Syntax of $\mathcal{H}_{\text{SPEC}}$-kernel types.

The language of $\mathcal{H}_{\text{SPEC}}$-kernel terms is shown in Figure 5.2. We use C to range over data constructors of built-in types as well as over those introduced by user-defined datatypes. The set of built-in constructors includes `()`, `False`, `True`, the empty-list constructor `[]`, and the *cons*-list constructor `::`. This cons-list constructor supports special syntax so that `::` e_1 e_2 can be written as e_1`::`e_2 using infix notation.

We also assume the existence of a number of special symbols `+`,`-`,`*`,... for denoting the built-in arithmetic, boolean and equality operators of $\mathcal{H}_{\text{SPEC}}$. All these symbols, analogously to `::`, may be written in prefix or infix notation for syntactic convenience. We may write `if` g `then` e_1 `else` e_2 as an alias for `if |` g `->` e_1 `|` $\sim g$ `->` e_2. Notice that bi-implication `<=>` is equivalent to `==` restricted to booleans; and conjunction `&&`, disjunction `||`, and implication `==>` can be interpreted as if-then-else expressions:

- e_1 `&&` $e_2 \equiv$ `if` e_1 `then` e_2 `else False`
- e_1 `||` $e_2 \equiv$ `if` e_1 `then True else` e_2
- e_1 `==>` $e_2 \equiv$ `if` e_1 `then` e_2 `else True`

This interpretation of the logical operators is inherited from PVS.

Term variables ($\mathcal{V}$)	x, y, z			
Integers ($\mathbb{Z}$)	i			
Data constructor	C	$\supseteq$	$\{\texttt{()}, \texttt{False}, \texttt{True}, \texttt{[]}\}$	
Terms (E)	e, g, t, u	::=	i	Literal
		|	x	Variable
		|	C	Data constructor
		|	$\backslash\ x$ `->` t	λ-abstraction
		|	$t\ u$	Application
		|	`let` $x[\texttt{:}\sigma]$ `=` u `in` t	Let binding
		|	`if |` g_1 `->` e_1 $\ldots$ `|` g_n `->` e_n	If expression
		|	`case` t `of` $\{p_1$ `->` e_1`;` $\ldots$`;` p_n `->` $e_n\}$	Case expression
		|	`forall` x`,` P	Quantified formula
Propositions ($\mathcal{P}$)	P, Q	::=	e	

Figure 5.2: Syntax of $\mathcal{H}_{\text{SPEC}}$-kernel terms.

Let-bindings may optionally have a type signature. Types signatures can be used to annotate terms with contracts that cannot be inferred otherwise. What is more important, in order to simplify the presentation of the typing rules, our let-bindings are not recursive. Instead, we assume the existence of a fix-point operator `fix`, of type `forall a. (a -> a) -> a`, such that a recursive binding `let` f `=` $e[f]$ `in` t can be converted into a non-recursive binding `let` f `= fix (\` f_1 `->` $e[f_1]$`) in` t, where f_1 is a fresh name.

Patterns	pat, p	::=	i	Literal pattern
		|	x	Variable pattern
		|	$C\ x_1\ \ldots\ x_n$	Constructor pattern
		|	$\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_k\texttt{)}$	Tuple pattern

Figure 5.3: Syntax of $\mathcal{H}_{\text{SPEC}}$-kernel patterns.

$\mathcal{H}_{\text{SPEC}}$-kernel patterns are simple (non-nested) and linear, as shown in Figure 5.3. Constructor patterns must be saturated and bound distinct variables, the same applies to k-tuple patterns. We write C to range over any data constructor, and analogously to expressions a `::`-pattern can be written as x_1`::`x_2 using infix notation.

5.2 Preliminaries

Definition 5.1 (Skolemization). The skolemization of a type σ, written $\hat{\sigma}$, consists of the substitution of every variable $a \in \mathsf{FTV}(\sigma)$ by a new skolem constant $\hat{a}$.

The coercion relation $\preceq$ determines whether an expression e can be coerced from the source type σ_s, to the target type σ_t. We write this as $\sigma_s \overset{e}{\preceq} \sigma_t$, and the logical formula that entails that such a coercion is possible is called a *type correctness condition* (TCC). The subtype relation between types is written as $\sigma_s \preceq \sigma_t$ and defined as $\forall x.\ \sigma_s \overset{x}{\preceq} \sigma_t$. Note that the latter is a more strong relation, for instance $\texttt{Int} \overset{1}{\preceq} \texttt{Nat}$ but $\texttt{Int} \not\preceq \texttt{Nat}$.

Definition 5.2 (Type coercion).

$$
\begin{array}{rcl}
a \overset{e}{\preceq} a & = & \top \\
T\ \tau_1 \ldots \tau_n \overset{e}{\preceq} T\ \upsilon_1 \ldots \upsilon_n & = & \texttt{all}_T\ \pi_{\tau_1 \preceq \upsilon_1} \ldots \pi_{\tau_1 \preceq \upsilon_1}\ e \\
\{x{:}\tau \mid P\} \overset{e}{\preceq} \upsilon & = & P[e] \Rightarrow \tau \overset{e}{\preceq} \upsilon \\
\tau \overset{e}{\preceq} \{y{:}\upsilon \mid Q\} & = & \psi \wedge (\psi \Rightarrow Q[e]) \\
& \& & \psi := \tau \overset{e}{\preceq} \upsilon \\
(x_1{:}\tau_1, \ldots, x_k{:}\tau_k) \overset{e}{\preceq} (y_1{:}\upsilon_1, \ldots, y_k{:}\upsilon_k) & = & \texttt{case}\ e\ \texttt{of}\ (x_1, \ldots, x_k)\ \texttt{->}\ \bigwedge_i \tau_i \overset{x_i}{\preceq} \upsilon_i[\overline{y \mapsto x}] \\
\{x{:}\tau_1\}\ \texttt{->}\ \tau_2 \overset{f}{\preceq} \{y{:}\upsilon_1\}\ \texttt{->}\ \upsilon_2 & = & \forall x.\ \psi \wedge (\psi \Rightarrow \tau_2 \overset{f\ x}{\preceq} \upsilon_2[x]) \\
& \& & \psi := \forall z.\ \upsilon_1 \overset{z}{\preceq} \tau_1 \quad z\ \textit{fresh} \\
\texttt{forall}\ \overline{a}.\ \tau \overset{e}{\preceq} \texttt{forall}\ \overline{b}.\ \upsilon & = & \hat{\tau} \overset{e}{\preceq} \upsilon[\overline{b \mapsto \hat{a}}]
\end{array}
$$

Definition 5.3 (Coercion predicate). The coercion predicate $\pi_{\sigma_s \preceq \sigma_t} \triangleq \lambda x.\ \sigma_s \overset{x}{\preceq} \sigma_t$, is the logical predicate that classifies the terms of type σ_s that are also of type σ_t.

A *plain type*, or μ-type, is a plain-old type without any kind of contract within it. That is, a plain Damas-Milner type with neither predicate subtypes nor dependent types. A precise definition of μ-types is given by a function μ mapping a type to its associated plain type.

Definition 5.4 (μ-type).

$$
\begin{array}{rcl}
\mu(a) & = & a \\
\mu(T\ \tau_1 \ldots \tau_n) & = & T\ \mu(\tau_1) \ldots \mu(\tau_n) \\
\mu(\{x{:}\tau \mid P\}) & = & \mu(\tau) \\
\mu((x_1{:}\tau_1, \ldots, x_k{:}\tau_k)) & = & (\mu(\tau_1), \ldots, \mu(\tau_k)) \\
\mu(\{x{:}\tau_1\}\ \texttt{->}\ \tau_2) & = & \mu(\tau_1)\ \texttt{->}\ \mu(\tau_2) \\
\mu(\texttt{forall}\ \overline{a}.\ \tau) & = & \texttt{forall}\ \overline{a}.\ \mu(\tau)
\end{array}
$$

It is worth noting that μ does not define a *supertype* function, so in general we have that $\sigma \not\preceq \mu(\sigma)$.

Definition 5.5 (μ-equivalence). Two types σ_1 and σ_2 are equivalent modulo μ, written as $\sigma_1 \equiv_\mu \sigma_2$, if and only if $\mu(\sigma_1) \equiv \mu(\sigma_2)$.

We also introduce a weaker version of μ, called μ_0, defining a notion of *direct supertype* such that $\sigma \preceq \mu_0(\sigma)$.

Definition 5.6 (μ_0-type).

$$\begin{aligned} \mu_0(\{x : A \mid P\}) &= \mu_0(A) \\ \mu_0(A) &= A \qquad \text{otherwise} \end{aligned}$$

Definition 5.7 (Type contract). We abbreviate $\pi_{\mu(\sigma) \preceq \sigma}$ as π_σ, which is the predicate that characterizes the contract specified by the type σ on top of $\mu(\sigma)$.

5.3 Type System

What makes this type system different from a basic Damas-Milner system is that the process of typing involves the generation of a set of *type coercions*. Each coercion $\sigma_s \overset{e}{\preceq} \sigma_t$ denotes a *proof obligation*, that is a logical formula to be proven to ensure that the coercion holds. This is a consequence of the undecidability of type-checking in the presence of contract types. More precisely, proof obligations are pairs (Γ, ϱ), where Γ is the typing context in which the coercion ϱ was generated. We will write P_ϱ to mean the logical formula denoted by the type coercion ϱ.

Typing context	Γ	$::=$	ϵ	Empty context
		$\mid$	$\Gamma, \hat{a}$	Skolem type
		$\mid$	Γ, T	Type constructor
		$\mid$	$\Gamma, C : \sigma$	Data constructor
		$\mid$	$\Gamma, x : \sigma$	Variable
		$\mid$	$\Gamma, x : \sigma := e$	Definition
		$\mid$	$\Gamma, e \overset{\text{case}}{=} pat$	Pattern match
		$\mid$	Γ, P	Hypothesis

Figure 5.4: $\mathcal{H}_{\text{SPEC}}$-kernel typing contexts.

The process of collecting proof obligations requires to keep track of extra information beyond the standard list of in-scope variables and their types. Figure 5.4 shows the kind of information that is stored in a typing context. An in-scope variable $x : \sigma$ can be interpreted, in the context of a proof obligation, as an universally quantified variable. Polymorphic contract types lead to type coercions involving skolem type constants, so that no logical construct to quantify over type variables is required. Local definitions introduced by let-bindings are stored in the typing context as $x : \sigma := e$. It is also necessary to keep track of the control-flow path in which a type coercion is generated. A case-alternative is identified by a fact $e \overset{\text{case}}{=} pat$, whereas an if-alternative is kept track by adding the guard to the typing context as a logical hypothesis.

A judgment $\Gamma \vdash \sigma$ means that the type σ is well-formed in the context Γ, the rules for this judgment are shown in Figure 5.5. Checking a predicate subtype $\{x{:}\tau \mid P\}$ requires checking both the validity of τ and of the $P(x)$ predicate. Since P is an arbitrary logical term, the typability of P may depend on proof obligations. Checking dependent types is mostly straightforward, the main peculiarity of these rules is that they introduce variables ranging over arbitrary terms into scope. Universally quantified types are checked in skolemized form, thus the only type variables occurring free in Γ are those introduced by Damas-Milner's type generalization.

$$\boxed{\Gamma \vdash \sigma}$$

$$\textsc{VarTy}\ \frac{}{\Gamma, \hat{a} \vdash \hat{a}} \qquad \textsc{ConstrTy}\ \frac{\Gamma \vdash \tau_i}{\Gamma \vdash T\ \tau_1\ \ldots\ \tau_n} i \in [1, n] \qquad \textsc{PredicateTy}\ \frac{\Gamma \vdash \tau \qquad \Gamma, x : \tau \vdash^{\text{prop}} P : \texttt{Bool}}{\Gamma \vdash \{x{:}\tau \mid P\}}$$

$$\textsc{TupleTy}\ \frac{\Gamma, \bigcup_{j \in [1,i)} (x_j : \tau_j) \vdash \tau_i}{\Gamma \vdash (x_1{:}\tau_1, \ldots, x_k{:}\tau_k)} i \in [1, k] \qquad \textsc{FunTy}\ \frac{\Gamma \vdash \tau_1 \qquad \Gamma, x : \tau_1 \vdash \tau_2}{\Gamma \vdash \{x{:}\tau_1\}\ \texttt{->}\ \tau_2} \qquad \textsc{ForallTy}\ \frac{\Gamma, \overline{\hat{a}} \vdash \hat{\tau}}{\Gamma \vdash \texttt{forall}\ \overline{a}.\ \tau}$$

Figure 5.5: Declarative typing rules for $\mathcal{H}_{\text{SPEC}}$-kernel types.

A typing judgment is of the form $\Gamma \vdash t : \sigma, \mathbb{P}$, meaning that a term t has type σ in the context Γ given that all the proof obligations in $\mathbb{P}$ hold. However, for the sake

of simplicity, we present the type system using an alternative (but equivalent) typing judgment $\Gamma \vdash t : \sigma$ in which the set of proof obligations $\mathbb{P}$ is implicitly constructed by *coercion judgments*. A coercion judgment $\Gamma \vdash^{\text{co}} \sigma_s \overset{e}{\preceq} \sigma_t$ means that the coercion $\sigma_s \overset{e}{\preceq} \sigma_t$ must hold in the context Γ, and every use of this judgment to build a typing derivation records a proof obligation $(\Gamma, \sigma_s \overset{e}{\preceq} \sigma_t)$. In the same way, we may write $\Gamma \vdash \sigma, \mathbb{P}$ to explicit the set of proof obligations generated while checking the validity of a type σ. The set of declarative typing rules for our system is given in Figure 5.6.

$$\boxed{\Gamma \vdash t : \sigma}$$

$$\textsc{Var}\ \frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \textsc{Constr}\ \frac{}{\Gamma, C : \sigma \vdash C : \sigma} \qquad \textsc{Int}\ \frac{}{\Gamma \vdash i : \texttt{Int}}\, i \in \mathbb{Z}$$

$$\textsc{Tuple}\ \frac{\Gamma \vdash e_i : \tau_i[x_1 \mapsto e_1][\ldots][x_{i-1} \mapsto e_{i-1}]}{\Gamma \vdash (e_1, \ldots, e_k) : (x_1{:}\tau_1, \ldots, x_k{:}\tau_k)}\, i \in [1,k] \qquad \textsc{ForallP}\ \frac{\Gamma, x : \tau \vdash^{\text{prop}} P : \texttt{Bool}}{\Gamma \vdash^{\text{prop}} \texttt{forall}\ x,\ P : \texttt{Bool}}$$

$$\textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \backslash\ x\ \texttt{->}\ t : \{x{:}\tau_1\}\ \texttt{->}\ \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash t : \{x{:}\tau_1\}\ \texttt{->}\ \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2[x \mapsto u]}$$

$$\textsc{Let}\ \frac{\Gamma \vdash u : \sigma \qquad \Gamma, x : \sigma := u \vdash t : \tau}{\Gamma \vdash \texttt{let}\ x[{:}\sigma]\ \texttt{=}\ u\ \texttt{in}\ t : \tau[x \mapsto u]} \qquad \textsc{If}\ \frac{\Gamma \vdash g_i : \texttt{Bool} \qquad \Gamma, g_i \vdash e_i : \tau}{\Gamma \vdash \texttt{if}\ \texttt{|}\ g_1\ \texttt{->}\ e_1\ \ldots\ \texttt{|}\ g_n\ \texttt{->}\ e_n : \tau}\, i \in [1,n]$$

$$\textsc{Case}\ \frac{\Gamma \vdash t : \tau_s \qquad \Gamma \vdash^{\text{pat}} pat_i : \tau_s, \Gamma_{pat_i} \qquad \Gamma, \Gamma_{pat_i}, t \overset{\text{case}}{=} pat_i \vdash e_i : \tau}{\Gamma \vdash \texttt{case}\ t\ \texttt{of}\ \{pat_1\ \texttt{->}\ e_1\texttt{;}\ \ldots\ \texttt{;}pat_n\ \texttt{->}\ e_n\} : \tau}\, i \in [1,n]$$

$$\textsc{Coerce}\ \frac{\Gamma \vdash e : \sigma_s \qquad \Gamma \vdash^{\text{co}} \sigma_s \overset{e}{\preceq} \sigma_t}{\Gamma \vdash e : \sigma_t} \qquad \textsc{Gen}\ \frac{\overline{a} \notin \mathsf{FTV}(\Gamma) \qquad \Gamma \vdash t : \tau}{\Gamma \vdash t : \forall \overline{a}.\ \tau} \qquad \textsc{Inst}\ \frac{\Gamma \vdash t : \forall \overline{a}.\ \tau}{\Gamma \vdash t : \tau[\overline{a \mapsto v}]}$$

Figure 5.6: Declarative typing rules for $\mathcal{H}_{\text{SPEC}}$-kernel terms.

Rule ForallP shows another relevant peculiarity of this type system. The typing relation $\vdash$ has a flag called *prop* that indicates whether the term being typed is part of a logical proposition. An universally quantified formula, which is not an executable term, is only valid in such situation. Any rule not mentioning it in its conclusion is valid whether the flag is set or not, and the value of the flag is implicitly propagated to the premises.

$$\boxed{\Gamma \vdash^{\text{pat}} pat : \sigma, \Gamma_{pat}}$$

$$\textsc{VarPat}\ \frac{}{\vdash^{\text{pat}} x : \tau, \{x : \tau\}} \qquad \textsc{IntPat}\ \frac{}{\vdash^{\text{pat}} i : \texttt{Int}, \emptyset}$$

$$\textsc{ConstrPat}\ \frac{\Gamma \vdash C : \{y_1 : v_1\}\ \texttt{->}\ \ldots\ \texttt{->}\ \{y_n : v_n\}\ \texttt{->}\ T\ \tau_1\ \ldots\ \tau_k}{\Gamma \vdash^{\text{pat}} C\ x_1\ \ldots\ x_n : T\ \tau_1\ \ldots\ \tau_k,\ \bigcup_{i \in [1,n]} (x_i : v_i[y_1 \mapsto x_1][\ldots][y_{i-1} \mapsto x_{i-1}])}$$

$$\textsc{TuplePat}\ \frac{}{\vdash^{\text{pat}} (x_1, \ldots, x_k) : (y_1 : \tau_1, \ldots, y_k : \tau_k),\ \bigcup_{i \in [1,k]} (x_i : \tau_i[y_1 \mapsto x_1][\ldots][y_{i-1} \mapsto x_{i-1}])}$$

Figure 5.7: Declarative typing rules for $\mathcal{H}_{\text{SPEC}}$-kernel patterns.

Typing case expressions require a new judgment $\Gamma \vdash^{\text{pat}} pat : \sigma, \Gamma_{pat}$ meaning that a pattern *pat* has type σ and binds the variables in Γ_{pat}. The rules for this judgment are given in Figure 5.7. Notice that configuring the Γ_{pat} environment in ConstrPat and TuplePat rules is considerably more complicated due to dependent typing.

5.3.1 Bidirectional type system

In order to move towards a type inference algorithm we provide a syntax-directed presentation of the $\mathcal{H}_{\text{SPEC}}$-kernel type system. Nonetheless, deriving an algorithmic version of the above type system is not as natural as for Damas-Milner, in which one simply has to identify a couple of typing rules in where type generalization/instantiation

take place. In this case, the syntax-directed conversion is more complicated because, for some kinds of terms, type-checking is considerably different from type-inference. The problem appears when we try to provide a single syntax-directed rule capturing both type-checking and type-inference. For this reason, the syntax-directed version of our type system is presented as a bidirectional type system.

A bidirectional type system is described by providing separate sets of typing rules for checking and inferring types. A typing judgments is thus presented in two different flavors: a judgment $\vdash_{\Downarrow}$ for type-checking, and a judgment $\vdash_{\Uparrow}$ for type-inference. The idea comes from a technique called *local type inference* that was originally introduced by Pierce and Turner in [53], and that was also demonstrated useful to extend a Damas-Milner system to support arbitrary-rank polymorphism [23]. Figure 5.8 shows the syntax-directed rules for types, which are the same given in Figure 5.5, except for the PredicateTy rule where a type-predicate P is now *checked* to be of type `Bool`.

$$\boxed{\Gamma \vdash \sigma}$$

$$\textsc{VarTy}\ \frac{}{\Gamma, \hat{a} \vdash \hat{a}} \qquad \textsc{ConstrTy}\ \frac{\Gamma \vdash \tau_i}{\Gamma \vdash T\ \tau_1\ \ldots\ \tau_n}\, i \in [1,n] \qquad \textsc{PredicateTy}\ \frac{\Gamma \vdash \tau \qquad \Gamma, x : \tau \vdash^{\text{prop}}_{\Downarrow} P : \texttt{Bool}}{\Gamma \vdash \{x{:}\tau \mid P\}}$$

$$\textsc{TupleTy}\ \frac{\Gamma,\ \bigcup_{j \in [1,i)} (x_j : \tau_j) \vdash \tau_i}{\Gamma \vdash (x_1{:}\tau_1, \ldots, x_k{:}\tau_k)}\, i \in [1,k] \qquad \textsc{FunTy}\ \frac{\Gamma \vdash \tau_1 \qquad \Gamma, x : \tau_1 \vdash \tau_2}{\Gamma \vdash \{x{:}\tau_1\}\ \texttt{->}\ \tau_2} \qquad \textsc{ForallTy}\ \frac{\Gamma, \overline{\hat{a}} \vdash \hat{\tau}}{\Gamma \vdash \texttt{forall}\ \overline{a}.\ \tau}$$

Figure 5.8: Syntax-directed typing rules for $\mathcal{H}_{\text{SPEC}}$-kernel types.

Figure 5.9 gives the syntax-directed checking rules for $\mathcal{H}_{\text{SPEC}}$-kernel; the main typing judgment is written $\Gamma \vdash_{\Downarrow} t : \tau$, read as "in context Γ the term t can be *checked* to have type τ". The down-arrow $\Downarrow$ suggests the idea of pushing a type down into a term. The syntax-directed inference rules are shown in Figure 5.10, in this case the typing judgment takes the form $\Gamma \vdash_{\Uparrow} t : \tau$, which is read as "in context Γ the term t can be *inferred* to have type τ". The up-arrow $\Uparrow$ suggests the idea of pulling a type out of a term. The judgment $\vdash^{\text{poly}}$ is generalized in the same way, while the judgment $\vdash^{\text{inst}}$ remains the same as for a Damas-Milner system.

$$\boxed{\Gamma \vdash_{\Downarrow} t : \tau}$$

$$\textsc{Var}\ \frac{\vdash^{\text{inst}} \sigma_x \leq \tau_x \qquad \Gamma \vdash^{\text{co}} \tau_x \overset{x}{\preceq} \tau}{\Gamma, x : \sigma_x \vdash_{\Downarrow} x : \tau} \qquad \textsc{Constr}\ \frac{\vdash^{\text{inst}} \sigma_c \leq \tau_c \qquad \Gamma \vdash^{\text{co}} \tau_c \overset{C}{\preceq} \tau}{\Gamma, C : \sigma_c \vdash_{\Downarrow} C : \tau} \qquad \textsc{Int}\ \frac{\Gamma \vdash^{\text{co}} \texttt{Int} \overset{i}{\preceq} \tau}{\Gamma \vdash_{\Downarrow} i : \tau}\ i \in \mathbb{Z}$$

$$\textsc{Tuple}\ \frac{\mu_0(\tau) = (x_1 : \tau_1, \ldots, x_k : \tau_k) \qquad \Gamma \vdash_{\Downarrow} e_i : \tau_i[x_1 \mapsto e_1][\ldots][x_{i-1} \mapsto e_{i-1}] \qquad \Gamma \vdash^{\text{co}} (x_1 : \tau_1, \ldots, x_k : \tau_k) \overset{(e_1,\ldots,e_k)}{\preceq} \tau}{\Gamma \vdash_{\Downarrow} (e_1, \ldots, e_k) : \tau}\ i \in [1, k]$$

$$\textsc{ForallP}\ \frac{\Gamma, x : \tau \vdash^{\text{prop}}_{\Downarrow} P : \texttt{Bool} \qquad \Gamma \vdash^{\text{co}} \texttt{Bool} \overset{\texttt{forall}\ x,\ P}{\preceq} \tau}{\Gamma \vdash^{\text{prop}}_{\Downarrow} \texttt{forall}\ x, P : \tau}$$

$$\textsc{Abs}\ \frac{\mu_0(\tau) = \{y : \tau_1\}\ \texttt{->}\ \tau_2 \qquad \Gamma, x : \tau_1 \vdash_{\Downarrow} t : \tau_2[y \mapsto x] \qquad \Gamma \vdash^{\text{co}} \{y : \tau_1\}\ \texttt{->}\ \tau_2 \overset{\backslash\ x\ \texttt{->}\ t}{\preceq} \tau}{\Gamma \vdash_{\Downarrow} \backslash\ x\ \texttt{->}\ t : \tau}$$

$$\textsc{App}\ \frac{\Gamma \vdash_{\Uparrow} t : \tau_t \qquad \mu_0(\tau_t) = \{x : \tau_1\}\ \texttt{->}\ \tau_2 \qquad \Gamma \vdash_{\Downarrow} u : \tau_1 \qquad \Gamma \vdash^{\text{co}} \tau_2[x \mapsto u] \overset{t\ u}{\preceq} \tau}{\Gamma \vdash_{\Downarrow} t\ u : \tau}$$

$$\textsc{Let}\ \frac{\Gamma \vdash^{\text{poly}}_{\Uparrow} u : \sigma \qquad \Gamma, x : \sigma := u \vdash_{\Downarrow} t : \tau}{\Gamma \vdash_{\Downarrow} \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \tau} \qquad \textsc{ALet}\ \frac{\Gamma \vdash^{\text{poly}}_{\Downarrow} u : \sigma \qquad \Gamma, x : \sigma := u \vdash_{\Downarrow} t : \tau}{\Gamma \vdash_{\Downarrow} \texttt{let}\ x : \sigma\ \texttt{=}\ u\ \texttt{in}\ t : \tau}$$

$$\textsc{If}\ \frac{\Gamma \vdash_{\Downarrow} g_i : \texttt{Bool} \qquad \Gamma, g_i \vdash_{\Downarrow} e_i : \tau}{\Gamma \vdash_{\Downarrow} \texttt{if}\ \texttt{|}\ g_1\ \texttt{->}\ e_1\ \ldots\ \texttt{|}\ g_n\ \texttt{->}\ e_n : \tau}\ i \in [1, n]$$

$$\textsc{Case}\ \frac{\Gamma \vdash_{\Uparrow} t : \tau_s \qquad \Gamma \vdash^{\text{pat}}_{\Downarrow} pat_i : \tau_s, \Gamma_{pat_i} \qquad \Gamma, \Gamma_{pat_i}, pat_i \overset{\text{case}}{=} t \vdash_{\Downarrow} e_i : \tau}{\Gamma \vdash_{\Downarrow} \texttt{case}\ t\ \texttt{of}\ \{pat_1\ \texttt{->}\ e_1\texttt{;}\ \ldots\texttt{;} pat_n\ \texttt{->}\ e_n\} : \tau}\ i \in [1, n]$$

$$\boxed{\Gamma \vdash^{\text{poly}}_{\Downarrow} t : \sigma}$$

$$\textsc{Gen}\ \frac{\overline{a} \notin \mathsf{FTV}(\Gamma) \qquad \Gamma \vdash_{\Downarrow} t : \tau}{\Gamma \vdash^{\text{poly}}_{\Downarrow} t : \forall \overline{a}.\ \tau}$$

$$\boxed{\vdash^{\text{inst}} \sigma \leq \tau}$$

$$\textsc{Inst}\ \frac{}{\vdash^{\text{inst}} \forall \overline{a}.\ \tau \leq \tau[\overline{a \mapsto v}]}$$

Figure 5.9: Syntax-directed checking rules for $\mathcal{H}_{\text{SPEC}}$-kernel terms.

$$\boxed{\Gamma \vdash_{\Uparrow} t : \tau}$$

$$\textsc{Var}\ \frac{\vdash^{\text{inst}} \sigma \leq \tau}{\Gamma, x : \sigma \vdash_{\Uparrow} x : \tau} \qquad \textsc{Constr}\ \frac{\vdash^{\text{inst}} \sigma \leq \tau}{\Gamma, C : \sigma \vdash_{\Uparrow} C : \tau} \qquad \textsc{Int}\ \frac{}{\Gamma \vdash_{\Uparrow} i : \texttt{Int}}\, i \in \mathbb{Z}$$

$$\textsc{Tuple}\ \frac{\Gamma \vdash_{\Uparrow} e_i : \tau_i}{\Gamma \vdash_{\Uparrow} (e_1, \ldots, e_k) : (\tau_1, \ldots, \tau_k)}\, i \in [1,k] \qquad \textsc{ForallP}\ \frac{\Gamma, x : \tau \vdash^{\text{prop}}_{\Downarrow} P : \texttt{Bool}}{\Gamma \vdash^{\text{prop}}_{\Uparrow} \texttt{forall}\ x,\ P : \texttt{Bool}}$$

$$\textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash_{\Uparrow} t : \tau_2}{\Gamma \vdash_{\Uparrow} \backslash\, x \texttt{ -> } t : \{x : \tau_1\} \texttt{ -> } \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash_{\Uparrow} t : \tau_t \qquad \mu_0(\tau_t) = \{x : \tau_1\} \texttt{ -> } \tau_2 \qquad \Gamma \vdash_{\Downarrow} u : \tau_1}{\Gamma \vdash_{\Uparrow} t\ u : \tau_2[x \mapsto u]}$$

$$\textsc{Let}\ \frac{\Gamma \vdash^{\text{poly}}_{\Uparrow} u : \sigma \qquad \Gamma, x : \sigma := u \vdash_{\Uparrow} t : \tau}{\Gamma \vdash_{\Uparrow} \texttt{let}\ x = u\ \texttt{in}\ t : \tau[x \mapsto u]} \qquad \textsc{ALet}\ \frac{\Gamma \vdash^{\text{poly}}_{\Downarrow} u : \sigma \qquad \Gamma, x : \sigma := u \vdash_{\Uparrow} t : \tau}{\Gamma \vdash_{\Uparrow} \texttt{let}\ x : \sigma = u\ \texttt{in}\ t : \tau[x \mapsto u]}$$

$$\textsc{If}\ \frac{\Gamma \vdash_{\Downarrow} g_i : \texttt{Bool} \qquad \Gamma, g_i \vdash_{\Uparrow} e_i : \tau}{\Gamma \vdash_{\Uparrow} \texttt{if | } g_1 \texttt{ -> } e_1 \ \ldots\ \texttt{ | } g_n \texttt{ -> } e_n : \tau}\, i \in [1,n]$$

$$\textsc{Case}\ \frac{\Gamma \vdash_{\Uparrow} t : \tau_s \qquad \Gamma \vdash^{\text{pat}}_{\Downarrow} pat_i : \tau_s, \Gamma_{pat_i} \qquad \Gamma, \Gamma_{pat_i}, pat_i \overset{\text{case}}{=} t \vdash_{\Uparrow} e_i : \tau}{\Gamma \vdash_{\Uparrow} \texttt{case}\ t\ \texttt{of}\ \{pat_1 \texttt{ -> } e_1 \texttt{;} \ldots \texttt{;} pat_n \texttt{ -> } e_n\} : \tau}\, i \in [1,n]$$

$$\boxed{\Gamma \vdash^{\text{poly}}_{\Uparrow} t : \sigma} \qquad \boxed{\vdash^{\text{inst}} \sigma \leq \tau}$$

$$\textsc{Gen}\ \frac{\overline{a} = \mathsf{FTV}(\tau) - \mathsf{FTV}(\Gamma) \qquad \Gamma \vdash_{\Uparrow} t : \tau}{\Gamma \vdash^{\text{poly}}_{\Uparrow} t : \forall \overline{a}.\ \tau} \qquad \textsc{Inst}\ \frac{}{\vdash^{\text{inst}} \forall \overline{a}.\ \tau \leq \tau[\overline{a \mapsto v}]}$$

Figure 5.10: Syntax-directed inference rules for $\mathcal{H}_{\text{SPEC}}$-kernel terms.

When typing case-expressions we have to infer the type of the scrutinee and use that type to *check* the type of the patterns in the case alternatives, so any type invariant will be propagated to the variables being bound. The only judgment for patterns is $\Gamma \vdash^{\text{pat}}_{\Downarrow} pat : \tau, \Gamma_{pat}$, and it means that a pattern pat matches a scrutinee of type τ

and bounds the variables in Γ_{pat}. The set of checking rules for patterns are given in Figure 5.11

$$\boxed{\Gamma \vdash^{\text{pat}}_{\Downarrow} pat : \tau, \Gamma_{pat}}$$

$$\textsc{VarPat}\ \frac{}{\vdash^{\text{pat}}_{\Downarrow} x : \tau, \{x : \tau\}} \qquad \textsc{IntPat}\ \frac{\mu_0(\tau) = \texttt{Int}}{\Gamma \vdash^{\text{pat}}_{\Downarrow} i : \tau, \emptyset}$$

$$\textsc{TuplePat}\ \frac{\mu_0(\tau) = (y_1 : \tau_1, \ldots, y_k : \tau_k)}{\vdash^{\text{pat}}_{\Downarrow} (x_1, \ldots, x_k) : \tau, \bigcup_{i \in [1,k]} (x_i : \tau_i[y_1 \mapsto x_1][\ldots][y_{i-1} \mapsto x_{i-1}])}$$

$$\textsc{ConstrPat}\ \frac{\mu_0(\tau) = T\ \tau_1\ \ldots\ \tau_k \qquad \vdash^{\text{inst}} \sigma_c \leq \{y_1 : \upsilon_1\} \texttt{->} \ \ldots\ \texttt{->} \{y_n : \upsilon_n\} \texttt{->} T\ \tau_1\ \ldots\ \tau_k}{\Gamma, C : \sigma_c \vdash^{\text{pat}}_{\Downarrow} C\ x_1\ \ldots\ x_n : \tau, \bigcup_{i \in [1,n]} (x_i : \upsilon_i[y_1 \mapsto x_1][\ldots][y_{i-1} \mapsto x_{i-1}])}$$

Figure 5.11: Syntax-directed checking rules for $\mathcal{H}_{\text{SPEC}}$-kernel patterns.

5.4 Inference algorithm

The inference algorithm is derived applying mostly the same principles that were applied for a Damas-Milner system. Whenever a type has to be guessed we introduce a meta type variable $\mathring{\tau}$, and the constraints collected during the typing process through type unification will later determine the concrete value of that placeholder.

The judgment $\Gamma \vdash \sigma$, which states the validity of a type, leads to the $\mathsf{check}^{\text{type}}$ algorithm shown in Figure 5.12.

The definitions check_σ and check_τ given in Figure 5.13 are derived from the defining rules of the judgments $\Gamma \vdash^{\text{poly}}_{\Downarrow} t : \sigma$ and $\Gamma \vdash_{\Downarrow} t : \tau$, which describe how type-checking is performed. During type-checking we continuously face the problem of matching the actual type τ_a of a term t against the expected type τ_e. This matching is performed in two steps, providing the separation between the decidable and the undecidable parts of type-checking: *a*) the unification $\tau_a \sim \tau_e$ decides whether the types $\mu(\tau_a)$ and $\mu(\tau_e)$

$$
\begin{array}{rcl}
\mathsf{check}^{\mathrm{type}}(\Gamma \cup \{\hat{a}\}, \hat{a}) & \mapsto & \mathsf{ok} \\
\mathsf{check}^{\mathrm{type}}(\Gamma, \hat{a}) & \mapsto & \mathsf{fail} \\
 & \mathsf{if} & \hat{a} \notin \Gamma \\
\mathsf{check}^{\mathrm{type}}(\Gamma, T\ \tau_1\ \ldots\ \tau_n) & \mapsto & \mathsf{check}^{\mathrm{type}}(\Gamma, \tau_i) \\
\mathsf{check}^{\mathrm{type}}(\Gamma, \{x{:}\tau \mid P\}) & \mapsto & \mathsf{check}^{\mathrm{type}}(\Gamma, \tau), \\
 & & \mathsf{check}^{\mathrm{prop}}_{\tau}(\Gamma \cup \{x : \tau\}, P, \mathtt{Bool}) \\
\mathsf{check}^{\mathrm{type}}(\Gamma, (x_1{:}\tau_1, \ldots, x_n{:}\tau_n)) & \mapsto & \mathsf{check}^{\mathrm{type}}(\Gamma \cup \bigcup^{j\in[1,i)}(x_j : \tau_j), \tau_i) \\
\mathsf{check}^{\mathrm{type}}(\Gamma, \{x{:}\tau_1\}\ \texttt{->}\ \tau_2) & \mapsto & \mathsf{check}^{\mathrm{type}}(\Gamma, \tau_1), \\
 & & \mathsf{check}^{\mathrm{type}}(\Gamma \cup \{x : \tau_1\}, \tau_2) \\
\mathsf{check}^{\mathrm{type}}(\Gamma, \mathtt{forall}\ \overline{a}.\ \tau) & \mapsto & \mathsf{check}^{\mathrm{type}}(\Gamma, \overline{\hat{a}}, \tau[\overline{a \mapsto \hat{a}}]) \\
 & \mathsf{if} & \overline{a} \notin \Gamma,\ \overline{\hat{a}}\ \textit{fresh}
\end{array}
$$

Figure 5.12: $\mathcal{H}_{\mathrm{SPEC}}$-kernel algorithm to check validity of types.

are equivalent; and *b*) $\vdash_\Gamma \tau_a \overset{t}{\preceq} \tau_e$ records a proof obligation that is valid if and only if t has type τ_e.

The functions infer_σ and infer_τ given in Figure 5.14 are derived from the defining rules of the judgments $\Gamma \vdash^{\mathrm{poly}}_{\Uparrow} t : \sigma$ and $\Gamma \vdash_{\Uparrow} t : \tau$, which describe the type-inference algorithm. The definition of functions *gen* and *inst* was omitted because it is the same given for a Damas-Milner system.

Finally, Figure 5.15 shows the $\mathsf{check}_{\mathrm{pat}}$ algorithm that implements the typing judgment $\Gamma \vdash^{\mathrm{pat}}_{\Downarrow} pat : \tau, \Gamma_{pat}$.

It is worth noting that all the algorithms given above must keep track of a set of equalities between types that are later solved by unification. These algorithms, excluding $\mathsf{check}_{\mathrm{pat}}$, also carry the set of proof obligations generated during type-checking. We preferred to provide a (*monadic*) presentation in which such details are hidden. In our view this makes the algorithms more readable.

5.4.1 Unification

We present a modified version of the unification algorithm described in § 2.3.3 that handles both predicate types and dependent types. This algorithm is a key part of the type inference process because it has a direct impact on the number of type annotations that must be provided. There is no single perfect unification algorithm for this type system, due to the problem of "unifying" type invariants. Our algorithm, which is

$$
\begin{array}{rcl}
\mathsf{check}_\sigma(\Gamma, t, \texttt{forall}\ \overline{a}.\ \tau) & \mapsto & \mathsf{check}_\tau(\Gamma, t, \tau[\overline{a \mapsto \hat{a}}]) \\
& \textsf{if} & \overline{a} \notin \mathsf{FTV}(\Gamma),\ \overline{\hat{a}}\ \textit{fresh} \\
\\
\mathsf{check}_\tau(\Gamma \cup \{x : \sigma_x\}, x, \tau) & \mapsto & \tau_x := \mathit{inst}(\sigma_x), \\
& & \tau_x \sim \tau \\
& \textsf{if} & \vdash_\Gamma \tau_x \overset{x}{\preceq} \tau \\
\mathsf{check}_\tau(\Gamma \cup \{C : \sigma_c\}, C, \tau) & \mapsto & \tau_c := \mathit{inst}(\sigma_c), \\
& & \tau_c \sim \tau \\
& \textsf{if} & \vdash_\Gamma \tau_c \overset{C}{\preceq} \tau \\
\mathsf{check}_\tau(\Gamma, i, \tau) & \mapsto & \texttt{Int} \sim \tau \\
\mathsf{check}_\tau(\Gamma, (e_1, \ldots, e_k), \tau) & \mapsto & \tau \overset{\text{tuple}}{\rightsquigarrow} (x_1{:}\tau_1, \ldots, x_k{:}\tau_k), \\
& & \mathsf{check}_\tau(\Gamma, e_i, \tau_i[x_1 \mapsto e_1][\ldots][x_{i-1} \mapsto e_{i-1}]) \\
& \textsf{if} & \vdash_\Gamma (x_1{:}\tau_1, \ldots, x_k{:}\tau_k) \overset{(e_1,\ldots,e_k)}{\preceq} \tau \\
\mathsf{check}^{\text{prop}}_\tau(\Gamma, \texttt{forall}\ x, P, \tau) & \mapsto & \mathsf{check}^{\text{prop}}_\tau(\Gamma, P, \texttt{Bool}), \\
& & \texttt{Bool} \sim \tau \\
& \textsf{if} & \vdash_\Gamma \texttt{Bool} \overset{\texttt{forall}\ x,\ P}{\preceq} \tau \\
\mathsf{check}_\tau(\Gamma, \backslash\ x\ \texttt{->}\ t, \tau) & \mapsto & \tau \overset{\text{fun}}{\rightsquigarrow} \{y{:}\tau_1\}\ \texttt{->}\ \tau_2, \\
& & \mathsf{check}_\tau(\Gamma \cup \{x : \tau_1\}, t, \tau_2[y \mapsto x]) \\
& \textsf{if} & \vdash_\Gamma \{y{:}\tau_1\}\ \texttt{->}\ \tau_2 \overset{\backslash\ x\ \texttt{->}\ t}{\preceq} \tau \\
\mathsf{check}_\tau(\Gamma, t\ u, \tau) & := & \tau_t := \mathsf{infer}_\tau(\Gamma, t), \\
& & \tau_t \overset{\text{fun}}{\rightsquigarrow} \{x{:}\tau_1\}\ \texttt{->}\ \tau_2, \\
& & \mathsf{check}_\tau(\Gamma, u, \tau_1), \\
& & \tau_2[x \mapsto u] \sim \tau \\
& \textsf{if} & \vdash_\Gamma \tau_2[x \mapsto u] \overset{t\ u}{\preceq} \tau \\
\mathsf{check}_\tau(\Gamma, \texttt{let}\ x = u\ \texttt{in}\ t, \tau) & \mapsto & \sigma := \mathsf{infer}_\sigma(\Gamma, u), \\
& & \mathsf{check}_\tau(\Gamma \cup \{x : \sigma := u\}, t, \tau) \\
\mathsf{check}_\tau(\Gamma, \texttt{let}\ x{:}\sigma = u\ \texttt{in}\ t, \tau) & \mapsto & \mathsf{check}_\sigma(\Gamma, u, \sigma), \\
& & \mathsf{check}_\tau(\Gamma \cup \{x : \sigma := u\}, t, \tau) \\
\mathsf{check}_\tau(\Gamma, \texttt{if}\ \texttt{|}\ g_1\ \texttt{->}\ e_1\ \ldots\ \texttt{|}\ g_n\ \texttt{->}\ e_n, \tau) & \mapsto & \mathsf{check}_\tau(\Gamma, g_i, \texttt{Bool}), \\
& & \mathsf{check}_\tau(\Gamma \cup \{g_i\}, e_i, \tau) \\
\mathsf{check}_\tau(\Gamma, \texttt{case}\ t\ \texttt{of}\ \{pat_1\ \texttt{->}\ e_1\texttt{;}\ \ldots\texttt{;}\ pat_n\ \texttt{->}\ e_n\}, \tau) & \mapsto & \tau_s := \mathsf{infer}_\tau(\Gamma, t), \\
& & \Gamma_{pat_i} := \mathsf{check}_{\text{pat}}(\Gamma, pat_i, \tau_s), \\
& & \mathsf{check}_\tau(\Gamma \cup \Gamma_{pat_i} \cup \{t \overset{\text{case}}{=} pat_i\}, e_i, \tau)
\end{array}
$$

Figure 5.13: $\mathcal{H}_{\text{SPEC}}$-kernel type checking algorithm.

shown in Figure 5.16, is intended to be as simple as possible. A very sophisticated algorithm may actually be considered counterproductive, because the programmer should be able to understand how type inference works and what the limits are.

An important difference resides in the asymmetry of the $\sim$ relation for this system,

$$
\begin{array}{rcl}
\mathsf{infer}_{\sigma}(\Gamma, t) & := & \mathit{gen}(\Gamma, \tau) \\
& \textsf{where} & \tau := \mathsf{infer}_{\tau}(\Gamma, t)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{infer}_{\tau}(\Gamma \cup \{x : \sigma\}, x) & := & \mathit{inst}(\sigma) \\
\mathsf{infer}_{\tau}(\Gamma \cup \{C : \sigma\}, C) & := & \mathit{inst}(\sigma) \\
\mathsf{infer}_{\tau}(\Gamma, i) & := & \texttt{Int} \\
\mathsf{infer}_{\tau}(\Gamma, (e_1, \ldots, e_k)) & := & (\tau_1, \ldots, \tau_k) \\
& \textsf{where} & \tau_i := \mathsf{infer}_{\tau}(\Gamma, e_i) \\
\mathsf{infer}^{\mathrm{prop}}_{\tau}(\Gamma, \texttt{forall}\ x,\ P) & := & \texttt{Bool} \\
& \textsf{where} & \mathsf{check}^{\mathrm{prop}}_{\tau}(\Gamma, P, \texttt{Bool}) \\
\mathsf{infer}_{\tau}(\Gamma, \backslash\ x\ \texttt{->}\ t) & := & \{x : \mathring{\tau}_1\}\ \texttt{->}\ \tau_2 \\
& \textsf{where} & \mathring{\tau}_1\ \mathit{fresh} \\
& & \tau_2 := \mathsf{infer}_{\tau}(\Gamma \cup \{x : \mathring{\tau}_1\}, t) \\
\mathsf{infer}_{\tau}(\Gamma, t\ u) & := & \tau_2[x \mapsto u] \\
& \textsf{where} & \tau_t := \mathsf{infer}_{\tau}(\Gamma, t) \\
& & \tau_t \overset{\mathrm{fun}}{\rightsquigarrow} \{x : \tau_1\}\ \texttt{->}\ \tau_2 \\
& & \mathsf{check}_{\tau}(\Gamma, u, \tau_1) \\
\mathsf{infer}_{\tau}(\Gamma, \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t) & := & \tau_2[x \mapsto u] \\
& \textsf{where} & \sigma := \mathsf{infer}_{\sigma}(\Gamma, u) \\
& & \tau := \mathsf{infer}_{\tau}(\Gamma \cup \{x : \sigma := u\}, t) \\
\mathsf{infer}_{\tau}(\Gamma, \texttt{let}\ x{:}\sigma\ \texttt{=}\ u\ \texttt{in}\ t) & := & \tau_2[x \mapsto u] \\
& \textsf{where} & \mathsf{check}_{\sigma}(\Gamma, u, \sigma) \\
& & \tau := \mathsf{infer}_{\tau}(\Gamma \cup \{x : \sigma := u\}, t) \\
\mathsf{infer}_{\tau}(\Gamma, \texttt{if}\ \texttt{|}\ g_1\ \texttt{->}\ e_1\ \ldots\ \texttt{|}\ g_n\ \texttt{->}\ e_n) & := & \mathring{\tau} \\
& \textsf{where} & \mathsf{check}_{\tau}(\Gamma, g_i, \texttt{Bool}) \\
& & \mathring{\tau}\ \mathit{fresh} \\
& & \mathsf{check}_{\tau}(\Gamma \cup \{g_i\}, e_i, \mathring{\tau}) \\
\mathsf{infer}_{\tau}(\Gamma, \texttt{case}\ t\ \texttt{of}\ \{pat_1\ \texttt{->}\ e_1\texttt{;}\ \ldots\ \texttt{;}pat_n\ \texttt{->}\ e_n\}) & := & \mathring{\tau} \\
& \textsf{where} & \tau_s := \mathsf{infer}_{\tau}(\Gamma, t) \\
& & \Gamma_{pat_i} := \mathsf{check}_{\mathrm{pat}}(\Gamma, pat_i, \tau_s) \\
& & \mathring{\tau}\ \mathit{fresh} \\
& & \mathsf{check}_{\tau}(\Gamma \cup \Gamma_{pat_i} \cup \{t \overset{\mathrm{case}}{=} pat_i\}, e_i, \mathring{\tau})
\end{array}
$$

Figure 5.14: $\mathcal{H}_{\mathrm{SPEC}}$-kernel type inference algorithm.

i.e. $\tau \sim \upsilon$ is not equivalent to $\upsilon \sim \tau$. Whenever the typing algorithm call unification it is trying to determine whether the actual type τ_a of a term matches the expected type τ_e for that term. The right way to call unification is $\tau_a \sim \tau_e$, with the actual type as the left argument and the expected type as the second. This is because the unification of two predicate types $\{x{:}\tau \,|\, P\} \sim \{y{:}\upsilon \,|\, Q\}$ is done by unifying the base type of the

$$
\begin{array}{rcl}
\mathsf{check}_{\text{pat}}(\Gamma, x, \tau) & := & \{x : \tau\} \\
\mathsf{check}_{\text{pat}}(\Gamma, i, \tau) & := & \emptyset \\
 & \text{where} & \tau \sim \texttt{Int} \\
\mathsf{check}_{\text{pat}}(\Gamma, (x_1, \ldots, x_k), \tau) & := & \bigcup_{i \in [1,k]} (x_i : \tau_i[y_1 \mapsto x_1][\ldots][y_{i-1} \mapsto x_{i-1}]) \\
 & \text{where} & \tau \overset{\text{tuple}}{\rightsquigarrow} (x_1 : \tau_1, \ldots, x_k : \tau_k) \\
\mathsf{check}_{\text{pat}}(\Gamma \cup \{C : \sigma_c\}, C\ x_1\ \ldots\ x_n, \tau) & := & \bigcup_{i \in [1,n]} (x_i : \upsilon_i[y_1 \mapsto x_1][\ldots][y_{i-1} \mapsto x_{i-1}]) \\
 & \text{where} & \{y_1 : \upsilon_1\} \texttt{ -> } \ldots \texttt{ -> } \{y_n : \upsilon_n\} \texttt{ -> } T\ \tau_1\ \ldots\ \tau_k := inst(\sigma_c) \\
 & & \tau \sim T\ \tau_1\ \ldots\ \tau_k
\end{array}
$$

Figure 5.15: $\mathcal{H}_{\text{SPEC}}$-kernel algorithm for checking patterns.

actual type with the expected type, that is $\tau \sim \{y : \upsilon \mid Q\}$.

The extension of the ς substitution became slightly more complicated and thus we introduced a new operator $\leftarrowtail$. We write $\mathring{\tau} \overset{\varsigma}{\leftarrowtail} \upsilon$ to mean that the meta type $\mathring{\tau}$ should be set to υ. Parametrized types make necessary to check for cycles, e.g. $\mathring{\tau}$ and $[\mathring{\tau}]$ cannot be unified. The other fundamental property of this unification algorithm is that meta type variables can only be assigned types of a restricted form, called κ-types.

Definition 5.8 (κ-types). Roughly, a κ-type is a type that only makes reference to built-in or global definitions. Given a type σ, $\kappa(\sigma)$ is a κ-type derived from σ by removing any reference to local definitions, such that $\sigma \equiv_\mu \kappa(\sigma)$.

In our opinion it would be possible to let unification assign arbitrary types to meta type variables, but that would introduce too much complexity. We use κ-types to prevent a meta type be assigned a type containing references to variables that are not in scope at the location in which the meta type was introduced.

Finally, we introduce two helpers $\overset{\text{fun}}{\rightsquigarrow}$ and $\overset{\text{tup}}{\rightsquigarrow}$, that are used during type-inference to match some arbitrary type τ against a function and a tuple type, respectively. If $\mu_0(\tau) = \{x : \tau_1\} \texttt{ -> } \tau_2$ then $\tau \overset{\text{fun}}{\rightsquigarrow} \{x : \tau_1\} \texttt{ -> } \tau_2$; otherwise τ is unified against an arbitrary function type $\tau \sim \mathring{\tau}_1 \texttt{ -> } \mathring{\tau}_2$ and, if the unification succeeds, then $\tau \overset{\text{fun}}{\rightsquigarrow} \mathring{\tau}_1 \texttt{ -> } \mathring{\tau}_2$. The matching $\tau \overset{\text{tuple}}{\rightsquigarrow} (x_1 : \tau_1, \ldots, x_k : \tau_k)$ is defined analogously. The matching cannot be performed just by unification because type dependencies would be removed when restricting to κ-types.

$$
\begin{array}{rcll}
\alpha \overset{\varsigma}{\sim} \alpha & \mapsto & \varsigma & \\
\mathring{\tau} \overset{\varsigma}{\sim} \mathring{\tau} & \mapsto & \varsigma & \\
\mathring{\tau}_1 \overset{\varsigma}{\sim} \tau_2 & \mapsto & \varsigma(\mathring{\tau}_1) \overset{\varsigma}{\sim} \tau_2 & \text{if } \mathring{\tau}_1 \in dom(\varsigma) \\
\mathring{\tau}_1 \overset{\varsigma}{\sim} \tau_2 & \mapsto & \mathring{\tau}_1 \overset{\varsigma}{\leftarrow} \varsigma(\tau_2) & \text{if } \mathring{\tau}_1 \notin dom(\varsigma) \\
\tau_1 \overset{\varsigma}{\sim} \mathring{\tau}_2 & \mapsto & \tau_1 \overset{\varsigma}{\sim} \varsigma(\mathring{\tau}_2) & \text{if } \mathring{\tau}_2 \in dom(\varsigma) \\
\tau_1 \overset{\varsigma}{\sim} \mathring{\tau}_2 & \mapsto & \mathring{\tau}_2 \overset{\varsigma}{\leftarrow} \varsigma(\tau_1) & \text{if } \mathring{\tau}_2 \notin dom(\varsigma) \\
T\ \tau_1\ \ldots\ \tau_n \overset{\varsigma}{\sim} T\ \upsilon_1\ \ldots\ \upsilon_n & \mapsto & \Sigma_i\ \tau_i \overset{\varsigma}{\sim} \upsilon_i & \\
\{x{:}\tau \mid P\} \overset{\varsigma}{\sim} \upsilon & \mapsto & \tau \overset{\varsigma}{\sim} \upsilon & \\
\tau \overset{\varsigma}{\sim} \{x{:}\upsilon \mid P\} & \mapsto & \upsilon \overset{\varsigma}{\sim} \tau & \\
(\tau_1, \ldots, \tau_k) \overset{\varsigma}{\sim} (\upsilon_1, \ldots, \upsilon_k) & \mapsto & \Sigma_i^{\varsigma}\ \tau_i \overset{\varsigma}{\sim} \upsilon_i & \\
\{x{:}\tau_1\} \texttt{->} \tau_2 \overset{\varsigma}{\sim} \{y{:}\upsilon_1\} \texttt{->} \upsilon_2 & \mapsto & \tau_2 \overset{\varsigma'}{\sim} \upsilon_2 & \\
 & \text{where} & \varsigma' := \upsilon_1 \overset{\varsigma}{\sim} \tau_1 & \\
\tau \overset{\varsigma}{\sim} \upsilon & \mapsto & \text{fail} & \text{otherwise} \\
 & & & \\
\mathring{\tau} \overset{\varsigma}{\leftarrow} \upsilon & \mapsto & \text{fail} & \text{if } \mathring{\tau} \in \mathsf{MTV}(\upsilon) \\
\mathring{\tau} \overset{\varsigma}{\leftarrow} \upsilon & \mapsto & \varsigma \cup \{\mathring{\tau} \mapsto \kappa(\upsilon)\} & \text{otherwise}
\end{array}
$$

Figure 5.16: Unification of types.

5.5 Pattern types

Pattern types are (essentially) a syntactic extension of predicate subtypes that replace the variable ranging over the elements of the base type by an arbitrary pattern. The changes that this extension introduces to the syntax of $\mathcal{H}_{\text{SPEC}}$ are minimal, and are shown in Figure 5.17. Besides predicate subtypes they are also reflected in function types and lambda abstractions, which now bind patterns instead of simple variables. In $\mathcal{H}_{\text{SPEC}}$ tuple types are also allow to bind patterns, but to support this feature is basically the same thing that supporting patterns in function types so it is omitted here for ease of presentation.

Figure 5.18 shows the rules that are modified for the declarative presentation of the type system. We write $\{pat{:}\tau\}$ to mean a pure pattern type, which stands for $\{pat{:}\tau \mid \texttt{True}\}$. The main complication introduced by pattern types is the instantiation of the range of a function type with a specific term, such as in rule App. When typing a term $t\ u$, every occurrence of a variable bound by the pattern pat specified in the domain

Monotypes ($\mathcal{T}$)	τ, υ	::=	...	
		|	$\{pat : \tau \mid P\}$	Predicate subtype
		|	...	
		|	$\{pat : \tau_1\}$ `->` τ_2	Dependent function
Terms (E)	e, g, t, u	::=	...	
		|	\ pat `->` t	λ-abstraction
		|	...	

Figure 5.17: Changes in syntax introduced by pattern-types.

of t should be replaced by some appropriate subterm of u according to pat. We write $\tau[pat]$ to make explicit that the type τ is parametrized by the pattern pat, and $\tau[t]$ to mean the instantiation of τ with t. Such instantiation can be performed in several ways, and since a detailed description may be very clumsy we preferred to illustrate it with an example. Consider the function type `{(x,y):(Int,Int)} -> {z:Int|z <= (x - y)}`, the instantiation of `{z:Int|z <= (x - y)}` could be: *a*) `{z:Int|z <= (t - u)}` for a term `(t,u)`; and *b*) `{z:Int| case e of {(t,u) -> z <= (t - u)}}` for some term `e`. The instantiation may also fail if the syntactic form of the term is incompatible with the pattern, constituting a typing error.

Figure 5.20 shows the rules that are modified for the syntax-directed presentation of the type system. In order to present these rules more concisely, we introduce a new notation $\tau[pat]$ which stands for $\tau[pat2exp(pat)]$. Notice that the syntax-directed Abs rule require to guess a type for the elements matching some pattern pat, and thus a new judgment $\Gamma \vdash^{\text{pat}}_{\Uparrow} pat : \tau, \Gamma_{pat}$ is specified in Figure 5.19.

$$\boxed{\Gamma \vdash \sigma}$$

$$\textsc{SubsetTy}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash^{\text{pat}} pat : \tau, \Gamma_{pat} \qquad \Gamma, \Gamma_{pat} \vdash^{\text{prop}} P : \texttt{Bool}}{\Gamma \vdash \{pat{:}\tau \mid P\}}$$

$$\textsc{FunTy}\ \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash^{\text{pat}} pat : \tau_1, \Gamma_{pat} \qquad \Gamma, \Gamma_{pat} \vdash \tau_2}{\Gamma \vdash \{pat{:}\tau_1\} \texttt{ -> } \tau_2}$$

$$\boxed{\Gamma \vdash t : \sigma}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash^{\text{pat}} pat : \tau_1, \Gamma_{pat} \qquad \Gamma, \Gamma_{pat} \vdash t : \tau_2}{\Gamma \vdash \backslash\ pat \texttt{ -> } t : \{pat{:}\tau_1\} \texttt{ -> } \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash t : \{pat{:}\tau_1\} \texttt{ -> } \tau_2[pat] \qquad \Gamma \vdash u : \{pat{:}\tau_1\}}{\Gamma \vdash t\ u : \tau_2[u]}$$

Figure 5.18: Changes in declarative typing rules introduced by pattern-types.

$$\boxed{\Gamma \vdash^{\text{pat}}_{\Uparrow} pat : \tau, \Gamma_{pat}}$$

$$\textsc{VarPat}\ \frac{}{\vdash^{\text{pat}}_{\Uparrow} x : \tau, \{x : \tau\}} \qquad \textsc{IntPat}\ \frac{}{\Gamma \vdash^{\text{pat}}_{\Uparrow} i : \texttt{Int}, \emptyset} \qquad \textsc{TuplePat}\ \frac{}{\vdash^{\text{pat}}_{\Uparrow} (x_1, \ldots, x_k) : (\tau_1, \ldots, \tau_k), \bigcup_{i \in [1,k]} (x_i : \tau_i)}$$

$$\textsc{ConstrPat}\ \frac{\vdash^{\text{inst}} \sigma_c \leq \{y_1{:}\upsilon_1\} \texttt{ -> } \ldots \texttt{ -> } \{y_n{:}\upsilon_n\} \texttt{ -> } T\ \tau_1\ \ldots\ \tau_k}{\Gamma, C : \sigma_c \vdash^{\text{pat}}_{\Uparrow} C\ x_1\ \ldots\ x_n : T\ \tau_1\ \ldots\ \tau_k, \bigcup_{i \in [1,n]} (x_i : \upsilon_i[y_1 \mapsto x_1][\ldots][y_{i-1} \mapsto x_{i-1}])}$$

Figure 5.19: Syntax-directed inference rules for $\mathcal{H}_{\text{SPEC}}$-kernel patterns.

$$\boxed{\Gamma \vdash_{\Downarrow} t : \tau}$$

$$\textsc{Abs}\quad \frac{\mu_0(\tau) = \{pat' : \tau_1\} \texttt{->} \tau_2[pat'] \qquad \Gamma \vdash^{\text{pat}} pat : \tau_1, \Gamma_{pat} \qquad \Gamma, \Gamma_{pat} \vdash_{\Downarrow} t : \tau_2[pat] \qquad \Gamma \vdash^{\text{co}} \{pat' : \tau_1\} \texttt{->} \tau_2 \overset{\texttt{\textbackslash}\ pat\ \texttt{->}\ t}{\preceq} \tau}{\Gamma \vdash_{\Downarrow} \texttt{\textbackslash}\ pat\ \texttt{->}\ t : \tau}$$

$$\textsc{App}\quad \frac{\Gamma \vdash_{\Uparrow} t : \tau_t \qquad \mu_0(\tau_t) = \{pat : \tau_1\} \texttt{->} \tau_2[pat] \qquad \Gamma \vdash_{\Downarrow} u : \{pat : \tau_1\} \qquad \Gamma \vdash^{\text{co}} \tau_2[u] \overset{t\ u}{\preceq} \tau}{\Gamma \vdash_{\Downarrow} t\ u : \tau}$$

$$\boxed{\Gamma \vdash_{\Uparrow} t : \tau}$$

$$\textsc{Abs}\quad \frac{\Gamma \vdash^{\text{pat}}_{\Uparrow} pat : \tau_1, \Gamma_{pat} \qquad \Gamma, \Gamma_{pat} \vdash_{\Uparrow} t : \tau_2}{\Gamma \vdash_{\Uparrow} \texttt{\textbackslash}\ pat\ \texttt{->}\ t : \{pat : \tau_1\} \texttt{->} \tau_2}$$

$$\textsc{App}\quad \frac{\Gamma \vdash_{\Uparrow} t : \tau_t \qquad \mu_0(\tau_t) = \{pat : \tau_1\} \texttt{->} \tau_2[pat] \qquad \Gamma \vdash_{\Downarrow} u : \{pat : \tau_1\}}{\Gamma \vdash_{\Uparrow} t\ u : \tau_2[u]}$$

Figure 5.20: Changes in syntax-directed typing rules introduced by pattern-types.

Chapter 6

Implementation of a Proof-of-Concept $\mathcal{H}_{\text{spec}}$ Compiler

This chapter presents the HSC $\mathcal{H}_{\text{SPEC}}$ Compiler[1], a proof-of-concept implementation in Haskell of the $\mathcal{H}_{\text{SPEC}}$ language and its type system as described in the previous chapters. The development of HSC has two primary motivations. First, it constitutes a reference implementation of our type inference algorithm, providing a definitive interpretation of the specification given in chapter 5. A reference implementation also proves that a specification is practically implementable, and may expose important implementation details. Second, we use HSC to demonstrate how a practical and lightweight approach to correct software development can be build in top of $\mathcal{H}_{\text{SPEC}}$. HSC provides a prototype checker that can be effectively used to find bugs or prove their absence.

First, we describe the compilation process of a $\mathcal{H}_{\text{SPEC}}$ module in § 6.1, and the software architecture of HSC in § 6.2. Section § 6.3 presents the intermediate language used by the HSC checker, which is described next in § 6.5. Finally, § 6.6 explains the lightweight dependently-typed programming techniques that were used in the implementation of HSC. Despite this chapter does not require much background in functional programming, we should advise that the techniques explained in § 6.6 might be difficult to follow for the inexperienced reader.

[1]The source code of HSC is available at `http://files.iagoabal.eu/UdC-EI-PFC/hsc.tar.gz`. See Appendix B for installation instructions.

6.1 Compiler pipeline

The HSC compiler takes a $\mathcal{H}_{\text{SPEC}}$ source file as input and produces a binary CORE file as output. As will be detailed later in § 6.3, a CORE file is an intermediate representation of a source module that is targeted for verification. This transformation process consists of a sequence of steps that together form the HSC pipeline as shown in Figure 6.1. The internal representation of the input module is transformed as it passes through this pipeline. The datatype `Module p` is a representation of a $\mathcal{H}_{\text{SPEC}}$ module, indexed by a parameter `p` that identifies the compilation phase as explained in § 6.6.

The HSC pipeline is more or less the standard for any compiler front-end, except for two extra steps —highlighted in light purple color— that are introduced to generate and filter the verification conditions (VCs) that establish the correctness of the input module. Next, each compilation stage is described in more detail.

6.1.1 Parse

First, the input source file is parsed to construct an abstract syntax tree that represents the input module. The $\mathcal{H}_{\text{SPEC}}$ (module) source file must have the extension *.hss*, which stands for $\mathcal{H}_{\text{SPEC}}$ *source*. This stage performs both the lexical and the grammatical analysis of the source code. If it succeeds, the output is a term of type `Module Pr`.

The parser is described by an LR grammar that, for the sake of simplicity, parses $\mathcal{H}_{\text{SPEC}}$ "generously". The illegal expressions that may be accepted by the parser will be filtered out later during the *renaming* phase. This approach is inherited from the GHC Haskell compiler. An example of this is the relaxed parsing of `'else'` as an ordinary guard expression; the restriction of `'else'` to be the last guard in a list of guarded alternatives is enforced by the renamer. One advantage of this approach is that subsequent stages of compilation tend to produce more helpful error messages.

6.1.2 Rename

This stage is named by its primary task, which is to rename all bound variables in the module to unique names, following Barendregt's variable convention. Such renaming is performed by attaching a unique identifier to each variable, e.g. `\ x -> 1 + x` becomes `\ x`$_k$ `-> 1 + x`$_k$ for some unique k. Barendregt convention is meant to simplify later

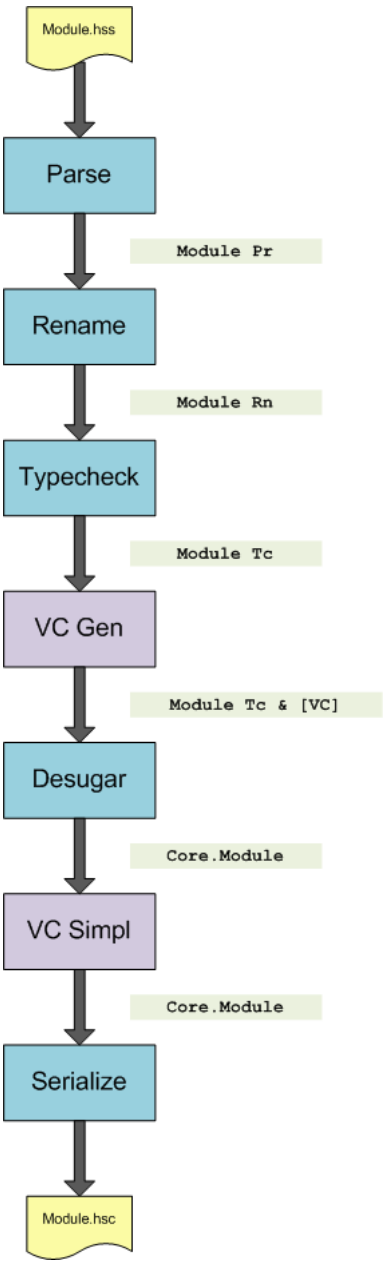


Figure 6.1: Compiler pipeline.

compilation stages, especially typechecking. Also, these unique identifiers allow for many optimizations in the handling of variables, such as the efficient implementation of typing environments.

In addition, the renamer also detects recursive bindings and performs several lexical error checkings such as: variables out of scope, linearity of patterns, etc. Many of these checkings rule out illegal terms accepted by the parser, as explained above. The output

of the renamer is a syntactically valid $\mathcal{H}_{\text{SPEC}}$ module, represented by a term of type `Module Rn`, that will be fed to the typechecker.

6.1.3 Typecheck

HSC typechecks a module following the typing algorithm described in chapter 5. Type inference works internally on a `Module Tc'` term where meta-type variables are used to infer types through unification. The typing process gradually fills every placeholder with its final type, and as a result a new module term of type `Module Tc` is constructed.

During typechecking type coercions resulting from predicate subtyping, i.e. applications of the $\vdash^{\text{co}}$ judgment, are inserted into the abstract syntax tree at the appropriate locations. The typechecker also performs a translation of the input module, which is implicitly-typed, to an explicitly-typed representation. This process, which is known as *term reconstruction*, annotates every bound variable with its type and introduces explicit type abstractions and applications, as in System F.

In HSC typechecking is performed on $\mathcal{H}_{\text{SPEC}}$ modules in their original form, before any desugaring is done. This approach complicates the typechecker a little bit, as it has to handle a more verbose syntax, but it also allows to infer pattern-types easily and to produce more meaningful error messages.

6.1.4 VC Gen

This stage is responsible for generating a set of verification conditions (VCs) whose validity entails the correctness of the input module. These verification conditions arise from different sources. Type-correctness conditions, which express type coercions as logical formulas, are the primary source of VCs. Verification conditions can also arise from the completeness and disjointness constraints that apply to branching constructs, namely case- and if-expressions. Finally, logical goals declared by the programmer within the module also lead to VCs.

In principle, the generation of VCs could take place during typechecking, but that would complicate the implementation too much. Partly because at that point type coercions may involve meta-type variables. The separation of these two compilation phases is also an architectural decision, it is for the purpose of modularity and maintainability.

It is also a natural consequence of the distinction between conventional typechecking and predicate subtyping that we want to enforce. The outcome of the VC generator is a list of VCs that are attached to the module constructed during typechecking.

6.1.5 Desugar

The next step consists of desugaring (i.e. removing syntactic sugar) the input module to translate it into a smaller and more concise intermediate language known as Core. The use of *core* languages is widespread in compiler's middle-ends to perform simplifications and optimizations before code generation. In this context, the core language is carefully chosen to be specially well-suited for the intended transformations, hence simplifying their implementation. Moreover, an intermediate language makes the middle-end fairly independent of the source language, so it is unlikely to be affected by the addition of new syntactic constructs. In the case of HSC, which is focused on verification rather than on code generation, the purpose of the Core language is to simplify the proof of verification conditions. The output of desugaring is a `Core.Module` term, which comprises the $\mathcal{H}_{\text{SPEC}}$ input module plus the list of VCs, both in desugared form.

6.1.6 VC Simpl

Just after desugaring, the compiler tidies up the set of verification conditions, and tries to discharge the most trivial ones. This is done by a rewriting-based VC simplification procedure that is depicted in § 6.4. Essentially, a set of simplification patterns for high-order logic are applied to each VC. In case such rewritings determine the validity of the formula, then the VC is either filtered out if it is valid, or else it is reported as an error —an invalid VC is considered as fatal as a type mismatch. Otherwise, the VC passes to the next stage of the pipeline. Note that this phase is not intended to perform heavy deductive reasoning, since that would slow down the compilaton process.

6.1.7 Serialize

Finally, the Core module resulting from the above transformations is serialized into a binary file with the extension *.hsc*, which stands for $\mathcal{H}_{\text{SPEC}}$ *core*. This file is to be handled by the verification toolset.

6.2 Software architecture

In this section we discuss the high-level architecture of HSC, which follows the standard front-end/back-end pattern implemented by any modern compiler. Here we will present the module structure of this software system, relying in UML package diagram notation. We are mostly interested in describing a view of the application as a hierarchy of modules, in order to depict how the functionality is distributed among the system. In this particular case, we have considered that a UML package is the construct that best fits our concept of module. Unfortunately, there is no standard UML profile defined for the functional paradigm that can be taken as a reference.

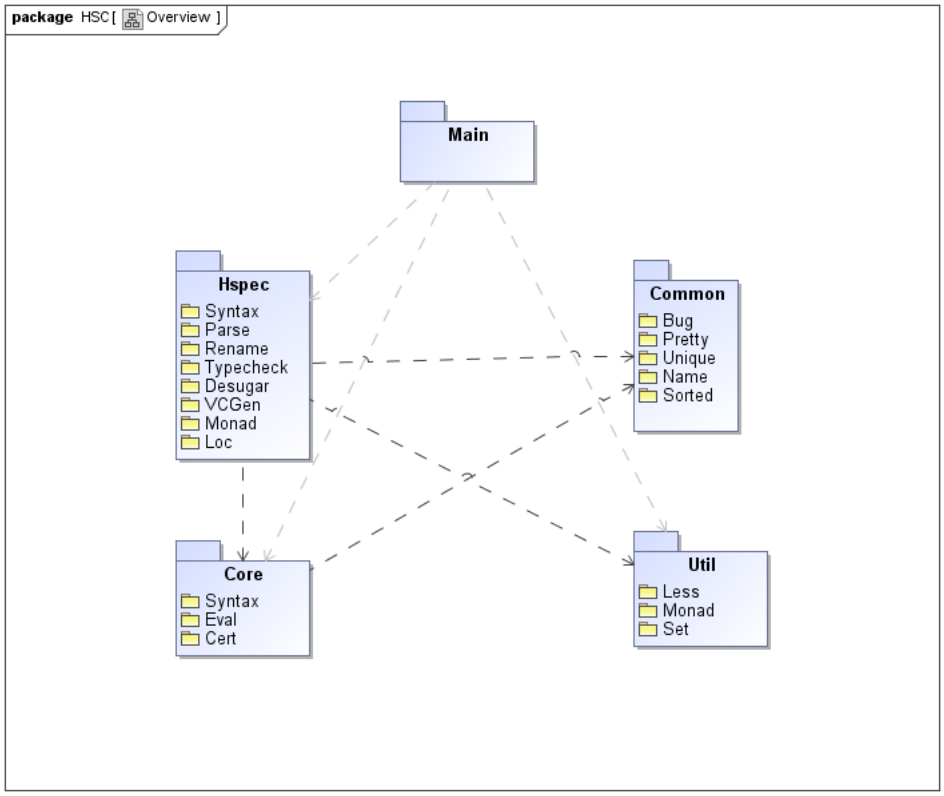


Figure 6.2: The *Overview* module hierarchy.

Figure 6.2 offers a top-level view of the module hierarchy. HSC is divided into five top-level modules. The *Main* module defines the entry point of HSC, it presents the compiler functionality to the user through a simple command-line interface (CLI). This *Main* module delegates into two major modules, *Hspec* and *Core*, that implement most of the functionality. The *Hspec* module implements the front-end functionality, which

includes most of the stages of the compilation pipeline. The *Core* module implements the certification facilities of HSC, it offers middle-end functionality as well as it provides connections to verification back-ends. The *Common* module factorizes common parts between these two major modules. A number of utilities, that do not define any compiler function by themselves, are collected in the *Utils* module.

6.2.1 Common

The *Common* module, detailed in Figure 6.3, is a collection of five sub-modules that offer functionality shared by the main components of the compiler.

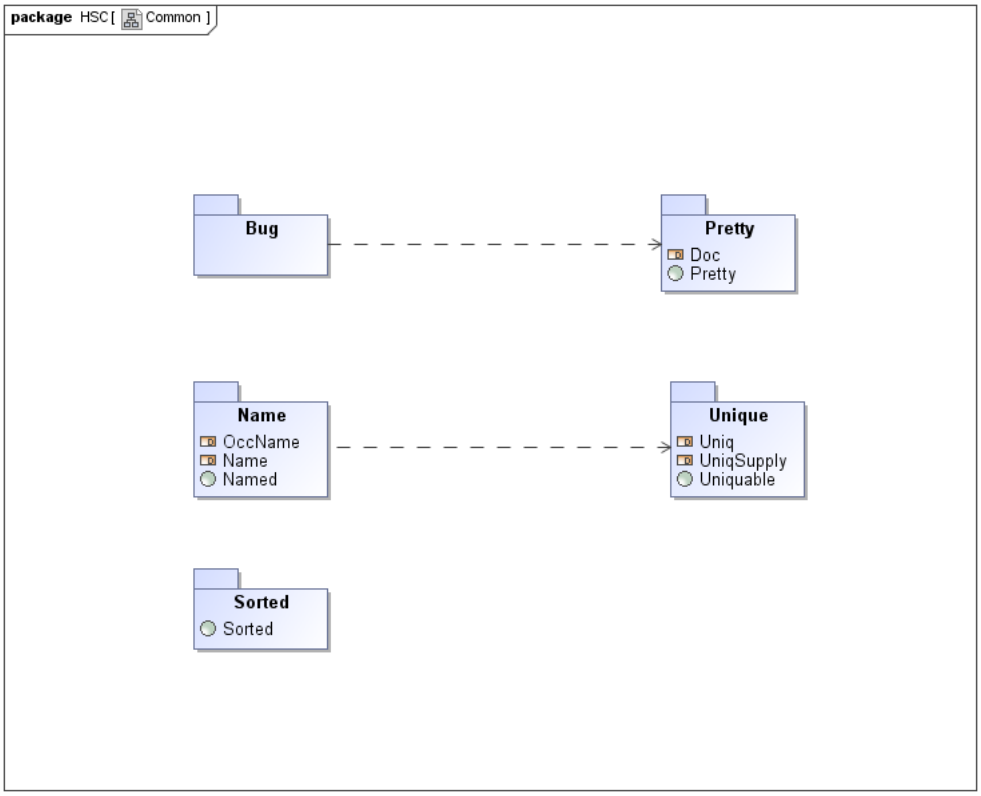


Figure 6.3: The *Common* module hierarchy.

▷ **Bug** offers a number of macros that help locating the source of common exceptions in Haskell programs.

▷ **Pretty** defines a pretty-printing combinator library specialized for source code. The library offers a number of styles and rendering modes that allow to control

things like indentation, length of lines, etc.

▷ **Unique** declares unique identifiers as well as the concept of *unique supply*, and defines several combinators and facilities to operate with them.

▷ **Name** offers types and facilities to deal with the naming of $\mathcal{H}_{\text{SPEC}}$ objects, like variables or data types.

▷ **Sorted** defines a class (i.e. a generic interface) of *sortable* objects. Types instantiating this class must define a `sortOf` function that computes *sorts* for terms of that type.

6.2.2 Hspec

The *Hspec* module implements the fron-end of HSC, which involves most of the functionality of the compilation pipeline. The *Syntax* sub-module defines the abstract syntax of $\mathcal{H}_{\text{SPEC}}$ and provides plenty of syntax-related facilities. The modules *Parse*, *Rename*, *Typecheck*, *VCGen* and *Desugar* are built around the *Syntax* module, and each of them implement the corresponding compiler stage (given by its name). Figure 6.4 shows the sub-module structure of this module, which is detailed next.

Syntax

The *Syntax* module hierarchy deals with $\mathcal{H}_{\text{SPEC}}$'s syntax. This module declares the abstract syntax tree of $\mathcal{H}_{\text{SPEC}}$, provides convenient smart constructors and queries, and also offers implementations for syntactic concepts such as free variables and substitution.

▷ **AST** defines the abstract syntax tree of $\mathcal{H}_{\text{SPEC}}$.

▷ **Expr** defines smart constructors and queries for expressions.

▷ **FTV** implements FTV, the set of free type variables.

▷ **FV** implements FV, the set of free variables.

▷ **MTV** implements MTV, the set of meta type variables.

▷ **Pattern** defines smart constructors and queries for patterns.

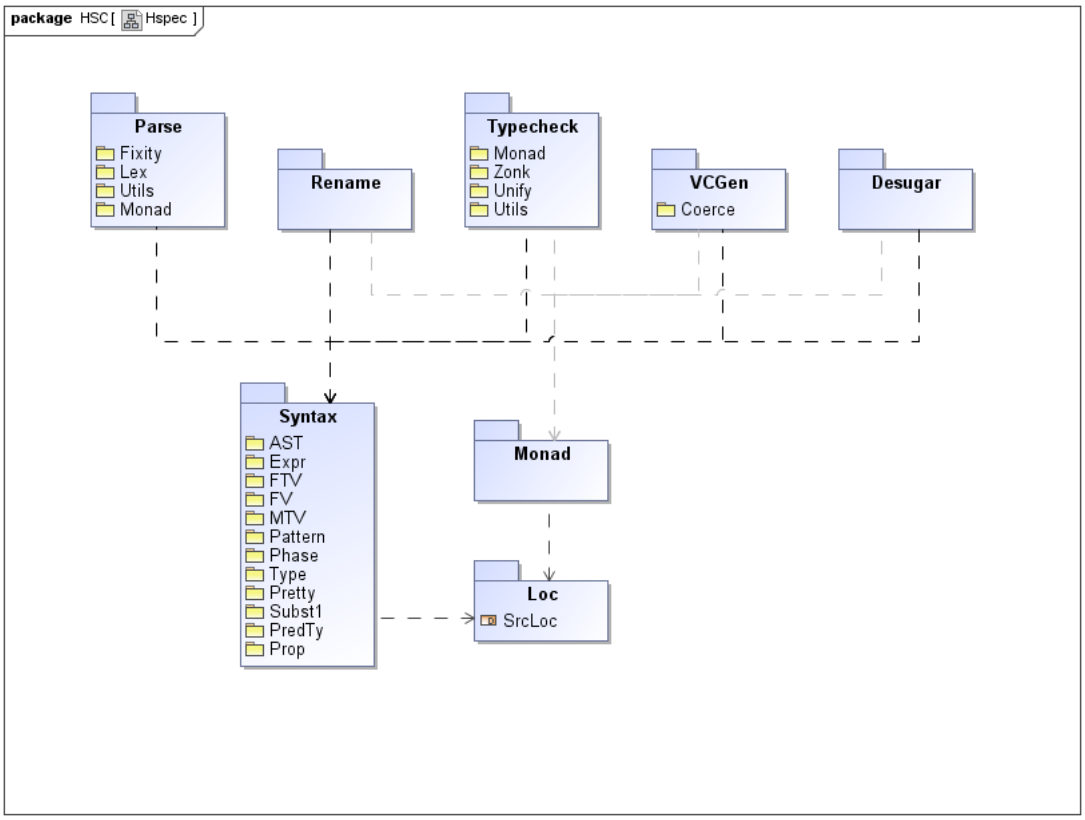


Figure 6.4: The *Hspec* module hierarchy.

▷ **Phase** reifies the concept of compilation phase.

▷ **Type** defines smart constructors and queries for types.

▷ **Pretty** offers pretty-printing functions for the abstract syntax.

▷ **Subst1** implements variable substitution.

▷ **Prop** defines smart constructors and utilities for handling logical formulas.

Parse

The *Parse* module implements the parsing step of the compilation pipeline. The module itself implements the LR grammar and exports the functions for parsing $\mathcal{H}_{\text{SPEC}}$ files. Many of the functionality is delegated into the following private sub-modules:

▷ **Fixity** defines a pass over the abstract syntax to fix infix applications.

- ▷ **Lex** declares language tokens and functions for lexical analysis.
- ▷ **Utils** provides several helpers to construct the abstract syntax tree.
- ▷ **Monad** provides the framework in which parsing takes place.

Rename

The *Rename* module implements the renaming step of the compilation pipeline.

Typecheck

The *Typecheck* module implements the typechecking step of the compilation pipeline. The module itself implements the typing rules, while a number of private sub-modules are responsible for auxiliary functions:

- ▷ **Monad** provides a framework in which typechecking takes place, this framework is a specialization of the one provided by the *Hspec.Monad* module.
- ▷ **Zonk** implements meta-type variable substitution.
- ▷ **Unify** implements the unification algorithm.
- ▷ **Utils** defines several typechecking utilities.

Desugar

The *Desugar* module implements the desugaring step of the compilation pipeline.

VCGen

The *VCGen* module implements the generation of verification conditions during the compilation process.

- ▷ **Coerce** implements the translation of type coercions into logical formulas given in Definition 5.2.

Monad

The *Hspec.Monad* module implements a generic framework in which all of the central stages of compilation (those excluding parsing and serialization) take place. This framework provides several services such as logging, error localization, error handling, etc. These services are implemented once and shared by all the above compilation stages, which built their own customized framework on top of this, to fit their particular needs. Thanks to sharing a common implementation for these basic services, compilation phases can easily be chained together into a pipeline.

Loc

This module offers types and combinators that lead with the manipulation of source locations, mostly used to point where errors occur in the source code.

6.2.3 Core

The *Core* module provides services to manipulate and transform Core programs. It defines the abstract syntax of Core as well as a number of syntactic facilities. It also implements an interpreter that provides the basic functionality to evaluate Core terms. The internal structure of this module is shown in Figure 6.5.

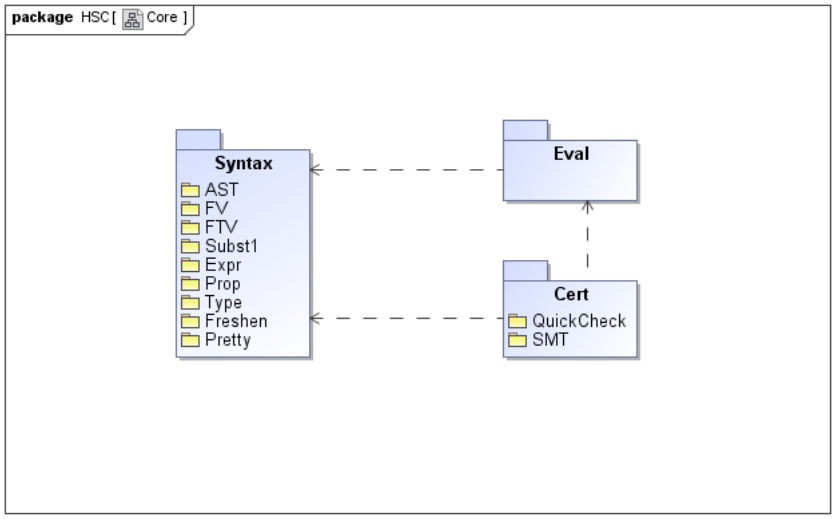


Figure 6.5: The *Core* module hierarchy.

Syntax

The *Syntax* module hierarchy deals with CORE's syntax. This module declares the abstract syntax tree of CORE, provides convenient smart constructors and queries, and also offers implementations for syntactic concepts such as free variables and substitution.

- ▷ **AST** defines the abstract syntax of CORE.
- ▷ **Expr** defines smart constructors and queries for expressions.
- ▷ **FTV** implements FTV, the set of free type variables.
- ▷ **FV** implements FV, the set of free variables.
- ▷ **Freshen** defines a pass to rename variables assigning fresh unique identifiers.
- ▷ **Type** defines smart constructors and queries for types.
- ▷ **Pretty** offers pretty-printing functions for the abstract syntax.
- ▷ **Subst1** implements variable substitution.
- ▷ **Prop** defines smart constructors and other utilities specialized for propositions.

Eval

This module implements a *lazy* interpreter for CORE.

Cert

The *Cert* module is the part of HSC responsible for managing the certification of programs. This module provides connections to multiple verification back-ends that can be used to discharge VCs. It is also supposed to keep track of the certification process, and to generate reports that certificate the degree of trust in the correctness of a program. This implementation of HSC, however, only provides support for discharging VCs as a proof of concept demonstration.

- ▷ **QuickCheck** provides a verification back-end based on randomized testing.
- ▷ **SMT** provides an SMT verification back-end, specific for the Yices SMT solver.

6.2.4 Util

The *Util* package collects a number of functions that do belong to any module of this system, but extend the functionality of existing Haskell libraries. Those are divided into three sub-modules as shown in Figure 6.6.

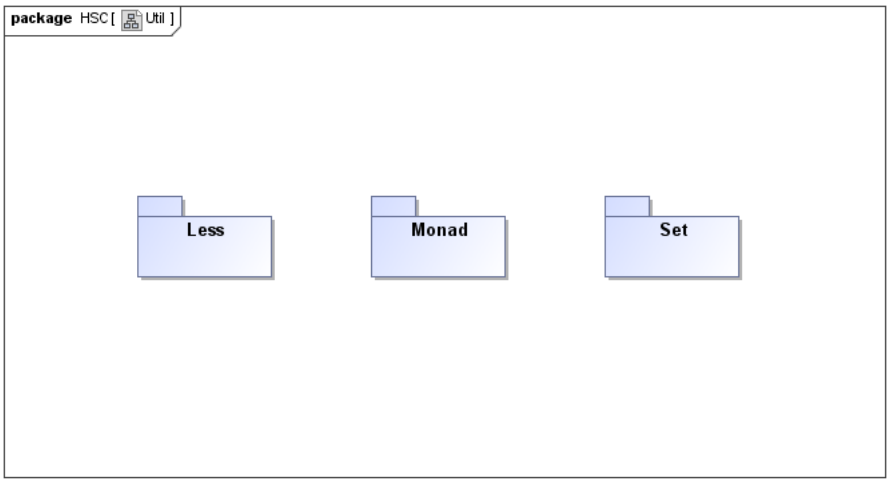


Figure 6.6: The *Util* module hierarchy.

- ▷ **Less** allows to paginate output using the `less` (UNIX) pager program.
- ▷ **Monad** defines functions not included in Haskell's *Control.Monad* module.
- ▷ **Set** defines functions not included in Haskell's *Data.Set* module.

6.3 The Core language

The purpose of core languages in compilers construction is twofold. First, a core language can be precisely designed to suit the objectives of a compiler function, hence simplifying its implementation. Second, it makes the internal machinery of the compiler independent of the source language, at least to some extent. For the same reasons we have defined a core language, called CORE, to be used by the VC checker of HSC. CORE is a slightly desugared version of $\mathcal{H}_{\text{SPEC}}$, that is targeted for establishing connections with verification back-ends.

The design of CORE is a compromise between expressiveness and simplicity. Some syntactic features of $\mathcal{H}_{\text{SPEC}}$, such as pattern types, can be used to express certain prop-

erties succinctly so that facilitating automated analysis. For example, given that `xs` has type {`(_::_):[t]`}, the formula `xs /= []` is trivially `True` because `[]` does not match the pattern `_::_`. However, if the construct is desugared, then the property that it encodes may become more obscure making this kind of reasoning more difficult. Thus, we have decided to keep the language of types as is —see Figure 6.7.

Type variables ($\mathcal{U}$)	a, b, c			
Type constructors	T	$\supseteq$	$\{\texttt{()}, \texttt{Bool}, \texttt{Int}\}$	
Monotypes ($\mathcal{T}$)	τ, υ	::=	a	Type variable
		|	$T\ \tau_1\ \ldots\ \tau_n$	Type constructor
		|	$\texttt{[}\tau\texttt{]}$	Lists
		|	$\{pat\texttt{:}\tau \texttt{\|} P\}$	Predicate subtype
		|	$(pat_1\texttt{:}\tau_1 \texttt{\|} P_1\texttt{,}\ \ldots\ \texttt{,}pat_k\texttt{:}\tau_k \texttt{\|} P_k)$	k-tuples
		|	$\{pat\texttt{:}\tau_1 \texttt{\|} P\}$ `->` τ_2	Dependent function
Polytypes ($\mathcal{S}$)	σ	::=	`forall` $\overline{a}$`.` τ	Type scheme

Figure 6.7: CORE types.

The language of terms must be simplified, however, so that it becomes closer to the logics supported by most verification systems. When it comes to the term language, the verbosity of $\mathcal{H}_{\text{SPEC}}$ hampers the implementation of verification procedures. Figure 6.8 shows a language of CORE terms very similar to that of $\mathcal{H}_{\text{SPEC}}$-kernel —see § 5.1. Most notably: all bound variables are explicitly typed, and both type abstraction and type instantiation are explicit; pattern bindings are restricted to *logical case expressions* —see § 4.4.10; operators can only be applied in (saturated) infix form; and case expressions have the same basic form as in $\mathcal{H}_{\text{SPEC}}$-kernel. Notice that CORE preserves nested patterns in types, as shown in Figure 6.9, but patterns appearing in case-expressions must be simple. In order to distinguish both cases we use *pat* and p to range over nested and simple patterns, respectively.

We decided to keep some of the syntactic sugar of $\mathcal{H}_{\text{SPEC}}$, such as function bindings, in the interest of readability. Despite HSC encourages the use of automated checkers, the programmer may still need to look at the code, e.g. to make sense of some VC.

Term variables ($\mathcal{V}$)	f, x, y, z			
Integers ($\mathbb{Z}$)	i			
Data constructor	C	$\supseteq$	$\{\texttt{()}, \texttt{False}, \texttt{True}\}$	
Terms (E)	e, g, t, u	::=	i	Literal
		|	x	Variable
		|	C	Data constructor
		|	$\texttt{(}e\texttt{)}$	Parenthesized expression
		|	$\texttt{[}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{]}$	List
		|	$e_1\texttt{::}e_2$	*Cons* list
		|	$\texttt{[}e_1\texttt{..}e_2\texttt{]}$	Arithmetic sequence
		|	$\texttt{[}e_1\texttt{,}e_2\texttt{..}e_3\texttt{]}$	Arithmetic sequence
		|	$\texttt{(}e_1\texttt{,} \ldots \texttt{,}e_k\texttt{)}$	k-tuple
		|	$\texttt{\textbackslash}\ \overline{\texttt{(}x\texttt{:}\tau\texttt{)}}\ \texttt{->}\ t$	λ-abstraction
		|	$t\ u$	Application
		|	$\sim e$	Not
		|	$\texttt{-}e$	Unary minus
		|	$e_1\ op\ e_2$	Operator
		|	$\texttt{@}\overline{a}\ \texttt{->}\ e$	Type abstraction
		|	$t\ \texttt{@}\overline{\tau}$	Type application
		|	$\texttt{let}\ \{bind_1\texttt{;} \ldots \texttt{;}bind_n\}\ \texttt{in}\ t$	Let bindings
		|	$\texttt{if}\ g\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	If-then-else
		|	$\texttt{if}\ \texttt{|}\ g_1\ \texttt{->}\ e_1\ \ldots\ \texttt{|}\ g_n\ \texttt{->}\ e_n$	If expression
		|	$\texttt{case}\ t\ \texttt{of}\ \{p_1\ \texttt{->}\ e_1\texttt{;} \ldots \texttt{;}p_n\ \texttt{->}\ e_n\}$	Case expression
		|	$\texttt{caseP}_{\mathbb{B}}\ \texttt{u of pat ->}\ \texttt{P}$	Logical case-expression
		|	$e\ \texttt{:}\ \sigma$	Type annotation
		|	$\triangle\ \overline{\texttt{(}x\texttt{:}\tau\texttt{)}},\ P$	Quantified formula
Propositions ($\mathcal{P}$)	P, Q	::=	e	
Operators	op	$\in$	$\{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{\&\&}, \texttt{||}, \texttt{==>}, \texttt{<=>}, \texttt{==}, \texttt{/=}, \texttt{<=}, \texttt{<}, \texttt{>=}, \texttt{>}\}$	
Bindings	$bind$	::=	$[\texttt{rec}]\ f\ \texttt{@}\overline{a}\ \overline{\texttt{(}x\texttt{:}\tau\texttt{)}}\ \texttt{=}\ e$	
Quantifiers	$\triangle$	::=	$\texttt{forall} \mid \texttt{exists}$	

Figure 6.8: Core terms.

Patterns	pat, p	::=	i	Literal pattern
		|	x	Variable pattern
		|	$C\ pat_1\ \ldots\ pat_n$	Constructor pattern
		|	$\texttt{[]}$	Empty list
		|	$pat_1\texttt{::}pat_2$	*Cons* list
		|	$\texttt{(}pat_1\texttt{,} \ldots \texttt{,}pat_k\texttt{)}$	Tuple pattern

Figure 6.9: Core patterns.

6.4 Simplifying VCs

Generating VCs is mostly a syntax-driven process and often leads to clumsy formulas. This is a real problem that may affect the performance of checkers, or even their ability to check a VC. Therefore, VCs are first tidied up during compilation by applying a number of rewrite patterns and simplifications. This process can be divided into three logical steps —even though in practice these are intertwined—: reduction of formulas, elimination of irrelevant hypothesis, and conversion to prenex normal form.

6.4.1 Reduction of formulas

Formulas are reduced by applying well-known algebraic equalities such as F `||` $F \equiv F$, F `&& True` $\equiv F$ and `False ==>` $F \equiv$ `True`. Additionally, we may also decide some equalities and inequalities such as e `==` $e \equiv$ `True` and e `<` 1 `+` $e \equiv$ `True`, or others more involved by taking advantage of information contained in types. It is worth noting that these reductions do not preserve, in general, total correctness. Nevertheless, we are only interested in partial correctness, and this is maintained.

6.4.2 Elimination of irrelevant hypothesis

Given a formula `forall` $\overline{(x\texttt{:}\tau)}$`,` H_1 `&&` $\ldots$ `&&` H_n `==>` F, we say that an hypothesis H_i is *irrelevant* if its truth value does not affect the truth of the conclusion F. In such a case, one can simply remove H_i from the set of hypothesis without affecting the truth of the formula. For instance, consider the following formula:

```
forall (xs0:[â]) (x:â) (xs:[â]),
    xs0 == x::xs ==> 1 + length @â xs >= 0
```

The hypothesis `xs0 == x::xs` involves the variables `xs0`, `x` and `xs`. However, it does not restrict the value of `xs` in any way, thus it does not affect the truth of the conclusion (that only mentions `xs`) too. The above formula is then equivalent to:

```
forall (xs:[â]), 1 + length @â xs >= 0
```

6.4.3 Conversion to prenex normal form

Finally, a bottom-up traversal of the formula merges and moves quantifiers outward. Here we apply rules such as $\triangle\ \overline{(x:\tau)},\ \triangle\ \overline{(y:v)},\ F \equiv \triangle\ \overline{(x:\tau)}\ \overline{(y:v)},\ F$ and F ==> $\triangle\ \overline{(x:\tau)},\ G \equiv \triangle\ \overline{(x:\tau)},\ F$ ==> G. Accidentally, as a result of these transformations, a formula may end up in prenex normal form.

6.5 Checking VCs

$\mathcal{H}_{\text{SPEC}}$ modules are compiled into core (*.hsc*) files to be managed by the VC checker. A core file contains both the desugared CORE module and the list of VCs that should be proven valid to complete typechecking. The VC checker supports multiple verification back-ends, to fit different needs, and each VC is individually checked using one or more of those. Alternatively, the code could also be instrumented with runtime contract checks. We emphasize a "pay as you go" and lightweight model, where checking is optional and (mostly) automated.

For demonstration purposes we have developed a proof-of-concept VC checker providing two back-ends that handle first-order formulas. The first is based on randomized testing, supporting a large fragment of formulas; whereas the second is based on automated proving, constituting a more restricted but reliable method.

6.5.1 The QuickCheck back-end

QuickCheck [54] is a tool for formulating and testing properties of Haskell programs. Properties are expressed as Haskell terms using QuickCheck combinators, and are tested on random input produced by test data generators. These generators are also Haskell terms built using a combinator library.

This back-end checks a formula by constructing and testing an equivalent QuickCheck property. The process is illustrated by the flowchart shown in Figure 6.10. In the case that the formula is quantifier-free, then we simply use the interpreter to reduce it to a boolean value.

If the formula has quantifiers then it passes through an "optimization" process to make it more suitable for testing. Ideally, the formula is optimized to reduce the burden of generating test data, and then it is converted into prenex normal form. Before

proceeding, the formula is analyzed for unsupported features. Generally, any first-order formula in prenex normal form would be supported. The optimization procedure may eliminate unsupported features so this test is done only after the formula has been optimized.

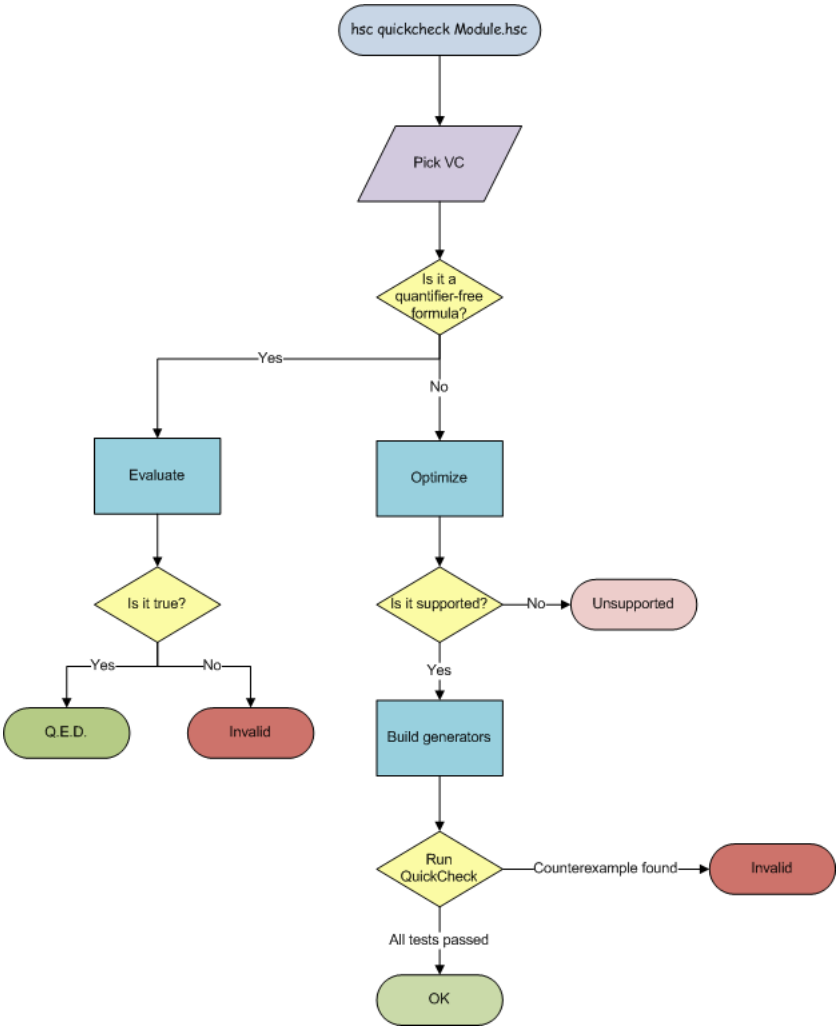


Figure 6.10: The QuickCheck backend.

Then we end up with a formula `forall` $\overline{(x{:}\tau)}$`,` F where F is quantifier-free. The next step builds a generator for every quantified variable x_i of type τ_i. Our prototype supports any non-function type, including predicate types, dependent tuples and inductive types. Finally, a QuickCheck property is constructed using the $\overline{x}$ test data generators to build random environments for evaluating F.

6.5.2 The SMT back-end

SMT solvers [55] are a class of automated theorem provers that deal with decidable fragments of first-order logic modulo background theories. In contrast with general-purpose first-order provers, SMT solvers focus on specialized theories of practical usefulness such as integer arithmetic, arrays or inductive data types. This technology is very effective in practice, and it is becoming increasingly popular in software and (especially) hardware verification.

The SMT back-end checks a VC by translating it into an SMT formula. This formula is then handed to an SMT solver, and the solution is translated back into Core. The flowchart shown in Figure 6.11 depicts the steps involved in this process.

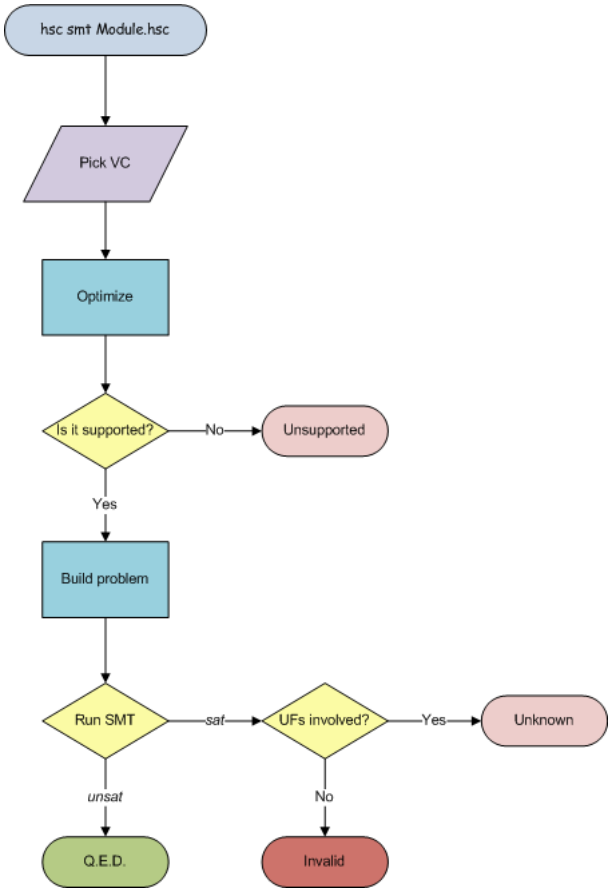


Figure 6.11: The SMT backend.

The VC is first "optimized" for SMT solving; mainly through desugaring of features like patterns types or partial function application, that may not be properly supported by these solvers. In order to make things a lot easier, this back-end relies on the

Yices SMT solver because it supports almost the same form of predicate subtyping and dependent types as $\mathcal{H}_{\text{SPEC}}$.

Finally, the SMT problem is built and passed to Yices. Since SMT cannot handle recursive definitions, these are translated as uninterpreted functions. An uninterpreted function (UF) is only defined by a name and a type, so the use of rich types becomes crucial to characterize a function precisely. If the SMT solver founds no counterexample then the VC is definitely valid. However, if a counterexample is found but it involves UFs, then it is likely caused by a misinterpretation of some function's behavior, and we conservatively report that the correctness of the VC is *unknown*.

6.6 Lightweight dependently-typed programming in HSC

HSC was developed following a (moderate) lightweight dependently-typed programming style —introduced in § 3.3. We have made use of this kind of type hackery specially to define the abstract syntax of $\mathcal{H}_{\text{SPEC}}$. The (ab)use of Haskell's type-level constructs allowed us to encode invariants of the $\mathcal{H}_{\text{SPEC}}$ syntax into its abstract representation. What is more, our abstract syntax representation is indexed by the compilation stage. This means that each stage of the compilation pipeline defines its own view of the abstract syntax.

Compilation stages are reified as data types: `Pr`, `Rn`, `Tc'` and `Tc` represent the *parse*, *rename*, *typecheck* and *VC generation* compiler phases, respectively. We use Haskell type-classes, plus some extensions, to encode a total order `:<:` between compilation phases, thus `Pr :<: Rn :<: Tc' :<: Tc`. In the same way we also reify the *rank* of a type: `TAU` stands for monomorphic (rank-0) types, whereas `SIG` stands for rank-1 polymorphic types.

An abstract type is thus indexed by both the type's rank and the compilation stage, as `Type r p`. Particular instances of this type represent types of rank `r` in the step `p` of the pipeline. For this purpose, we resort to a restricted form of indexed data types known as *generalized algebraic data types* (GADTs). As shown below, a GADT allows to declare a data constructor for just some particular instances of the data type being defined.

```
type Sigma = Type SIG
```

```
type Tau   = Type TAU

data Type r p where
  -- | type variable
  VarTy :: TyVAR p -> Type r p
  -- | type constructor (saturated) application
  ConTy :: TyCON p -> [Tau p] -> Type r p
  ...
  -- | placeholder (unknown) type
  UnkTy :: p :<: Tc' => Type r p
  -- | meta type variable
  MetaTy :: MetaTyVar -> Type r Tc'
  -- | rank-1 polymorphic type (type-scheme)
  ForallTy :: [TyVAR p] -> Tau p -> Sigma p
```

A type-scheme, as introduced by the `ForallTy` constructor, has type `Sigma p` indicating that it is indeed a rank-1 type. Remarkably, the fact that `ForallTy` only constructs truly rank-1 types is enforced by its type signature —notice the type of its second argument. In the same way, the type of `ConTy` reflects that the arguments of a type constructor must be monomorphic.

The phase parameter is also used to evince the transformation processes that take place through the compilation pipeline. The *unknown* type —`UnkTy`— is an implementation-level concept, that can be used as a placeholder to be filled by the typechecker. Notice that unknown types are restricted to compilation phases preceding typechecking; `UnkTy` cannot appear where `Type r p` is expected if `p :<: Tc'` does not hold. Analogously, meta type variables are only allowed to occur during typechecking.

The compilation stage also determines another aspects of the abstract syntax, such as the representation of variables.

```
type family TyVAR phase
type instance TyVAR Pr  = OccName
type instance TyVAR Rn  = Name
type instance TyVAR Tc' = TyVar
type instance TyVAR Tc  = TyVar
```

Here `TyVAR` is a type function, that is, a function from types to types. It maps a phase-type to the representation of type variables that is used during that compilation phase.

For instance, after parsing variables are represented by `OccName`'s (basically strings), and those are converted into `Name`'s once the renamer assigns unique identifiers to each variable.

The abstract representation of expressions follows the same design schema described above. An expression is indexed by the compilation stage, which influences the form of expressions and determines the representation of variables.

```
data Exp p where
  -- | variable
  Var :: VAR p -> Exp p
  -- | literal
  Lit :: Lit -> Exp p
  ...
  ElseGuard :: Exp Pr
  ...
  -- | abstraction
  Lam :: SrcLoc -> [Pat p] -> LamRHS p -> Exp p
  TyLam :: p :>=: Tc' ⇒ [TyVar] -> Exp p -> Exp p
  -- | application
  App :: Exp p -> Exp p -> Exp p
  TyApp :: p :>=: Tc' ⇒ Exp p -> [Tau p] -> Exp p
  ...
  -- | if /exp/ then /exp/ else /exp/
  Ite :: Tau p -> Prop p -> Exp p -> Exp p -> Exp p
  ...
  -- | type annotation
  TyAnn :: SrcLoc -> Exp p -> Sigma p -> Exp p
  ...
```

Note that type abstractions and applications can only be introduced during typechecking or in later stages of compilation. Moreover, the type of the type-application construct, which require arguments to be monotypes, states that our system is predicative.

Actually, GADTs have been used almost everywhere in the abstract syntax to encode invariants within the types of constructors, and to reflect the changes introduced by the flow of transformations during compilation. We have found the approach of parametrizing the abstract syntax with the compilation stage an interesting application

of GADTs and type-functions that, to our knowledge, it has not been used before. The use of this dependently-typed programming style is appealing because many properties became formally encoded into types constituting (automatically) checked documentation. Note that also the functions implementing the different steps of the pipeline get very precise types, so they cannot be chained incorrectly.

```
parseModule :: String -> ParseResult (Module Pr)
rnModule :: Module Pr -> RnM (Module Rn)
tcModule :: Module Rn -> TcM (Module Tc')
vcModule :: Module Tc' -> VcM (Module Tc,[VC])
dsModule :: Module Tc -> [VC] -> DsM Core.Module
```

However, in our experience, the *pros* and *cons* of this approach should be considered carefully. The use of these type-level features, especially type-functions, may introduce additional complexity. For instance, it tends to complicate the implementation of phase-polymorphic functions on the abstract syntax (e.g. variable substitution).

Chapter 7

Case Studies

This chapter presents a set of case studies aimed to analyze the practical use of $\mathcal{H}_{\text{SPEC}}$ to develop reliable software. In order to draw definitive conclusions it would be necessary to further develop the HSC compiler and use it to undertake an industrial-scale case study, but that goes far beyond the bounds of this project. However, we believe that our case studies are representative enough to get an insight about the potential of the proposed approach.

These case studies are introduced in the following sections, and the source code of all them is given in Appendix A. For each one, we extract a few metrics, such as the number of VCs and the percentage of testable ones, that try to reflect the burden of specification and proof associated with $\mathcal{H}_{\text{SPEC}}$. We also estimate the percentage of VCs that would be *tractable* by a more sophisticated simplifier, so the reader can make an idea of the potential of future versions of HSC. Besides checking that our implementations are correct, we also deliberately introduce errors into the code and use the VC checker to find them. As a result, we point out strengths and weakness of our approach.

7.1 Prelude

The *Prelude* case study consists of many standard definitions used in functional languages. It is intended to be the counterpart of the Haskell's *Prelude* module. In Haskell, *Prelude* is a distinguished module that is imported into any other module by default. In languages of the ML-family this module is usually called *Pervasives*.

This case study provides some simple but illustrative examples of the *contract inference* capabilities of $\mathcal{H}_{\text{SPEC}}$'s type system. Consider the function `head`, that extracts the first element of a non-empty list:

```
head (x::_) = x
```

In Haskell `head` receives the type `forall a. [a] -> a`. It is a partial function and will fail when passed an empty list: `*** Exception: Prelude.head: empty list`, this is (sadly) a well-known exception for any Haskell programmer. In contrast, since `head` is defined by a single equation, the $\mathcal{H}_{\text{SPEC}}$ type system infers a more precise type for it resorting to pattern types: `forall a. {(x::xs):[a]} -> a`. Thus, `head` is no longer a partial function —it only applies to non-empty lists, and an expression like `head []` is ill-typed and rejected by the typechecker.

Another interesting example is `map`, that applies a given function to each element of a list. The type of `map` is `forall a b. (a -> b) -> [a] -> [b]`, and it can be informally defined as `map f [`x_1`, `x_2`, ..., `x_n`] == [f `x_1`, f `x_2`, ..., f `x_n`]`. Intuitively, `map head` applies to a list of non-empty lists, and this intuition is effectively captured by the type inferred by $\mathcal{H}_{\text{SPEC}}$: `forall a. [{(x::xs):[a]}] -> [a]`. Notice that contract types are just like any other type, they can be used to instantiate polymorphic functions, and type inference works as one would expect.

7.1.1 Bug finding

We actually found a couple of bugs while coding this case study. The function `take`

```
take : forall a. {n:Nat}
                 -> {xs:[a]|length xs >= n}
                 -> {ys:[a]|length ys == n}
```

is used to extract a prefix from a list; more precisely, `take n xs` returns the prefix of `xs` of length `n`. We make use of dependent function types to guarantee —through typechecking— that `xs` has at least `n` elements, and that the result of `take n xs` is exactly of length `n`. Our first tentative definition was:

```
take _ []       =  []
take n (x::xs) =  x :: take (n-1) xs
```

The first equation may be shocking because it says that `[]` is the `n`-prefix of `[]`, which implies that `length [] == n`, independently of `n`. But this equation is fine since, given

`xs` $\mapsto$ `[]`, the only value of `n` satisfying the precondition `length [] >= n` is `0`. The problem is in the second equation, where `n` is wrongly assumed to be positive when the given list is non-empty. Although the VC generated is a little involved to be solved by our simplifier, the VC tester quickly found that the precondition of `take` is violated in the second equation by the expression `n-1` when `n` $\mapsto$ `0`.

7.1.2 Metrics

HSC generated a total of 28 VCs: 18 coercions and 2 completeness VCs, plus one additional VC per each one of the 8 declared lemmas. Most of the VCs can be tested using the QuickCheck back-end, excluding those that involve quantification over functions. A total of 16 out of 18 coercion VCs are testable, the other two came from the `filter` (higher-order) function:

```
filter : forall a. {p:a -> Bool} -> [a] -> {xs:[a]|all p xs}
filter p []              = []
filter p (x::xs) | p x   = x :: filter p xs
                 | else = filter p xs
```

For instance, the typechecker generates a coercion VC to guarantee that `[]` has type {`xs:[a]|all p xs`}, as required by the first equation of `filter`. The CORE formula to be proven is

```
forall (p:a -> Bool), all @a p ([] @a)
```

which is not supported by our tester because it requires generating arbitrary functions.

Approximately half of the VCs generated due to typechecking (excluding those originated from lemmas declared by the programmer) could be filtered out by a (slightly) more sophisticated simplifier. For example, the above VC can be shown true applying the defining equation `all p [] = True`. This data is summarized in the metrics shown in Table 7.1.

Functions#	**LOC**	**VCs (coercion/total)**	**Testable VCs**	**Tractable VCs**
35	120	18/28	23 (82%)	9 (32%)

Table 7.1: *Prelude* metrics.

7.2 Quicksort

This case study implements the well-known *Quicksort* sorting algorithm developed by Tony Hoare. Quicksort is probably one of the most popular examples to test the effectiveness of approaches to software correctness. The implementation of Quicksort used for this purpose is also more or less standard. First, one introduces a predicate for determining whether two lists are a permutation of each other, and another to determine if a list is sorted. Using these functions we can define a type that captures the precise definition of sorting algorithm:

```
type SortingAlgorithm =
        {p:[Int]} -> {q:[Int] | permutation p q && sorted q}
```

Then an auxiliary `split` function is usually defined, which splits a list into elements lower than *pivot* and the rest.

```
split : {p:Int} -> {xs:[Int]}
                -> (ls:[Int]|all (<p) ls,rs:[a]|all (>=p) rs)
```

However, we decided to replace `split` with the more general `partition` function, which splits a list based upon some given predicate. This is a common function in Haskell, defined in the *Data.List* (base) module, so it is more natural to use it instead of defining an ad-hoc split function. Anyway, it is easy to define `split` in terms of `partition` as `split p xs = partition (<p) xs`.

```
partition : forall a. {p:a -> Bool}
                      -> [a]
                      -> (ys:[a]|all p ys,ns:[a]|all (not p) ns)
```

Finally, it follows the representative implementation of Quicksort. For the base case, an empty list is already sorted. For a non-empty list, we pick the first element of the list as the *pivot*, and the rest of the elements are partitioned into those less than the pivot and those that are not. The final solution is computed by sorting each of these sublists separately, and then concatenating the two sorted sublists together.

```
qsort : SortingAlgorithm
qsort [] = []
qsort (x::xs) = append (qsort ls) (x :: qsort rs)
  where (ls,rs) = partition (<x) xs
```

7.2.1 Bug finding

We have intentionally introduced several bugs into the definition of `qsort`, and in every case the VC tester was able to find a counterexample that pointed out the existence of a bug. For instance, in the second equation of `qsort`, we could forget to include the pivot `x` as part of the solution:

```
qsort (x::xs) = append (qsort ls) (qsort rs)
  where (ls,rs) = partition (<x) xs
```

Such a minor mistake has a great impact, it converts `qsort` into a constant function that always returns `[]`. This causes the violation of `qsort`'s postcondition, since the output —even when sorted— is no longer a permutation of the input. The satisfaction of the postcondition is translated into two coercion VCs, one requiring the right hand side of the equation to be of type {`q:[Int] | permutation (x::xs) q && sorted q`}. This coercion requires the following formula to be proven:

```
forall (x:Int) (xs:[Int]),
    case partition (\x1 -> x1 < x) xs of
        (ls,rs) -> let q = append (qsort ls) (qsort rs)
                   in permutation (x::xs) q && sorted q
```

The VC tester is able to find a minimal counterexample. For `x` $\mapsto$ `0` and `xs` $\mapsto$ `[]`, since we know that `q` is always `[]`, `sorted q` holds but `permutation [0] q` does not.

7.2.2 Metrics

Typechecking generates 5 coercion VCs, 3 originated from `partition` and 2 from `qsort`. The three VCs associated with `partition` cannot be tested because they quantify over functions —but they are being instrumented by other tests. Nonetheless, the two VCs that entail the correctness of our Quicksort implementation are effectively handled by our VC tester. In contrast, the correctness of the very same `qsort` function in Coq requires a 30-line proof, which is larger than the code itself. Moreover, 4 out of the 5 generated VCs belong to tractable classes of formulas. This data is summarized in the metrics shown in Table 7.2.

Functions#	LOC	VCs (coercion/total)	Testable VCs	Tractable VCs
4	25	5/5	2 (40%)	4 (80%)

Table 7.2: *Quicksort* metrics.

7.3 Lambda calculus

We develop a partial formalization of the simply-typed lambda calculus in $\mathcal{H}_{\text{SPEC}}$. This example is intended to show the potential of our approach in compiler construction. The abstract syntax is exactly the one described in § 2.1, where the set of base types contains a single type `A`:

```
data Var = V Nat

data Type = A
          | Fun Type Type

data Term = Var Var
          | Lam Var Type Term
          | App Term Term
```

Variables are represented by natural numbers. For convenience, we will write variables and variable occurrences as $v0, v1, \ldots$ when providing examples of lambda terms.

We define a predicate `isNF` to mean whether a term is in normal form, and a type synonym `NF` for denoting the universe of terms in normal form.

```
type NF = {t:Term | isNF t}
```

This `NF` type is used to annotate a reduction function `red` that evaluates a term to normal form.

```
red : Term -> NF
```

Next, we implement a type inference algorithm for this lambda calculus. The `typeOf` function returns the type of a term if it is well-typed, or "fails" —returning `IllTyped`— otherwise. For simplicity, every free variable is assumed to have type `A`.

```
data TypeOf = Ok Type
            | IllTyped

typeOf : Term -> TypeOf
```

It is also useful to define a type denoting the universe of well-typed terms:

```
type WellTyped = {t:Term|typeOf t /= IllTyped}
```

Finally, we declare a lemma that states the *subject reduction* property. In type systems, this property means that evaluation preserves the type of a term.

```
lemma subject_reduction =
        forall (t:WellTyped), typeOf t == typeOf (red t)
```

7.3.1 Bug finding

We have introduced a couple of errors in the reduction function and used the VC tester to find them successfully. For instance, we changed the right hand side of the equation

```
red (Lam x a t) = Lam x a (red t)
```

from `Lam x a (red t)` to `red t`. This modification does not violate the specification of `red` —`red t` has type `NF`—, but breaks subject reduction. The tester finds the following counterexample: `t` $\mapsto$ `(Lam` $v0$ `A` $v1$`)`. The type of such `t` is `Fun A A` but the buggy `red` evaluates it to $v1$, which has type `A`.

This case study has exposed a weakness of the testing back-end. Even when a bug is found by the tester, sometimes it is not clear where it comes from. In particular, errors introduced in some equation of the reduction function may also be detected while testing VCs associated with other equations.

7.3.2 Metrics

The function `red` leads to 4 coercion VCs to check that it truly reduces terms to normal form. All these VCs are tractable and can be solved automatically during compilation using a more sophisticated simplifier. Another VC is due to the lemma stating the subject reduction property of our lambda calculus implementation. Note that proving subject reduction usually requires heavy inductive reasoning, hence being a non-trivial task that is unlikely be accomplish using automated provers. Nevertheless, here we managed to test the validity of subject reduction automatically, and the usefulness of the VC tester to find errors that break this property became manifest. Metrics collected for this case study are shown in Table 7.3.

Functions#	LOC	VCs (coercion/total)	Testable VCs	Tractable VCs
6	56	4/5	5 (100%)	4 (80%)

Table 7.3: *Lambda calculus* metrics.

7.4 ListSet

This example provides an implementation of sets on top of lists. Despite one usually picks implementations based on tree and hash maps, sets based on lists are useful for several scenarios. Many programming languages include this data structure in their standard library.

A set is defined as a list without duplicates. Given the *Prelude*'s function `nub`, which removes duplicates from a list, we specify that a list `l` has no duplicates iff `l == nub l`.

```
data Set a = Set {l:[a]|nub l == l}
```

A few standard, and mostly straightforward, set functions are defined. These functions have plain Damas-Milner types.

```
singleton : forall a. a -> Set a
member : forall a. a -> Set a -> Bool
isSubsetOf : forall a. Set a -> Set a -> Bool
insert : forall a. a -> Set a -> Set a
remove : forall a. a -> Set a -> Set a
```

Our intuition is that, in this case, it is more practical to describe the behavior of the operations through lemmas that establish relations between them:

```
lemma subset_member =
  forall s t x, isSubsetOf s t && member x s ==> member x t
lemma insert_member = forall x s, member x (insert x s)
lemma remove_member = forall x s, ~(member x (remove x s))
```

Otherwise, precise rich types would involve quantified formulas, or else types become large and cumbersome.

Nevertheless, we define the union operator for sets, trying to specify it accurately. For this purpose we resort to the first-order encoding of set operations.

```
union : forall a. {s:Set a}
            -> {t:Set a}
            -> {u:Set a|forall x, member x s || member x t <=> member x u}
```

It is anyway necessary to declare a few lemmas to state commutativity and associativity of this operator. Note that we have to define an equality function `eq` for sets that take into account that sets are unordered collections of elements —since the built-in equality `==` is too strong.

```
lemma union_id = forall s, union s empty == s
lemma union_comm = forall s t, eq (union s t) (union t s)
lemma union_assoc =
  forall s t u, eq (union s (union t u)) (union (union s t) u)
```

7.4.1 Bug finding

This case study has revealed another weakness of our testing back-end, related with the automatic generation of test cases. Most of the VCs involve quantification over sets, and generation of set values turn out to be a slow process. Since the `Set` type is just a wrapper over {`l:[a]|nub l == l`}, the VC tester proceeds by generating arbitrary lists and filtering out those that do not satisfy the type predicate `nub l == l`. As the length of the lists increases, it becomes less probable to get a list without duplicates, so the rate of test cases slows down progressively. This problem could be solved by allowing the programmer to provide hints to the tester, in order to guide the generation of test data.

7.4.2 Metrics

A total of 11 coercion VCs are generated, 8 of them serve to check that the invariant of the `Set` type is preserved by the operations, the other 3 are originated from the postcondition of the `union` function. Most of these VCs are particularly hard, even for state-of-the-art automated theorem provers, and thus the importance of the VC tester. This data is summarized in the metrics shown in Table 7.4.

Functions#	**LOC**	**VCs (coercion/total)**	**Testable VCs**	**Tractable VCs**
9	39	11/17	17 (100%)	4 (25%)

Table 7.4: *ListSet* metrics.

7.5 Summary

In our view, the above case studies make evident the usefulness of $\mathcal{H}_{\text{SPEC}}$ for developing more reliable software. The burden of specification is not so high compared to the use of any testing framework, and the cost associated with basic contract specification —mainly to avoid runtime crashes— is low. What is more important is that $\mathcal{H}_{\text{SPEC}}$ allows for incremental specification and verification.

Since typed functional programming tends to be less prone to runtime errors, giving slightly enriched type signatures for top-level definitions is often enough to guarantee runtime safety (and more). We do not count these type signatures as part of the specification burden, since annotating top-level definitions became a common practice in functional programming[1].

The burden of verification is remarkably low, due to our choice of automated methods for VC checking. The percentage of testable VCs is in average of 80%, and adding support for random generation of functions would increase this percentage to almost 100%. Thus, the potential of the VC testing approach is enourmous, providing an automated and cost-effective way to verify that a program meets its specification.

[1]For instance, the GHC Haskell compiler offers a `-fwarn-missing-signatures` flag to warn about any top-level definition missing a type signature.

Chapter 8

Conclusion

There is no definitive approach to the problem of software reliability. Type systems based on simply-typed λ-calculus, which have freed computer programs from a wide class of errors, are the most successful and widely used verification technology nowadays. Yet, mainstream type systems are weak specification languages, because expressiveness is restricted in favor of decidability. Ideally, programming languages based on dependent type theory would eventually be practical for mainstream programming and supersede today's languages. In the author's opinion, however, the inherent complexity of dependent type theory, and the burden of proof associated with (pure) dependently-typed programming, are likely to prevent its adoption in industrial settings.

In this work, we have presented a programming language —$\mathcal{H}_{\text{SPEC}}$— with a rich type system that serves also as a specification language. The concepts of pre/postcondition and type invariant are built into $\mathcal{H}_{\text{SPEC}}$'s type system, and smoothly integrated with Damas-Milner polymorphism. As for programming, $\mathcal{H}_{\text{SPEC}}$ is not very different from mainstream functional languages such as Caml and Haskell. Regarding typechecking, we make a clear distinction between traditional (decidable) type checking and contract checking. Programs are statically checked to be safe modulo plain-old types, thus offering the same typing guarantees as Damas-Milner. Contract satisfaction is however translated as a set of verification conditions that are checked separately. We also proposed a cost-saving approach to contract verification, mostly based on automated random testing. Despite testing does not prove the absence of errors, it actually finds bugs and can provide enough confidence for mainstream purposes.

Our approach is not in conflict with others. On the contrary, we strongly believe that developing high-assurance software necessarily involves the use of different approaches. For instance, $\mathcal{H}_{\text{SPEC}}$ already supports the so called *phantom types*, and therefore it is possible to encode some limited properties purely at the type-level —without resorting to any form of contract type, as explained in § 3.3. This technique is very useful in many scenarios, where contracts are usually not appropriate. Nonetheless, the experience acquired while developing the HSC $\mathcal{H}_{\text{SPEC}}$ compiler suggests us that such type-level encodings should be used carefully. The (ab)use of the type system in this way tends to make programming painful.

8.1 Contributions

This work has both theoretical and pragmatic contributions. Most remarkably, we have extended a Damas-Milner type system to incorporate predicate types and limited forms of dependent types; and we have presented, to the best of our knowledge, the first type inference algorithm for such a system. Moreover, we have introduced a new syntactic form of predicate subtyping based on pattern-matching, that we coined as *pattern types*. This theoretical work has been developed and implemented into a Haskell-like functional programming language in chapters 4 through 6. The use of this language for undertaking some representative programming problems, explored in chapter 7, has been demonstrated promising.

Neat extension of Damas-Milner

We managed to extend Damas-Milner with predicate subtypes while preserving most of the simplicity of its original presentation. This can be appreciated by comparing the declarative description of both type systems, Damas-Milner's introduced in § 2.3.2 and $\mathcal{H}_{\text{SPEC}}$'s in § 5.3.

Simple type inference

Our type inference method is intended to be conceptually simple, hence comprehensible and easily predictable for the programmer. This inherent simplicity is also reflected in the algorithm itself, that can be formulated as a slight adaptation of Algorithm $\mathcal{M}$

—a popular variation of the original Algorithm $\mathcal{W}$ by Damas and Milner. The main difference of our type inference algorithm, in comparison to the standard Algorithm $\mathcal{M}$, resides in a modified unification algorithm that (carefully) handles the propagation of type predicates. In contrast, other algorithms for inferring refinement predicates rely on constraint solving [11]. Indeed, by using constraint solvers it is possible to infer very precise type-predicates but, in our view, it makes inference a more obscure process. The programmer should understand the basis of type inference in order to make sense of the results and errors produced by the typechecker.

Pattern types

We introduce a new syntactic form of predicate subtyping based on pattern matching. A type like `{(x::xs):[a]}`, denoting non-empty lists of type `a`, can be desugared to `{l:[a]|case l of {[] -> False; x::xs -> True}}`. Pattern types are very useful for many purposes such as: writing some predicate types more succinctly, as in `{(x::xs):[a]|x>maximum xs}`; reducing the number of verification conditions, since coercions between pattern types can often be statically decided; and driving test data generation, by taking advantage of knowing the shape of the data that has to be generated.

8.2 Future work

There are two major directions for future work to pursue: the extension of the $\mathcal{H}_{\text{SPEC}}$ system with other desirable features present in most programming languages, like a module system; and the improvement of the contract checking apparatus, either to reduce the number of verification conditions or to provide other means to solve them.

We are especially interested in extending our work to develop a variant of Haskell with contract types, as they were introduced in $\mathcal{H}_{\text{SPEC}}$. The use of Haskell in industry is relatively new, but many companies are adopting it for settings where reliability is very important, from aerospace and defense, to finances. For this reason we expect that a Haskell dialect for high-assurance programming will be very welcomed.

Integration with other Damas-Milner extensions

The main challenge to develop such a Haskell dialect would be to integrate the many extensions to Damas-Milner present in Haskell into our system. The inclusion of many of these extensions would not be specially problematic. Namely the Haskell's module and record system, mutually recursive definitions, and arbitrary rank polymorphism [23]. The most challenging feature to accommodate is probably the Haskell's *type class* mechanism [56], which provides an uniform solution to function overloading —e.g. the use of + to refer to both integer and floating-point addition.

A smart type-driven simplifier

The analysis of some representative examples in chapter 7 has shown that the number of verification conditions generated by HSC could be reduced from 25 to 80 percent. It should be investigated a smarter algorithm for logic simplification, that would make good use of the logical facts included within types. This algorithm is targeted for simplifying (and solving) verification conditions during compilation, so there is an important tradeoff between effectiveness and efficiency.

Checking of subtype coercions

It is imperative to improve and grow the toolset available for verification. Since we focused on a test-based approach, one of our priorities would be to overcome the weakness of this back-end —already stated in chapter 7. That is, to add support for random generation of functions, and for user-defined data generators. The generation of random functions has already been studied and is usually addressed through memoization [27].

In order to meet higher assurance levels, it becomes necessary to tackle the integration of formal verification techniques into our checking apparatus. Despite our SMT back-end is only a proof-of-concept, the use of SMT solvers for software verification is not new, and there exist many interesting encodings that should be explored [57]. We are particularly interested in the use of *extended static checking* techniques for verifying subtype coercions [14]. These techniques trade precision for automation, so they can prove a coercion valid or *probably* valid, as they can find errors or *potential* errors.

Appendix A

Case Studies

This appendix gives the source code of the case studies discussed in chapter 7.

A.1 Prelude

```
id : forall a. a -> a
id x = x

const : forall a b. a -> b -> a
const x _ =  x

flip : forall a b c. (a -> b -> c) -> b -> a -> c
flip f x y =  f y x

max : {a:Int} -> {b:Int} -> {m:Int|(m == a || m == b) && m>=a && m>=b}
max a b | a > b = a
        | else  = b

min : {a:Int} -> {b:Int} -> {m:Int|(m == a || m == b) && m<=a && m<=b}
min a b | a < b = a
        | else  = b

head : forall a. {(_::_):[a]} -> a
head (x::_) = x

tail : forall a. {(_::_):[a]} -> [a]
tail (_::xs) = xs
```

```
last : forall a. {(_::_):[a]} -> a
last (x::xs) = case xs of
                    []    -> x
                    _::_  -> last xs

null : forall a. [a] -> Bool
null []     =  True
null (_::_) =  False

length : forall a. [a] -> Nat
length [] = 0
length (_::xs) = 1 + length xs

elem : forall a. a -> [a] -> Bool
elem a [] = False
elem a (x::xs)
  | a == x = True
  | else   = elem a xs

delete1 : forall a. a -> [a] -> [a]
delete1 a [] = []
delete1 a (x::xs)
  | a == x = xs
  | else   = x :: delete1 a xs

delete : forall a. a -> [a] -> [a]
delete a [] = []
delete a (x::xs)
  | a == x = delete a xs
  | else   = x :: delete a xs

lemma delete_elem = forall x xs, ~(elem x (delete x xs))

append : forall a. [a] -> [a] -> [a]
append []      ys = ys
append (x::xs) ys = x :: (append xs ys)

lemma append_length =
```

```
  forall xs ys, length (append xs ys) == length xs + length ys

map : forall a b. (a -> b) -> [a] -> [b]
map f [] = []
map f (x::xs) = f x :: map f xs

lemma map_id = forall f xs, map f (map f xs) == map f xs
lemma map_length = forall f xs, length (map f xs) == length xs

foldr : forall a b. (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z
foldr f z (x::xs) = f x (foldr f z xs)

lemma foldr_id = foldr (::) [] == id

foldr1 : forall a. (a -> a -> a) -> {(_::_):[a]} -> a
foldr1 f (x::xs) = f x (foldr f x xs)

foldl : forall a b. (a -> b -> a) -> a -> [b] -> a
foldl f z []      = z
foldl f z (x::xs) = foldl f (f z x) xs

foldl1 : forall a. (a -> a -> a) -> {(_::_):[a]} -> a
foldl1 f (x::xs) = foldl f x xs

reverse : forall a. [a] -> [a]
reverse = foldl (flip (::)) []

lemma reverse_length = forall xs, length (reverse xs) == length xs

all : forall a. (a -> Bool) -> [a] -> Bool
all p [] = True
all p (x::xs)
  | p x  = all p xs
  | else = False

any : forall a. (a -> Bool) -> [a] -> Bool
any p [] = False
any p (x::xs)
```

```
  | p x  = True
  | else = any p xs

filter : forall a. {p:a -> Bool} -> [a] -> {xs:[a]|all p xs}
filter p []             = []
filter p (x::xs) | p x  = x :: filter p xs
                 | else = filter p xs

nub : forall a. [a] -> [a]
nub []      = []
nub (x::xs) = x :: nub (delete x xs)

lemma nub_idempotent = forall xs, nub (nub xs) == nub xs
lemma nub_elem = forall x xs, ~(elem x xs) ==> nub(x::xs) == x::nub xs

concat : forall a. [[a]] -> [a]
concat xss = foldr append [] xss

take : forall a. {n:Nat}
                -> {xs:[a]|length xs >= n}
                -> {ys:[a]|length ys == n}
take n xs | n == 0 = []
          | else   = case xs of
                          x::xs1 -> x :: take (n-1) xs1


drop : forall a. {n:Nat}
                -> {xs:[a]|length xs >= n}
                -> {ys:[a]|length ys == length xs - n}
drop n xs | n == 0 = xs
          | else   = case xs of
                          _::xs1 -> drop (n-1) xs1

sum : [Int] -> Int
sum = foldr (+) 0

product : [Int] -> Int
product = foldr (*) 1
```

```
maximum : {s@(_::_):[Int]} -> {a:Int|forall b, elem b s ==> a >= b}
maximum xs =  foldl1 max xs

minimum : {s@(_::_):[Int]} -> {a:Int|forall b, elem b s ==> a <= b}
minimum xs =  foldl1 min xs

and : {bs:[Bool]} -> {f:Bool|f <==> all (==True) bs}
and bs = foldr (&&) True bs

or : {bs:[Bool]} -> {f:Bool|f <==> any (==True) bs}
or bs = foldr (||) False bs

data Maybe a = Nothing | Just a

maybe : forall a b. b -> (a -> b) -> Maybe a -> b
maybe  b _f Nothing  = b
maybe _b  f (Just a) = f a

lookup : forall a b. a -> [(a,b)] -> Maybe b
lookup x [] = Nothing
lookup x ((a,b)::ps)
  | x == a = Just b
  | else   = lookup x ps
```

A.2 Quicksort

```
permutation : forall a. [a] -> [a] -> Bool
permutation []      ys = null ys
permutation (x::xs) ys
  | elem x ys = permutation xs (delete1 x ys)
  | else      = False

sorted : [Int] -> Bool
sorted [] = True
sorted [x] = True
sorted (a::b::xs)
  | a <= b = sorted xs
  | else   = False
```

```
type SortingAlgorithm =
        {p:[Int]} -> {q:[Int] | permutation p q && sorted q}

partition : forall a. {p:a -> Bool}
                      -> [a]
                      -> (ys:[a]|all p ys,ns:[a]|all (not p) ns)
partition p [] = ([],[])
partition p (x::xs)
  | p x  = (x::ys,ns)
  | else = (ys,x::ns)
  where (ys,ns) = partition p xs

qsort : SortingAlgorithm
qsort [] = []
qsort (x::xs) = append (qsort ls) (x :: qsort rs)
  where (ls,rs) = partition (<x) xs
```

A.3 Lambda calculus

```
data Var = V Nat

data Type = A
          | Fun Type Type

data Term = Var Var
          | Lam Var Type Term
          | App Term Term

fv : Term -> [Var]
fv (Var x)     = [x]
fv (Lam x _ t) = delete x (fv t)
fv (App f t)   = append (fv f) (fv t)

subst : Var -> Term -> Term -> Term
subst x e t@(Var y)
  | x == y = e
  | else   = t
subst x e t@(Lam y a u)
  | x == y = t
```

```
  | else   = Lam y a (subst x e u)
subst x e (App f t) = App (subst x e f) (subst x e t)

isNF : Term -> Bool
isNF (Var _)             = True
isNF (Lam x _ t)         = isNF t
isNF (App (Var _) t)     = isNF t
isNF (App (Lam _ _ _) t) = False
isNF (App f@(App _ _) t) = isNF f && isNF t

type NF = {t:Term | isNF t}

red : Term -> NF
red (Var x)     = Var x
red (Lam x a t) = Lam x a (red t)
red (App f t)   = case f' of
                       Var _     -> App f' (red t)
                       Lam x _ u -> red (subst x t u)
                       App _ _   -> App f' (red t)
  where f' = red f

data TypeOf = Ok Type
            | IllTyped

type Env = [(Var,Type)]

typeOf : Term -> TypeOf
typeOf t = go [] t
  where go env (Var x)
          = case lookup x env of
                 Just a  -> Ok a
                 Nothing -> Ok A
        go env (Lam x a t)
          = case go ((x,a)::env) t of
                 Ok b     -> Ok (Fun a b)
                 IllTyped -> IllTyped
        go env (App f t)
          = case go env f of
                 Ok A          -> IllTyped
```

```
                Ok (Fun a b) ->
                   case go env t of
                       Ok a' | a == a' -> Ok b
                             | else    -> IllTyped
                       IllTyped        -> IllTyped
                IllTyped     -> IllTyped

welltyped : Term -> Bool
welltyped t = typeOf t /= IllTyped

type WellTyped = {t:Term|welltyped t}

lemma subject_reduction =
        forall (t:WellTyped), typeOf t == typeOf (red t)
```

A.4 ListSet

```
data Set a = Set {l:[a]|nub l == l}

eq : forall a. Set a -> Set a -> Bool
eq (Set []) (Set ys) = null ys
eq (Set (x::xs)) (Set ys)
  | elem x ys = eq (Set xs) (Set (delete x ys))
  | else      = False

empty : forall a. Set a
empty = Set []

singleton : forall a. a -> Set a
singleton x = Set [x]

set : forall a. [a] -> Set a
set xs = Set (nub xs)

member : forall a. a -> Set a -> Bool
member a (Set xs) = elem a xs

isSubsetOf : forall a. Set a -> Set a -> Bool
isSubsetOf (Set xs) (Set ys) = all (flip elem ys) xs
```

```
lemma subset_member =
  forall s t x, isSubsetOf s t && member x s ==> member x t

insert : forall a. a -> Set a -> Set a
insert a s@(Set xs)
  | elem a xs = s
  | else      = Set (a::xs)

lemma insert_member = forall x s, member x (insert x s)

remove : forall a. a -> Set a -> Set a
remove a (Set xs) = Set (delete a xs)

lemma remove_member = forall x s, ~(member x (remove x s))

union : forall a. {s:Set a}
            -> {t:Set a}
            -> {u:Set a|forall x, member x s || member x t <==> member x u}
union s@(Set xs) t@(Set ys)
  = case xs of
       []   -> t
       _::_ -> case ys of
                    []   -> s
                    _::_ -> Set (nub (append xs ys))

lemma union_id = forall s, union s empty == s
lemma union_comm = forall s t, eq (union s t) (union t s)
lemma union_assoc =
  forall s t u, eq (union s (union t u)) (union (union s t) u)
```

Appendix B

Installation of HSC

1. Install the Haskell Platform version 2011.2.0.1 for your operating system. Download it from `http://hackage.haskell.org/platform/` and follow the installation instructions. The supported platforms are Windows, Mac and Linux.

2. Install the Yices SMT solver. Download it from `http://yices.csl.sri.com/download.shtml` and follow the installation instructions. Then add the *yices* binary to your program path.

   - **Note:** This step is optional, it is only required if you are going to use the SMT back-end.

3. Download the source distribution of HSC from `http://files.iagoabal.eu/UdC-EI-PFC/hsc.tar.gz`. Extract the `hsc.tar.gz` file somewhere, and finally enter the `hsc-*` directory and type: `cabal install`.

4. The above step installs the *hsc* binary in `$HOME/.cabal/bin`. You should add `$HOME/.cabal/bin` to your program path.

   - **Note:** To uninstall HSC just remove `$HOME/.cabal/bin/hsc`, or delete the directory `$HOME/.cabal`.

5. Type `hsc --help` for getting usage information. You can try HSC with the modules included in the `examples` directory.

Appendix C

Planning

This work was planned to be developed from May 2010 to September 2012, while doing a 2-year Master's degree in formal methods at Universidade do Minho, in Portugal. Because of the Master's workload the working days and times have vary substantially from quarter to quarter. During the summer quarter of each year the work was scheduled for 40 hours per week, but during the other three quarters the number of working days was reduced considerably. According to this planning, the whole project would be $1,728$ hours of work. Assuming a salary of €10/hour for a last-year engineering student, the total cost of the project would be of €$17,280$.

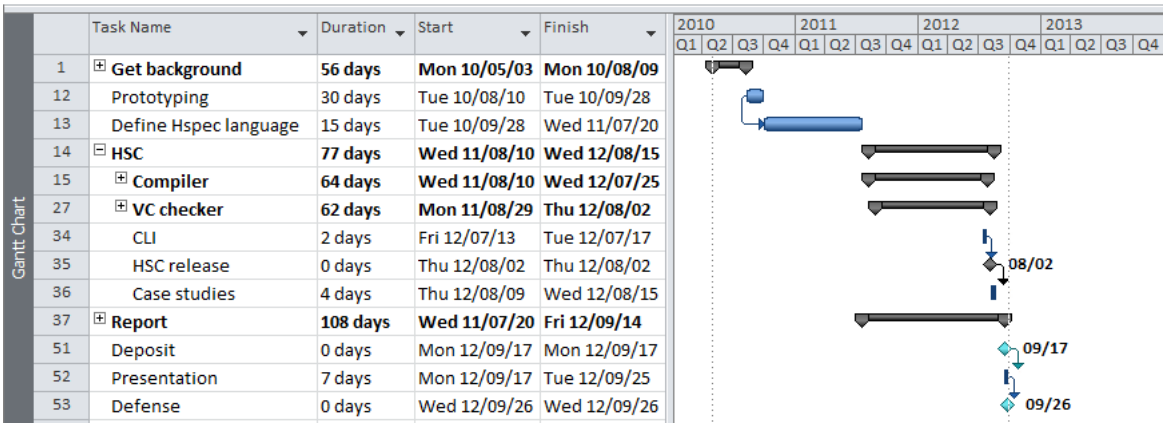


Figure C.1: Plan overview

The overview plan of the project is shown in Figure C.1. The work consists of six main tasks that can be logically divided into four phases:

1. *Background study of correct software development, type theory, and compiler tech-*

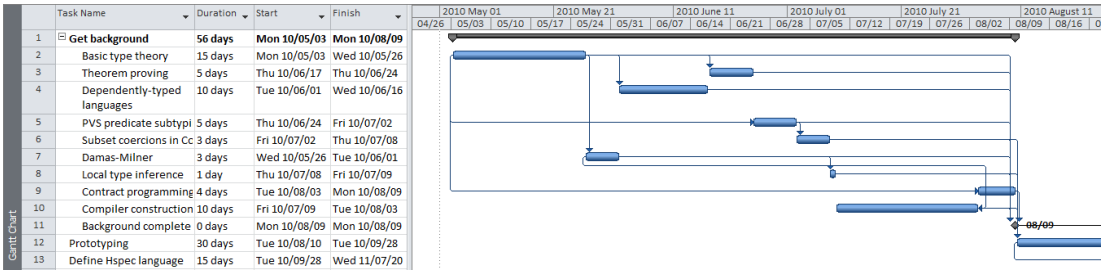
nology. Due to the inherent complexity of the field of study (especially type theory), and the research nature of the project, this phase (task) lasted for more than 2 months.

2. *Theoretical development of the $\mathcal{H}_{\text{SPEC}}$ system.* This phase includes the construction of a prototype implementing a naive lambda calculus extended with predicate subtypes. Then we described the $\mathcal{H}_{\text{SPEC}}$ language and formalized its type system. The prototype helped us to get a better understanding of the type inference process in presence of predicate subtypes, before formalizing the type system as an extension of Damas-Milner.

3. *Proof-of-concept implementation of $\mathcal{H}_{\text{SPEC}}$.* This phase comprises the implementation of a reference typechecker for $\mathcal{H}_{\text{SPEC}}$, and a prototype VC checker. Both functions are interrelated and were packaged into a single command-line tool, called HSC ($\mathcal{H}_{\text{SPEC}}$ compiler). The experience acquired developing the prototype of the previous phase turned out to be of great value for the implementation of the $\mathcal{H}_{\text{SPEC}}$ typechecker.

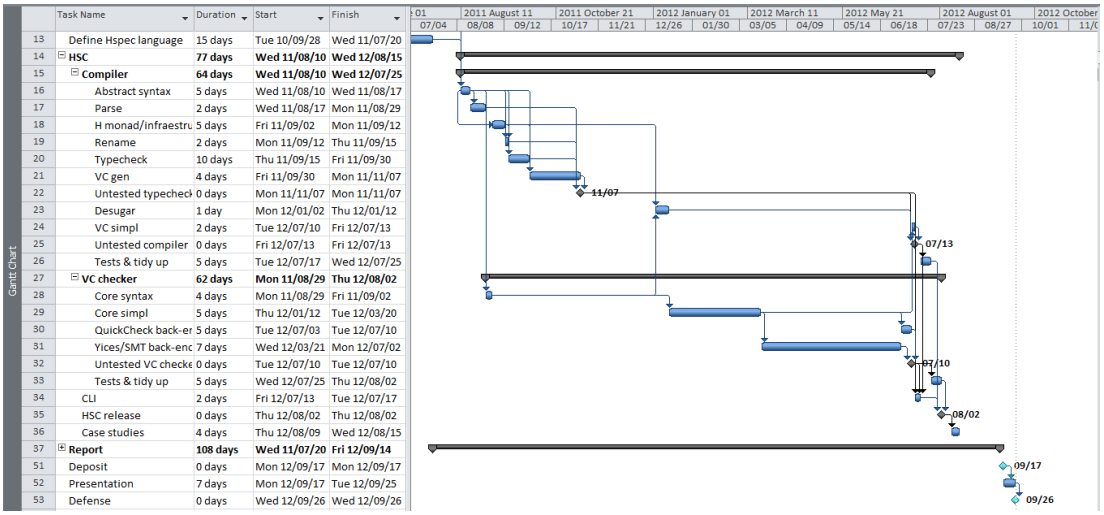
4. *Conclusion of the project.* In order to conclude the project, HSC was tested against some case studies, looking for evidence that the approach proposed in this work is indeed of practical usefulness for developing correct software. The writing of the report, although it started right after the $\mathcal{H}_{\text{SPEC}}$ language was formally defined, also belongs to this phase. This final report constitutes a complete organization of our arguments and results.

A detailed view of the plan is shown in Figure C.2, where the main three summary tasks of the project are expanded and all subtasks displayed.



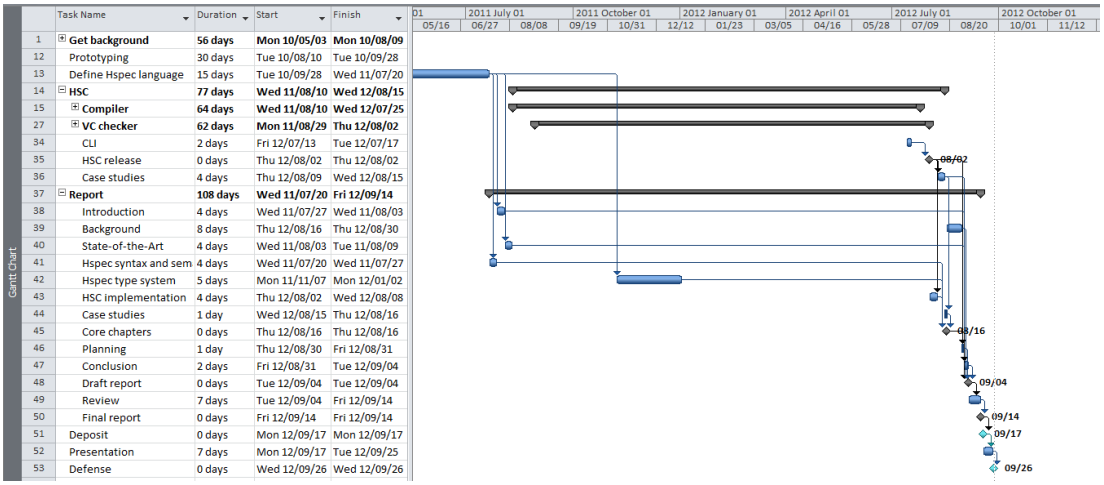
	Task Name	Duration	Start	Finish
1	**Get background**	**56 days**	**Mon 10/05/03**	**Mon 10/08/09**
2	Basic type theory	15 days	Mon 10/05/03	Wed 10/05/26
3	Theorem proving	5 days	Thu 10/06/17	Thu 10/06/24
4	Dependently-typed languages	10 days	Tue 10/06/01	Wed 10/06/16
5	PVS predicate subtypi	5 days	Thu 10/06/24	Fri 10/07/02
6	Subset coercions in Cc	3 days	Fri 10/07/02	Thu 10/07/08
7	Damas-Milner	3 days	Wed 10/05/26	Tue 10/06/01
8	Local type inference	1 day	Thu 10/07/08	Fri 10/07/09
9	Contract programming	4 days	Tue 10/08/03	Mon 10/08/09
10	Compiler construction	10 days	Fri 10/07/09	Tue 10/08/03
11	Background complete	0 days	Mon 10/08/09	Mon 10/08/09
12	Prototyping	30 days	Tue 10/08/10	Tue 10/09/28
13	Define Hspec language	15 days	Tue 10/09/28	Wed 11/07/20

(a) Background study



	Task Name	Duration	Start	Finish
13	Define Hspec language	15 days	Tue 10/09/28	Wed 11/07/20
14	**HSC**	**77 days**	**Wed 11/08/10**	**Wed 12/08/15**
15	**Compiler**	**64 days**	**Wed 11/08/10**	**Wed 12/07/25**
16	Abstract syntax	5 days	Wed 11/08/10	Wed 11/08/17
17	Parse	2 days	Wed 11/08/17	Mon 11/08/29
18	H monad/infraestru	5 days	Fri 11/09/02	Mon 11/09/12
19	Rename	2 days	Mon 11/09/12	Thu 11/09/15
20	Typecheck	10 days	Thu 11/09/15	Fri 11/09/30
21	VC gen	4 days	Fri 11/09/30	Mon 11/11/07
22	Untested typecheck	0 days	Mon 11/11/07	Mon 11/11/07
23	Desugar	1 day	Mon 12/01/02	Thu 12/01/12
24	VC simpl	2 days	Tue 12/07/10	Fri 12/07/13
25	Untested compiler	0 days	Fri 12/07/13	Fri 12/07/13
26	Tests & tidy up	5 days	Tue 12/07/17	Wed 12/07/25
27	**VC checker**	**62 days**	**Mon 11/08/29**	**Thu 12/08/02**
28	Core syntax	4 days	Mon 11/08/29	Fri 11/09/02
29	Core simpl	5 days	Thu 12/01/12	Tue 12/03/20
30	QuickCheck back-er	5 days	Tue 12/07/03	Tue 12/07/10
31	Yices/SMT back-enc	7 days	Wed 12/03/21	Mon 12/07/02
32	Untested VC checke	0 days	Tue 12/07/10	Tue 12/07/10
33	Tests & tidy up	5 days	Wed 12/07/25	Thu 12/08/02
34	CLI	2 days	Fri 12/07/13	Tue 12/07/17
35	HSC release	0 days	Thu 12/08/02	Thu 12/08/02
36	Case studies	4 days	Thu 12/08/09	Wed 12/08/15
37	**Report**	**108 days**	**Wed 11/07/20**	**Fri 12/09/14**
51	Deposit	0 days	Mon 12/09/17	Mon 12/09/17
52	Presentation	7 days	Mon 12/09/17	Tue 12/09/25
53	Defense	0 days	Wed 12/09/26	Wed 12/09/26

(b) HSC development



	Task Name	Duration	Start	Finish
1	**Get background**	**56 days**	**Mon 10/05/03**	**Mon 10/08/09**
12	Prototyping	30 days	Tue 10/08/10	Tue 10/09/28
13	Define Hspec language	15 days	Tue 10/09/28	Wed 11/07/20
14	**HSC**	**77 days**	**Wed 11/08/10**	**Wed 12/08/15**
15	**Compiler**	**64 days**	**Wed 11/08/10**	**Wed 12/07/25**
27	**VC checker**	**62 days**	**Mon 11/08/29**	**Thu 12/08/02**
34	CLI	2 days	Fri 12/07/13	Tue 12/07/17
35	HSC release	0 days	Thu 12/08/02	Thu 12/08/02
36	Case studies	4 days	Thu 12/08/09	Wed 12/08/15
37	**Report**	**108 days**	**Wed 11/07/20**	**Fri 12/09/14**
38	Introduction	4 days	Wed 11/07/27	Wed 11/08/03
39	Background	8 days	Thu 12/08/16	Thu 12/08/30
40	State-of-the-Art	4 days	Wed 11/08/03	Tue 11/08/09
41	Hspec syntax and sem	4 days	Wed 11/07/20	Wed 11/07/27
42	Hspec type system	5 days	Mon 11/11/07	Mon 12/01/02
43	HSC implementation	4 days	Thu 12/08/02	Wed 12/08/08
44	Case studies	1 day	Wed 12/08/15	Thu 12/08/16
45	Core chapters	0 days	Thu 12/08/16	Thu 12/08/16
46	Planning	1 day	Thu 12/08/30	Fri 12/08/31
47	Conclusion	2 days	Fri 12/08/31	Tue 12/09/04
48	Draft report	0 days	Tue 12/09/04	Tue 12/09/04
49	Review	7 days	Tue 12/09/04	Fri 12/09/14
50	Final report	0 days	Fri 12/09/14	Fri 12/09/14
51	Deposit	0 days	Mon 12/09/17	Mon 12/09/17
52	Presentation	7 days	Mon 12/09/17	Tue 12/09/25
53	Defense	0 days	Wed 12/09/26	Wed 12/09/26

(c) Report writing

Figure C.2: Detailed project planning.

Glossary

HSC is a proof-of-concept $\mathcal{H}_{\text{SPEC}}$ compiler. 84

Barendregt convention denotes terms where the names of bound variables are always chosen to be different from free ones, and where different binders (e.g. λ operators) bind different variables. 10, 15, 80

BNF (stands for Backus-Naur Form) is a notation for describing context-free grammars, often used to describe the syntax of programming languages. 36

Curry-style refers to the omision of type annotations in λ-abstractions. 11, 16

dependent function type is a generalization of the function space by allowing the range to depend on the domain. 35

desugaring is the process of removing syntactic sugar, translating *sugared* constructs into more fundamental language constructs. 82, 83, 88

EBNF (Extended BNF) is family of extensions to the BNF notation. The ISO has adopted an EBNF standard (ISO/IEC 14977). 5, 36

heavyweight formal methods are a class of (theoretically) rigorous formal methods which promote full formalization. The application of these methods require a considerable background in their theoretical foundations, and substantial expertise in the use of the tools that support them. 2

high-order function is a function that takes a function as input or returns a function as output. 105

iff is an abbreviation of *if and only if*, the logical biconditional or bi-implication. 44, 49, 110

ill-typed refers to a term for which there is no typing derivation, so it is not a valid term with respect to the type system. 38, 104

kernel is a term originated in the German word "kern", which means *core*. 35

lazy evaluation is an evaluation strategy that delays the evaluation of an expression until its value is needed. 90

LR grammar is a grammar that can be parsed by an LR parser. 80, 87

monotype stands for *monomorphic* type. 18, 22, 100

parametric polymorphism a kind of polymorphism that allows a function to be defined generically, so that it can handle values identically without depending on their type. This class of polymorphism is known as *generic programming* within the object-oriented community. 7, 14

PNF prenex normal form 17, 18

polytype stands for *polymorphic* type. 18

postcondition is a condition that must hold after the execution of a program unit (function, procedure, method, ...). A function postcondition establishes a relation between the input and the output values of a function. 2

precondition is a condition that must hold before the execution of a program unit (function, procedure, method, ...). A function precondition is a restriction of the function's domain. 2

predicate subtype is a term introduced by the PVS system to name subset types, when those are handled as subtypes. 35

predicativity (regarding type polymorphism) refers to the restriction of type variables to range over monomorphic types only. 40

principal type is the most general type of a term, such that any other type that could be assigned to the term is an instance of it. 18

programming by contract (or contract programming) is an approach to software develpment that prescribes the specification of programs with logical assertions, in the form of pre- and postcondition, and data type invariants. 35

redex is a reducible expression. 10

SMT solver is an automated theorem prover that deals with decidable fragments of first-order logic modulo background theories, like integer arithmetic or arrays. 90

TCC Type Correctness Condition 58, 62, 82

typable refers to a term or program that is valid according to a set of typing rules. 10, 16

type mismatch is an error reported by the typechecker when the inferred type for a term does not match the expected type for that term. 83

UML Unified Modeling Language 84

V&V verification and validation 1

VC Verification Condition 80, 82, 91, 92, 94, 103, 105, 107, 112

well-typed is the same as typable. 13, 109

Bibliography

[1] Tassey, G., *Economic Impacts of Inadequate Infrastructure for Software Testing.* Diane Pub Co, 2003.

[2] Grady, R. B. and Caswell, D. L., *Software metrics: establishing a company-wide program.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

[3] Grady, R. B., *Practical software metrics for project management and process improvement.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[4] Logozzo, F., "Our experience with the codecontracts static checker," in *Verified Software: Theories, Tools, Experiments* (Joshi, R., Müller, P., and Podelski, A., eds.), vol. 7152 of *Lecture Notes in Computer Science*, pp. 241–242, Springer Berlin / Heidelberg, 2012.

[5] Chalin, P., "Early detection of jml specification errors using esc/java2," in *Proceedings of the 2006 conference on Specification and verification of component-based systems*, SAVCBS '06, (New York, NY, USA), pp. 25–32, ACM, 2006.

[6] Filliâtre, J. C. and Marché, C., "The why/krakatoa/caduceus platform for deductive program verification," in *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, (Berlin, Heidelberg), pp. 173–177, Springer-Verlag, 2007.

[7] Barnes, J., *High integrity software: the Spark approach to safety and security.* Addison-Wesley, 2003.

[8] Fähndrich, M. and Logozzo, F., "Static contract checking with abstract interpretation," in *Proceedings of the 2010 international conference on Formal verification of*

object-oriented software, FoVeOOS'10, (Berlin, Heidelberg), pp. 10–30, Springer-Verlag, 2011.

[9] Findler, R. B. and Felleisen, M., "Contracts for higher-order functions," in *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, (New York, NY, USA), pp. 48–59, ACM, 2002.

[10] Flanagan, C., "Hybrid type checking," in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, (New York, NY, USA), pp. 245–256, ACM, 2006.

[11] Knowles, K. and Flanagan, C., "Type reconstruction for general refinement types," in *Proceedings of the 16th European conference on Programming*, ESOP'07, (Berlin, Heidelberg), pp. 505–519, Springer-Verlag, 2007.

[12] Régis-Gianas, Y. and Pottier, F., "A hoare logic for call-by-value functional programs," in *Proceedings of the 9th international conference on Mathematics of Program Construction*, MPC '08, (Berlin, Heidelberg), pp. 305–335, Springer-Verlag, 2008.

[13] Rondon, P. M., Kawaguci, M., and Jhala, R., "Liquid types," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, (New York, NY, USA), pp. 159–169, ACM, 2008.

[14] Xu, D. N., Peyton Jones, S., and Claessen, K., "Static contract checking for haskell," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, (New York, NY, USA), pp. 41–52, ACM, 2009.

[15] Owre, S., Rushby, J. M., and Shankar, N., "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)* (Kapur, D., ed.), vol. 607 of *Lecture Notes in Artificial Intelligence*, (Saratoga, NY), pp. 748–752, Springer-Verlag, June 1992.

[16] Rushby, J., Owre, S., and Shankar, N., "Subtypes for specifications: Predicate subtyping in PVS," *IEEE Transactions on Software Engineering*, vol. 24, no. 9, pp. 709–720, 1998.

[17] Sozeau, M., "Subset coercions in Coq," in *Types for Proofs and Programs* (Altenkirch, T. and McBride, C., eds.), vol. 4502 of *Lecture Notes in Computer Science*, pp. 237–252, Springer Berlin / Heidelberg, 2007.

[18] Damas, L. and Milner, R., "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, (New York, NY, USA), pp. 207–212, ACM, 1982.

[19] Church, A., "A set of postulates for the foundation of logic," *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366, 1932.

[20] Church, A., "A set of postulates for the foundation of logic (second paper)," *Annals of Mathematics*, vol. 34, no. 4, pp. 839–864, 1933.

[21] Pierce, B. C., *Types and programming languages*. Cambridge, MA, USA: MIT Press, 2002.

[22] Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. J., *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[23] Jones, S. P., Vytiniotis, D., Weirich, S., and Shields, M., "Practical type inference for arbitrary-rank types," *Journal of Functional Programming*, vol. 17, no. 01, p. 1, 2006.

[24] Baader, F. and Snyder, W., "Unification theory," in *Handbook of Automated Reasoning* (Robinson, J. A. and Voronkov, A., eds.), vol. 1, pp. 447–533, MIT Press, 2001.

[25] Wildmoser, M. and Nipkow, T., "Certifying machine code safety: Shallow versus deep embedding," in Slind *et al.* [63], pp. 305–320.

[26] Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. J., *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[27] Owre, S., "Random testing in PVS," in *In: Workshop on Automated Formal Methods (AFM*, 2006.

[28] Nipkow, T., Paulson, L. C., and Wenzel, M., *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.

[29] Wenzel, M., Paulson, L. C., and Nipkow, T., "The isabelle framework," in *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, (Berlin, Heidelberg), pp. 33–38, Springer-Verlag, 2008.

[30] Haftmann, F., "From higher-order logic to haskell: There and back again," *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 155–158, 2010.

[31] The Coq Development Team, *The Coq Proof Assistant: Reference Manual*. PPS laboratory, CNRS and Université Paris Diderot.

[32] Bertot, Y. and Castéran, P., *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, Springer Verlag, 2004.

[33] Howard, W. A., *The formulae-as-types notion of construction*, pp. 480–490. Academic Press, 1980.

[34] McKinna, J., "Why dependent types matter," in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, (New York, NY, USA), pp. 1–1, ACM, 2006.

[35] Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., and Birkedal, L., "Ynot: Reasoning with the awkward squad," in *In ACM SIGPLAN International Conference on Functional Programming*, 2008.

[36] Norell, U., *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[37] Norell, U., "Dependently typed programming in agda," *Proceedings of the 4th international workshop on Types in language design and implementation TLDI 09*, 2008.

[38] Stump, A., Deters, M., Petcher, A., Schiller, T., and Simpson, T., "Verified programming in guru," in *Proceedings of the 3rd workshop on Programming languages meets program verification*, PLPV '09, (New York, NY, USA), pp. 49–58, ACM, 2008.

[39] Xi, H., "Dependent ml an approach to practical programming with dependent types," *J. Funct. Program.*, vol. 17, pp. 215–286, Mar. 2007.

[40] Chen, C. and Xi, H., "Combining programming with theorem proving," in *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pp. 66–77, ACM Press, 2005.

[41] McBride, C., "Faking it simulating dependent types in haskell," *Journal of Functional Programming*, vol. 12, pp. 375–392, July 2002.

[42] Sheard, T., "Putting curry-howard to work," in *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, (New York, NY, USA), pp. 74–85, ACM, 2005.

[43] Leijen, D. and Meijer, E., "Domain specific embedded compilers," in *Proceedings of the 2nd conference on Conference on Domain-Specific Languages - Volume 2*, DSL'99, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 1999.

[44] Fluet, M. and Pucella, R., "Phantom types and subtyping," *Journal of Functional Programming*, vol. 16, pp. 751–791, Nov. 2006.

[45] Kahrs, S., "Red-black trees with types," *Journal of Functional Programming*, vol. 11, no. 4, pp. 425–432, 2001.

[46] Eaton, F., "Statically typed linear algebra in haskell," in *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, (New York, NY, USA), pp. 120–121, ACM, 2006.

[47] Dagit, J., "Type-correct changes - a safe approach to version control implementation," Master's thesis, Mar.

[48] Meyer, B., "Applying "design by contract"," *Computer*, vol. 25, pp. 40–51, Oct. 1992.

[49] Barnett, M., Leino, K. R. M., and Schulte, W., "The spec# programming system: an overview," in *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, (Berlin, Heidelberg), pp. 49–69, Springer-Verlag, 2005.

[50] Peyton Jones, S., *Haskell 98 Language and Libraries: the Revised Report*. 2003.

[51] "Information technology - syntactic metalanguage - extended bnf," tech. rep., 8 1996.

[52] Wirth, N., "What can we do about the unnecessary diversity of notation for syntactic definitions?," *Communications of the ACM*, vol. 20, no. 11, pp. 822–823, 1977.

[53] Pierce, B. C. and Turner, D. N., "Local type inference," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 1, pp. 1–44, 2000.

[54] Claessen, K. and Hughes, J., "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, (New York, NY, USA), pp. 268–279, ACM, 2000.

[55] Barrett, C., Sebastiani, R., Seshia, S. A., and Tinelli, C., "Satisfiability modulo theories," in *Handbook of Satisfiability* (Biere, A., van Maaren, H., and Walsh, T., eds.), vol. 4, ch. 26, IOS Press, 2009.

[56] Hall, C. V., Hammond, K., Peyton Jones, S. L., and Wadler, P. L., "Type classes in haskell," *ACM Trans. Program. Lang. Syst.*, vol. 18, pp. 109–138, Mar. 1996.

[57] Leino, K. R. M., "Automating induction with an smt solver," in *Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, (Berlin, Heidelberg), pp. 315–331, Springer-Verlag, 2012.

[58] Roorda, J.-W., "Pure Type Systems for Functional Programming," Master's thesis, University of Utrecht, 2000.

[59] Hoare, C. A. R., "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.

[60] Hoare, T., "The verifying compiler: A grand challenge for computing research," *Journal of the ACM*, vol. 50, pp. 63–69, Jan. 2003.

[61] Kaufmann, M., Moore, J. S., and Manolios, P., *Computer-Aided Reasoning: An Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[62] Knowles, K., Tomb, A., Gronski, J., Freund, S. N., and Flanagan, C., "Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic," tech. rep., 2006.

[63] Slind, K., Bunker, A., and Gopalakrishnan, G., eds., *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*, vol. 3223 of *Lecture Notes in Computer Science*, Springer, 2004.