Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Εξαντλητικότητα Ταιριάσματος Προτύπων για Γενικευμένους Αλγεβρικούς Τύπους Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΑΡΑΧΑΛΙΑΣ ΓΕΩΡΓΙΟΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Εξαντλητικότητα Ταιριάσματος Προτύπων για Γενικευμένους Αλγεβρικούς Τύπους Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΑΡΑΧΑΛΙΑΣ ΓΕΩΡΓΙΟΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Ιανουαρίου 2014.

Νικόλαος Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.	Κωστής Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Κώστας Κοντογιάννης Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2014

Καραχάλιας Γεώργιος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ο Glasgow Haskell Compiler (GHC) είναι αυτή τη στιγμή ο κορυφαίος μεταγλωττιστής για τη γλώσσα Haskell. Πέραν του προτύπου της Haskell 2010, υποστηρίζει πολυάριθμες επεκτάσεις, συμπεριλαμβανομένων των *Γενικευμένων Αλγεβρικών Τύπων*. Δυστυχώς, ενώ οι γενικευμένοι αλγεβρικοί τύποι έχουν ενσωματωθεί πλήρως στο σύστημα τύπων της Haskell, ο εντοπισμός μη εξαντλητικών ταιριασμάτων προτύπων σε περιπτώσεις που περιέχουν γενικευμένους αλγεβρικούς τύπους αποδείχτηκε ελλιπής.

Ο στόχος αυτής της διπλωματικής εστιάζει στο σχεδιασμό ενός μηχανισμού για τον εντοπισμό μη εξαντλητικών ταιριασμάτων προτύπων, ο οποίος θα διαχειρίζεται τόσο ταιριάσματα προτύπων επί Αλγεβρικών Τύπων όσο και Γενικευμένων Αλγεβρικών Τύπων, με ενιαίο τρόπο. Ο βασικός μας ισχυρισμός είναι ότι ο έλεγχος της εξαντλητικότητας ταιριάσματος προτύπων θα πρέπει να αντιμετωπίζεται πλέον περισσότερο ως σημασιολογικό, παρά συντακτικό ζήτημα. Αντί να είναι διαχωρισμένο απο τον έλεγχο τύπων, θεωρούμε πως πρέπει να ενσωματωθεί σε αυτόν, έτσι ώστε να είναι δυνατή η αξιοποίηση των τοπικών περιορισμών που εισαγάγονται απο ταιριάσματα γενικευμένων αλγεβρικών τύπων.

Στην παρούσα εργασία παρουσιάζουμε έναν τέτοιο μηχανισμό, ως επέκταση του τρέχοντος, ο οποίος είναι εύληπτος, εύκολα υλοποιήσιμος και αποδοτικός. Εκτός απο το σχεδιασμό, υλοποιήσαμε μερικώς την επέκταση μας στον GHC, συνεπώς παραθέτουμε και τεχνικές πληροφορίες, σχετικά με την υλοποίηση μας. Τέλος, συγκρίνουμε την απόδοση και τα αποτελέσματα που δίνει ο GHC με και χωρίς την επέκτασή μας, δείχνοντας ότι ο μηχανισμός μας είναι πράγματι ορθός και αποδοτικός, με αποτέλεσμα να μπορεί να ενσωματωθεί στο μέλλον στον εν λόγω μεταγλωττιστή.

Λέξεις κλειδιά

Ταίριασμα Προτύπων, Έλεγχος Τύπων, Γενικευμένος Αλγεβρικός Τύπος Δεδομένων, Glasgow Haskell Compiler, Haskell

Abstract

Glasgow Haskell Compiler (GHC) is the state-of-the-art compiler for the programming language Haskell. Striving to be on the edge, in March 2006, Generalized Algebraic Data Types (GADTs) were incorporated as an extension of GHC and, henceforth, GADTs turned from a specialized hobby into a mainstream programming technique. Unfortunately, although perfectly integrated in GHC's type system, along with numerous other features and extensions (type classes, type families, functional dependencies, implicit parameters, arbitrary-rank types and more besides), the detection of non-exhaustive pattern matches in cases that include GADTs proved to be incomplete.

The objective of our work was to design a new mechanism for the detection of non-exhaustive pattern matches, that, not only would be complete in a setting with GADTs (and all other GHC's exotic features), but would also treat all cases uniformly. Our main assertion is that exhaustiveness check must be promoted from a syntactic to a rather semantic matter. Instead of keeping it separated from the phase of type-checking, we suggest that the mechanism must be integrated into the latter. Hence, via constraint solving, we can exploit the local constraints introduced by GADT matches, in order to accurately detect missing patterns.

In this work, we present such a mechanism, as an extension of the previous one, that is considerably efficient and relatively easy to understand and implement. Apart from the design, we have also partially implemented it in GHC 7.7. Therefore, we additionally enclose further technical information, regarding our implementation. Finally, we compare the performance and the soundness of GHC with and without our extension, and show that our mechanism behaves well in such a setting.

Key words

Pattern Matching, Type Checking, Generalized Algebraic Data Types, Glasgow Haskell Compiler, Haskell

Ευχαριστίες

Κατ' αρχάς, θα ήθελα να ευχαριστήσω τον καθηγητή μου, Νικόλαο Παπασπύρου, όχι μόνο για τις πολύτιμες γνώσεις, αλλά κυρίως για την καθοδήγηση που μου προσέφερε κατα τις σπουδές μου και, ακόμα περισσότερο, ως επιβλέπων καθηγητής της διπλωματικής μου. Ιδιαιτέρως θα ήθελα επίσης να ευχαριστήσω τον Δημήτρη Βυτινιώτη. Με την ενεργή συμμετοχή και τις πολύτιμες συμβουλές του καθ' όλη την διάρκεια εκπόνησης της διπλωματικής μου, συνέβαλλε σημαντικά στην ολοκλήρωση αυτής της εργασίας. Επίσης, θα ήθελα πραγματικά να ευχαριστήσω και τον καθηγητή μου Δημήτρη Φωτάκη, για την πίστη του σε μένα και τον ενθουσιασμό που πάντα μου μετέδιδε με τον πιο όμορφο τρόπο, ωθώντας με να προσπαθώ για το καλύτερο. Εν συνεχεία, θα ήθελα να ευχαριστήσω τον συναδέλφους Γιάννη Τσιούρη και Κατερίνα Ρουκουνάκη, για την πολύτιμη συνεισφορά τους σε πολλά κρίσιμα σημεία εκπόνησης της διπλωματικής μου. Τέλος, θα ήθελα να ευχαριστήσω τα αδέρφια μου, όλους μου τους φίλους και ιδιαιτέρως τον αδερφό μου Λευτέρη, την θεία μου Βιργινία και την σύντροφό μου Φωτεινή, οι οποίοι, όλα αυτά τα χρόνια, στηρίζουν κάθε μου επιλογή έμπρακτα και με ενθουσιασμό.

Καραχάλιας Γεώργιος,

Αθήνα, 8η Ιανουαρίου 2014

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-1-14, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιανουάριος 2014.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Dedicated to my partner,

Foteini

Contents

List of Figures

Chapter 1

Introduction

1.1 Objectives

This thesis aims at designing a mechanism for detecting non-exhaustive pattern matches in pattern-matching constructs of Haskell code. Although the target was initially focused on GADT-matches, our goal also includes the design of a mechanism that solves the problem of non-exhaustiveness in general and treats uniformly both ADT and GADT matches.

Additionally, apart from the designing part, this thesis also aims at implementing such a mechanism in a real and practical compiler, like the *Glasgow Haskell Compiler*. This indicates that, instead of looking for just a theoretical solution, we seek for an efficient mechanism that can potentially be applied in other languages that support GADTs, with little or no adjustments at all.

1.2 Motivation

Strictly typed functional languages like Haskell have proved to be extremely useful in applications of high importance, due to their static typing. Languages with static typing are unlikely to produce runtime errors (since the majority of errors show up during compilation) and this feature makes them the most suitable for the implementation of robust applications with demanding specifications. Such applications include international banks' systems, narrowband software radio systems, modelling of hardware designs, and lots more.

Even more powerful and expressive are dependently typed languages like Coq or Agda, but, unfortunately, are very difficult to understand and use by the average programmer. Furthermore, efficiency reasons make them unsuitable for complex industrial applications, where high performance is necessary.

Haskell, on the other hand, is a state-of-the-art, general-purpose purely functional programming language. Starting from the other end, the robust and stable core of Haskell has been carefully extended the past years with concepts of the dependent-type theory, achieving a great increase in expressiveness, for a very modest cost in terms of intellectual and implementation complexity. That makes Haskell a great programming language that effectively maintains a balance, focused on real-world applications, whilst, cost-effetively preserving many important program properties and offering lightweight partial program verification.

More specifically, the *Glasgow Haskell Compiler* is currently the most powerful and complete compiler for Haskell, supporting numerous extensions and features (apart from the *Haskell 2010 Standard*). One of the features GHC has really early integrated into its type system is that of *Generalized Algebraic Data Types*, an object similar to the inductive families of data types, found in dependently-typed languages.

1.3 Outline of the Thesis

The rest of the thesis is organised as follows: First of all, in Chapter 2 we describe Algebraic Data Types and their generalization, GADTs. We focus mostly on how they can be used to structure a functional program, through relatively small examples in Haskell. Chapter 3 describes GHC's compilation pipeline, focusing on the front end that is closely related to our work. Additionally, this chapter also includes some more information on some specific aspects of GHC, that played a major role in the design of our mechanism. In Chapter 4 we describe our mechanism in detail: Firstly, we define the problem we came up against in a straightforward way. After that, we analyze our mechanism and cite the respective parts of our implementation. In Chapter 5 we evaluate our mechanism, through the results we collected from numerous tests. Not only completeness, but also efficiency is deduced. Finally, in Chapter 6, we conclude our work and present some future work that could be done on the subject.

Chapter 2

Algebraic Data Types and their Generalization

2.1 Algebraic Data Types

Algebraic Data Types (or simply ADTs) are a powerful feature that is supported by most programming languages but is part and parcel for Functional Programming Languages (FP), such as Clean, F#, OCaml, Scala, Standard ML and, of course, Haskell. They made their first appearance in programming languages in the 1970s, as part of *Hope* [Hope], a rather small functional language but quite important in the development of functional programming. In the next section we cite the definition of ADTs and illustrate their power, by describing several of their advantages and by showing how many types can be defined as ADTs, from built-in types such as integers or characters to even more complex ones, like lists and trees.

2.1.1 Definition, Terminology and Examples

One could say that an Algebraic Data Type is an ordered pair, consisting of a *Type Constructor*, i.e. a type-function that returns the declared type, together with a set of *Data Constructors*, i.e. a set of value-level functions that have as result values of the declared type. Using Haskell notation, data and type constructors start with an uppercase letter and type variables start with a lowercase letter and this is quite convenient. So, an ADT declaration in Haskell would look something like this:

```
data TypeConstructor  type_variables_list
   = DataConstructor1 type_parameter_list1
   | DataConstructor2 type_parameter_list2
   | ...
   | DataConstructorN type_parameter_listN
```

Listing 2.1: Type Definition Pattern

Now, let's take a step back and take a closer look at the pattern above. This small and innocent pattern arms us with numerous opportunities for structured and safe programming:

Enumerations If there are no type variables in the `type_variables_list` and no parameters in $\texttt{type\_parameter\_list}_i$, $\forall i$, we have an enumeration type. Types such as `Bool` or `WorkDay` can easily be defined using this syntax:

```
data Bool = True | False

data WorkDay = Monday | Tuesday | Wednesday | Thursday | Friday
```

Listing 2.2: Bool and WorkDay Type Declaration

Abusing notation, someone could also define `Int` and `Char` types this way:

```
data Int = -2147483648
         | ...
         | -1 | 0 | 1
         | ...
         | 2147483647

data Char = ... | 'a' | 'b' | ...
```

Listing 2.3: Possible Int and Char Type Declaration

It's quite surprising to know that the above definition of `Bool` is the actual definition in GHCs Prelude (type `Bool` is not a primitive type as far as concerns the typechecker)! Of course, primitive types' definitions (such as `Int` and `Char`) are a bit different but, surprisingly, not that far from the perspective we describe here.

Wrapper Types If there is only one data constructor, with only one type parameter, we have a wrapper for a specific type. This may seem useless at first sight but is quite useful in practice. Suppose we have the following definitions:

```
data Height = Height Float
data Weight = Weight Float
```

Listing 2.4: Height and Float Type Declarations

It's impossible for someone to add accidentally values of types `Height` and `Weight` because GHC's typechecker would complain that cannot match these two types. Type wrappers lead to readability and safety and, less obvious but quite useful, allow multiple implementations of the same functions upon the same underlying type[1].

Polymorphism Parameterized data types (potentially recursive) can be viewed as a kind of abstract types, offering great reusability. For example, the List type defined below, is abstract, in the sense that type variable `a` can be instantiated with *any* type, responding another `List` type every time.

```
data List a = Nil
            | Cons a (List a)
```

Listing 2.5: Parameterized List Type

A list of `Int`s will have type `List Int` whilst a list of `Bool`s has type `List Bool`. The function `length` we define below can work on every type of list.

```
length :: List a -> Int
length Nil         = 0
length (Cons x xs) = 1 + length xs
```

Listing 2.6: Length Function

Before moving on to the next chapter, it would be really important to discuss some more aspects of ADTs, abstraction and polymorphic types in Haskell.

[1] Since these properties are highly desirable, the Glasgow Haskell Compilation System provides us with **`newtype`** keyword, an alternative that is more efficiently implemented and is designed especially for this particular reason.

1. Defining ADTs, we described a *Type Constructor* as a *type-function*. I think that this deserves more thorough thought. In cases like `Bool`, `WorkDay`, `Height` or `Weight`, the type constructor is the type itself. In the case of list, `List Int` or `List Bool` surely are types but what exactly is `List` itself? This could be a good place to introduce *Concreteness*: We call a type `t` concrete if there can be values of this type `t`. It surely makes sense to have objects of type `List Int` (lists of `Int`s) but would it make sense to have an object of type `List`? If we ask for the kind of type constructor `List`, `ghci` gives us the following response:

```
> :kind List
List :: * -> *
```

Listing 2.7: Kind of Type Constuctor List

Now it's obvious that `List` needs a type-argument to yield a concrete type and (as expected) there cannot be a value of type `List`.

2. When declaring a datatype, sometimes it's quite useful to be explicit on types and kind of data and type constructors respectively. Increases readability and gives us a better intuition of what exactly we are defining and how it can be used later. We rewrite here the definition of `List`, under this perspective, using valid Haskell syntax:

```
data List :: * -> * where
  Nil  :: List a
  Cons :: a -> List a -> List a
```

Listing 2.8: List Type with Explicit Typing and Kinding

3. Finally, we could also use a mix of the previous definitions that looks like this:

```
data List a where
  Nil  :: List a
  Cons :: a -> List a -> List a
```

Listing 2.9: List Type with Explicit Typing

One has to notice the type variable `a` that spreads through the whole definition. Brought in scope in the head of the definition (`data List a where ...`), is later available to be used in the types of data constructors. It's the definite same type variable. In ADTs every data constructor's return type **must be** `List a`. Although this detail may seem unimportant (and maybe meaningless) at first sight, it's quite a restriction and the major drawback of Algebraic Data Types and we're going to prove it in the next section.

2.2 Generalizing Algebraic Datatypes

2.2.1 One Step Closer to GADTs

Suppose now we want to write an interpreter for a simply typed expression language with integers, booleans and pairs. We need a new data type to represent terms, a parser, some kind of a typechecker and an evaluator function. For starters, considering the parser-part trivial, we need an appropriate type `Term` to use thoughout all stages. In Listing 2.10 below we define such a type, as a simple Algebraic Data Type, using the typed notation we mentioned earlier.

```haskell
data Term where
  Lit  :: Int  -> Term
  Inc  :: Term -> Term
  IsZ  :: Term -> Term
  If   :: Term -> Term -> Term -> Term
  Pair :: Term -> Term -> Term
  Fst  :: Term -> Term
  Snd  :: Term -> Term
```

Listing 2.10: Simply Typed Term

It does not take long to find out there are two major points here:

Evaluator's Return Type Suppose we declare an `eval` function that takes a `Term` and returns the value calculated. Depending on the "type" of the `Term` we feed the evaluator, there may be multiple return types. This is well illustrated below:

$$\texttt{eval (Lit 7)} \rightsquigarrow \texttt{7}$$
$$\texttt{eval (IsZ (Lit 7))} \rightsquigarrow \texttt{False}$$
$$\texttt{eval (Pair (Lit 6) (Lit 7))} \rightsquigarrow \texttt{(6,7)}$$

We would have to define a new data type for values like the one presented below:

```haskell
data Value
   = VI Int         -- Integer Value
   | VB Bool        -- Boolean Value
   | VP Value Value -- Pair    Value
```

Listing 2.11: Values

Load for Typechecker Our evaluator has to check thoroughly the types of all mid-expressions in the expression to evaluate (and typecheck). The following definition shows this problem: We have to manually check that the recursive calls to `eval` return the expected type of values. This is both tiresome for the programmer and results to additional runtime overhead.

```haskell
eval :: Term -> Value
eval (Lit x) = VI x
eval (Inc t)
  | VI x <- eval t = VI (x+1)
  | otherwise = error "Inc: Not an Int"
eval (IsZ t)
  | VI x <- eval t = VB (x==0)
  | otherwise = error "IsZ: Not a Bool"
eval (If t x y)
  | VB b <- eval t = if b then eval x
                          else eval y
  | otherwise = error "If: Not a Bool"
eval (Pair x y) = VP (eval x) (eval y)
eval (Fst t)
  | VP v1 _ <- eval t = v1
  | otherwise = error "Fst: Not a Pair"
eval (Snd t)
  | VP _ v2 <- eval t = v2
  | otherwise = error "Snd: Not a Pair"
```

Listing 2.12: Evaluator

Reasonably enough, emerges the question: *"Since our types are already supported by Haskell, couldn't we just take advantage of GHC's typechecker and let it do the work for us?"*. And that's *exactly* what we're going to do.

Before revealing the solution to our problem, it's worth to forecast the best direction. Observing the three terms we evaluated before, one could presume their types by giving just a glance, it's like their structure "gives away" their type. If only we could encapsulate some more of the structural information in the type of every term, it would be GHC's typechecker matter to typecheck only well-typed terms and infer the best return type for our `eval` function. We generalize ADTs by abstracting over the type of each constructor and we are finally led to *Generalized Algebraic Data Types* (GADTs):

2.2.2 GADTs in Action

We are *abstracting over* in the sense that we lift the restriction of the same result type for each data constructor. We supply type `Term` with a type parameter, indicating the type of the term it represents and end up with the following declaration of our `Term` type:

```
data Term a where
  Lit  :: Int -> Term Int
  Inc  :: Term Int -> Term Int
  IsZ  :: Term Int -> Term Bool
  If   :: Term Bool -> Term a -> Term a -> Term a
  Pair :: Term a -> Term b -> Term (a,b)
  Fst  :: Term (a,b) -> Term a
  Snd  :: Term (a,b) -> Term b
```

Listing 2.13: Simply Typed Term using GADTs

Henceforth, malformed terms like `IsZ (IsZ (Lit 10))` are statically rejected as ill-typed: Term `Lit 10` has type `Term Int` and by applying constructor `IsZ` we get a term of type `Term Bool`. By applying again constructor `IsZ` we are led to a type-error since `IsZ` expects a term of type `Term Int` and not `Term Bool` we feed it with.

Furthermore, armed with this level of generalization, the declaration of our evaluating function `eval` is now absolutely direct:

```
eval :: Term a -> a
eval (Lit i)    = i
eval (Inc t)    = eval t + 1
eval (IsZ t)    = eval t == 0
eval (If t a b) = if eval t then eval a else eval b
eval (Pair a b) = (eval a, eval b)
eval (Fst t)    = fst (eval t)
eval (Snd t)    = snd (eval t)
```

Listing 2.14: Simply Typed Evaluator

2.2.3 Actual Representation

Although the change in the syntax is quite small, the impact on the typechecker and the type-inference engine is tremendous. Not clear from the previous syntax is the fact that GADTs introduce *local constraints*[1]. When matching against a GADT expression, these constraints come in scope, to be used by

[1] By *local constraints* we mean type constraints that hold in some parts of the program but not others. A more detailed description of the term can be found in *OutsideIn(X)* [Vyti11] and *System F with Type Equality Coercions* [Sulz07].

the type-checker. We rewrite here the above definition of `Term a` with explicit type equality constraints (using valid Haskell syntax):

```
data Term a where
  Lit  :: forall a. (a~Int)  => Int -> Term a
  Inc  :: forall a. (a~Int)  => Term Int -> Term a
  IsZ  :: forall a. (a~Bool) => Term Int -> Term a
  If   :: forall a. Term Bool -> Term a -> Term a -> Term a
  Pair :: forall a b c. (a~(b,c)) => Term b -> Term c -> Term a
  Fst  :: forall a b. (a~b)  => Term (b,c) -> Term a
  Snd  :: forall a c. (a~c)  => Term (b,c) -> Term a
```

Listing 2.15: Term GADT with explicit equality constraints

Regarding our work, one major point can be extracted from the above definition:

All classic pattern-matching techniques may fail to identify non-exhaustive pattern-matches in a world with GADTs because of the local constraints. Considering the simple ADT definition of Listing 2.9, a function expecting an object of type `List a` can handle both constructors `Nil` and `Cons`. In contrast, a function that expects an argument of type `Term Int` can handle patterns as `Lit x` and `Inc t` but not `IsZ t`. A classic algorithm would warn about `IsZ _` to be a pattern not matched, although, if it appeared, would be a piece of inaccessible code. Obviously, classic techniques do not exploit type information so are unable to detect special cases like this one. Before presenting our solution to this problem (Chapter 4), we briefly describe in the next chapter (Chapter 3) the *Glasgow Haskell Compilation System's* structure, in order to grant to the reader the ability to understand GHC's compilation pipeline and judge our design choices reasonably.

Chapter 3

The Glorious Glasgow Haskell Compilation System

3.1 Overview

The Glorious Glasgow Haskell Compilation System (more commonly known as the Glasgow Haskell Compiler) is a highly optimising native code compiler for the functional programming language Haskell. Currently the largest and most powerful compiler for Haskell, GHC not only supports the Haskell 98 and 2010 Standard, but also, numerous extensions, aggressive optimizations and one of the most comprehensive error-messaging. Although all these features increased the complexity of the project exponentially, it still is one highly structured program that can be relatively easy to extend and modify. Of course, the six-staged compiler model (lexical analysis, preprocessing, syntactic and semantic analysis, code generation and optimization) is not enough for all these features and GHC, not only goes through a lot more phases, but also rearranges the basic ones. The main focus of this chapter is to help the reader get acquainted with the internals of GHC and prepare himself for what lies ahead. Therefore, the chapter is structured as follows:

1. In the first section we describe each phase of the compilation pipeline, along with the types that are passed around in each phase[1]. Additionally, we include a diagram that displays all this information in a more compact way.
2. In the next section, we focus on some specific aspects of GHC, mostly regarding the current mechanism for the detection of unused patterns. All these issues closely relate to our work and deeply affected our approaches towards our goal.

3.2 Compilation Pipeline

3.2.1 The Front End

The front end consists of the *Parser*, the *Renamer* and the *Typechecker*. All three of them share the same datatype, `HsSyn`, a type that is parameterized over the types of the term variables it contains. In more detail:

Parser GHC's Parser is written using *Alex*, for lexical analysis, *Happy*, for the parser itself, and *RdrHsSyn*, for Haskell support functions. During this phase, the type produced is `HsSyn` parameterized by `RdrName`, roughly speaking, a simple `String`. Since the language provides us with great syntactic flexibility (infix operators, many different precedences and fixities, whitespace indentation or curly braces to delimit blocks etc.), many things that one could consider to be done

[1] A more detailed description of GHC's compilation pipeline and general structure can be found at: *GHC Commentary: The Compiler:* `http://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler`

by the parser are not. For instance, infix operators are initially all parsed as right-associative, since their fixity and precedence is defined by the programmer and cannot be determined during this phase.

Renamer The Renamer sits between the parser and the typechecker and does most of the preprocessing needed for the next phases to operate. Its main task is to replace `RdrNames` with `Names`, i.e., an `RdrName` accompanied by a `Unique` that uniquely identifies it. The renamer is responsible for associating each identifier with its binding instance and ensuring that all occurences which associate to the same binding instance share a single `Unique`. Furthermore, the renamer deals with the following:

- Sorts out fixities: Using the fixities declared in the module, the renamer re-associates nested operator applications parsed by the parser (which, as we said, ignores fixities and precedence of operators).
- Does dependency analysis for mutually-recursive groups of declarations. Through this operation, the declarations are divided into strongly-connected components.
- Heavy lexical error checking: Through this phase, many errors and warnings can be issued, such as out-of-scope variables, unused bindings, unused imports, multiple usage of the same binder in a single pattern, shadowing bindings and many more.

Finally, one of the most important tasks of the renamer is to build the global rdr-env for the module, of type `GlobalRdrEnv`. This environment allows us to take a qualified or and un-qualified `RdrName` and figure out which `Name` it means. It's built by looking at all the imports and the top-level declarations of the module and is heavily used not only by the typechecker but also by the desugarer.

Typechecker Probably the most important phase in the front end of GHC. Through this phase, `Name` is replaced by `TcId` (or simply its type-synonym `Id`) which is simply a `Name` plus a `Type`. Rather unusual is the fact that GHC type checks programs in their original Haskell form (`HsSyn`), before the desugarer converts them into Core code (`CoreSyn`, a datatype with only eight constructors). Although this improves error messages, it really complicates things and effectively slows down the phase of typechecking, as it has to handle the much more verbose Haskell AST. Both renamer and typechecker use the same monad, `TcRn` and share the same entry point, the function `tcRnModule`, which is provided by the module *TcRnDriver*. This stage consists of two separate phases, *Constraint Generation* and *Constraint Solving*. The first is a traversal of the *Abstract Syntax Tree* (AST) during which, all type constraints are generated. The latter, solves all constraints and is (almost) the final phase of typechecking. The current system is described in detail (both theoretically and techincally) in [Vyti11], an outstanding work by Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers and Martin Sulzmann.

3.2.2 In Between

Desugarer After the three phases we previously described, comes the phase of *Desugaring*, where the massive `HsSyn` type (more than 52 data types) is converted to `CoreSyn`, GHC's Intermediate Language (only eight data constructors) [1]. Although this stage comes immediately after the type checker, the desugarer lives in another monad. Both use instantiations of `TcRnIf` monad, but, the renamer and the typechecker use `TcM` while `DsM` specializes the `TcRnIf` monad for the desugarer. Just before the end of the typechecking-phase, where all types are completely known, the environment of the desugarer is created, where, only the necessary information is kept and the rest is garbage-collected. From this, it can be clearly seen that the desugarer sets a definite

[1] The *Core Language* is basically Girard's System F [Gira89, pp. 81–148] enhanced with let, case and constructors' constructs. The current core language is actually the system $F_c^{\uparrow}$ [Yorg12], an extension of the previous F_c [Sulz07].

demarcation between the front and the back end. Quite importantly, although this stage focuses mostly on the transformation, during the process, produces a few errors and warnings. These include pattern-match overlap warnings and – most importantly for our work – pattern-match non-exhaustive warnings.

SimplCore Pass Now that we have the original program in Core Haskell, aggressive optimisations take place. This phase consists of many Core-to-Core passes that optimise the program through various transformations. The main passes include:

- *The Simplifier*. This pass applies lots of small and local optimisations.
- The *Float-in* and *Float-out* transformations, where let-bindings are moved inwards and outwards respectively, in order to produce faster code.
- *The Strictness Analyser*.
- *The Liberate-Case* transformation.
- *The Constructor-Specialisation* transformation.
- *The Common Sub-Expression Elimination (CSE)* transformation.

CoreTidy Pass During this procedure, the code gets into a form in which it can be imported into subsequent modules.

Data Flow Analysis The tidied Core program is dumped into an interface file. The process happens in two stages: At first the program is converted to IfaceSyn and, after that, IfaceSyn is serialised into a binary output file. This phase ends the intermediate process and now it's time for the core program to be fed to the Back End.

3.2.3 The Back End

Although our work relates mostly to the phases of typechecking and desugaring, we also cite here some information about the operations of the back end, mainly for the sake of completeness.

Core to STG At this phase, `CoreSyn` is transformed to `StgSyn` through a two-stage process. The first step is called CorePrep and put the program in A-Normal Form (ANF) where the argument of every application is either a variable or a literal (more complicated arguments are let-bound). The second step, called CoreToStg, moves to the `StgSyn` datatype, something quite easy after the CorePrep transformation has taken place.

Code Generation Now the STG program is converted by the Code Generator to a C-- program.

Many Back Ends Finally, depending on our needs, GHC provides us with three possible final outputs:

- If we are generating GHC's stylised C code, we can just pretty-print the C-- code as stylised C.
- If we are generating native code, we invoke the native code generator.
- If we are generating LLVM code, we invoke the LLVM code generator.

3.2.4 The Whole Picture

In Figure 3.1 we present the compilation pipeline of GHC, in a more concise way[1].

[1] This figure can also be found at GHC's Commentary, in Chapter HscMain.

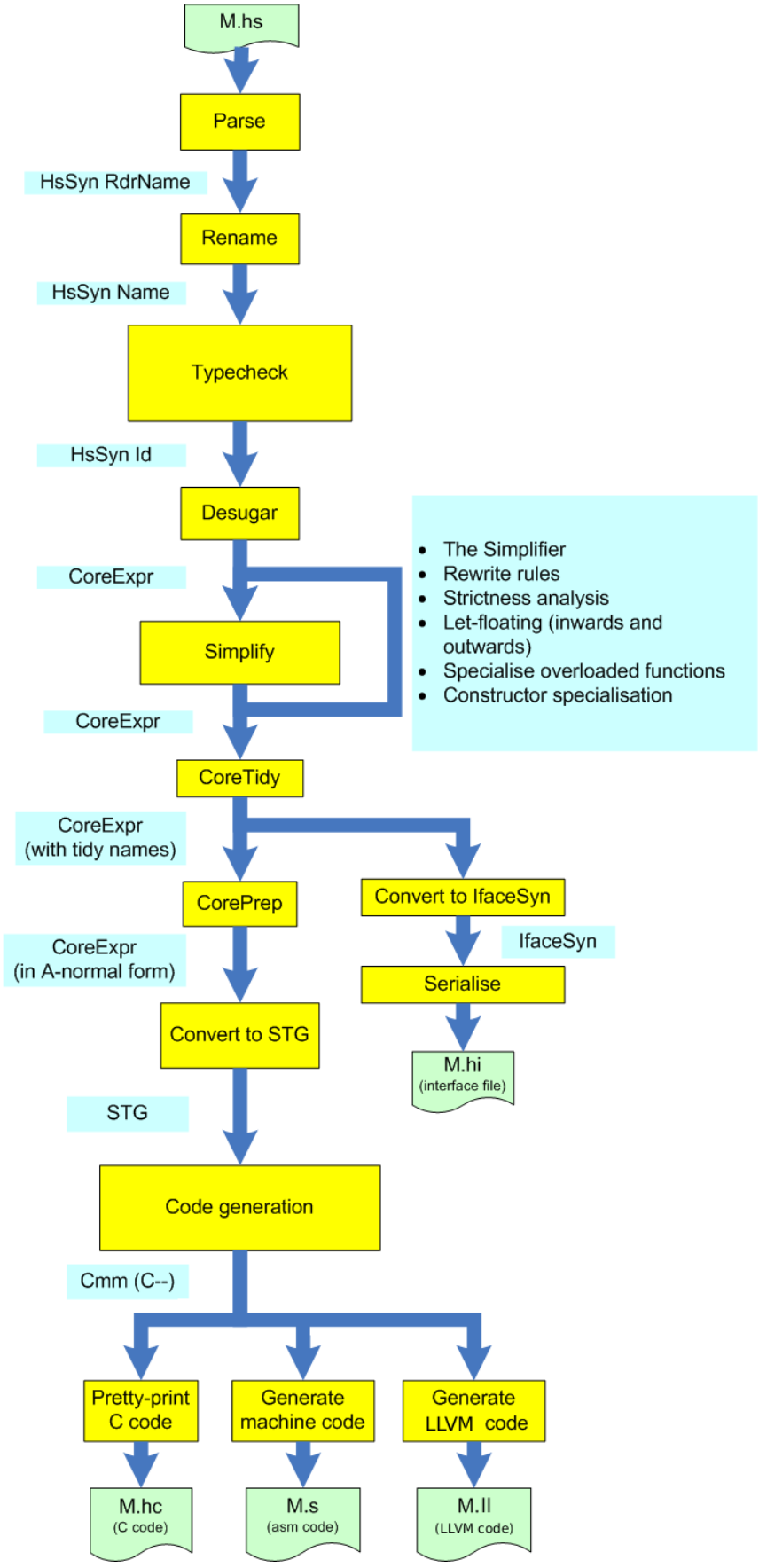

Figure 3.1: GHC's Compilation Pipeline

3.3 Important Technical Issues

3.3.1 In and Out Convention

Pattern's type (`Pat id`) provides us with two different data constructors to represent constructor patterns: `ConPatIn` and `ConPatOut`. Although this design choice may seem strange at first sight, it proves to be quite useful and is used frequently in GHC. The suffixes *-In* and *-Out*, stand for a pattern before and after typechecking, respectively. Intuitively, this means that before type checking all constructor patterns must use only the `ConPatIn` constructor, while, all type-checked constructor patterns will make use of `ConPatOut`. This simple convention is one more definite separation of typechecker and desugarer: The type checker can take as input *only* `ConPatIn` whilst, the desugarer can be fed *only* with `ConPatOut`.

3.3.2 Purity of the Mechanism

Although the exhaustiveness check is currently done in desugaring, the actual function that identifies all non-exhaustive pattern matches is a pure function. This proved to be of great importance, regarding our work, since we could "peek" at the unused patterns of a match, without having to bear side effects that could modify the global or local state irretrievably.

3.3.3 Impurity of the Type-Checker

For efficiency reasons, during type checking, type variables are represented by `TcRefs` (simple type synonym for IORef), i.e. mutable variables. Whilst this design choice significantly increases performance, makes the task of reasoning about our code really complex. Almost all type checking functions are accompanied by tons of side-effects: generation of fresh type variables for all expressions, instantiation of types, storing of type contrstraints etc.

3.3.4 TcM and DsM

As we previously mentioned, the phases of typechecking and desugaring live in different monads, TcM and DsM, respectively. This fact makes it impossible to move code from one phase to the other, a tremendous disadvantage that costs not only in adaptability but also in scalability. Of course, this is unavoidable, because carrying all the intermediate information along more stages would definately lead to space leaks.

3.3.5 Separation of Constraint Generation and Solving

Type constraints are not solved until they are all collected. Since the current type system supports type families [Schr08], phantom types [Xi03], [Chen03] and [Peyt06], promoted constructors [Yorg12], and many more features, the task of type checking has become a really challenging problem. Additionally, due to Haskell's *Open World Assumption*, new axioms can be added at any time, so, constraint solving cannot take place before all the needed type information is collected. As we stated before, the phases of generation and solution of the constraints are definately separated.

Chapter 4

Exhaustiveness of GADT Matches

4.1 Introducing The Problem

The mechanism used by GHC for the detection of unused patterns in non-exhaustive matches has only one drawback: While it takes into account type information (the check is done after typechecking, so, all types are completely known), it does not perform further constraint solving. Of course, before integrating GADTs into the system, type check during this phase was absolutely redundant. Since we have all types available, what else could we need? Indeed, in the case of ADTs, all constructors have the same result type, so, in a context where one can fit, all constructors –of the same type constructor– can. This does not hold in the case of GADTs, where each data constructor may return a different type. For example, recall the GADT Term definition 2.13. Although data constructors `Inc` and `IsZ` are in the same *family*[1], the one has result type `Term Int` and the other has result type `Term Bool`.

The rest of the chapter is divided in two sections. Firstly, we set forth the actual bug, through a series of some problematic situations. After that, we present our design choices, the different approaches we took and our final solution.

4.2 The Incompleteness of Pattern Matching

For the rest of the chapter, we are going to use the GADT definition shown in Listing 4.1, a rather small –and of little usefulness– definition but, enough to illustrate the incompleteness of GHC's exhaustiveness check.

```
1 data F :: * -> * -> * where
2   F1 :: F Int  Int
3   F2 :: F Int  Char
4   F3 :: F Char Int
5   F4 :: F Char Char
```

Listing 4.1: A Simple GADT Definition

4.2.1 Suppressing The Bug

In simple cases, like the `example1` (shown in Listing 4.2), the task is exactly the same like in an ADT definition. Since type variables `a` and `b` are free, any of the four constructors can match. The tedious warnings occur in more complex situations, like in `example2`. Since the signature of `example2`

[1] In GHC, we frequently call family a type constructor, along with the respective data constructors. If we refer to type families, we use explicitly the term *type family* and not just family, in order to overcome the ambiguity.

indicates that the parameter's type is `F Int Int`, not all constructors can appear in this context (to be exact, only `F1` can match the specified type). An older version of GHC whould issue a non-exhaustive warning for `example2`, regarding constructors `F2`, `F3`, and `F4`. One must notice that this specific case is quite easy to solve: If, the constructor `F4` appeared, the type equality `Int ~ Char` would have to hold, something that can be relatively easy to check. Since both types do not contain type variables, no unification is needed and we can check immediately for the equality of the types. Due to the simplicity of such a check (changes to test equality between instantiated types are minor), this test has been implemented in GHC, and, erroneous warnings are avoided in such trivial cases.

```
example1 :: F a b -> Int
example1 F1 = 1
example1 F2 = 2
example1 F3 = 3
example1 F4 = 4

example2 :: F Int Int -> Int
example2 F1 = 42

example3 :: F a a -> Int
example3 F1 = 1
example3 F4 = 4
```

Listing 4.2: Three Exhaustive Examples

4.2.2 General Case Still Unsolved

Although cases like the ones we previously mentioned are handled by GHC, the general case remained unsolved. This is well illustrated in the definition of function `example3`, in the Listing 4.2 above. When compiling the previous function, GHC issues the non-exhaustive warning presented in Listing 4.3 below. Of course, constructors `F2` and `F3` could not match such a context, because, for example `F2`, would introduce the set of constraints: C_{F2} = {a ~ b, Int ~ b, Char ~ b}, which is insoluble. If one was to add such a brach, the typechecker would complain that can't match type `Char` with `Int`, and fail with an error. Hence, the programmer is caught in the middle of this situation, compelled to tolerate such a warning, whilst his definition is actually complete[1]!

```
  Warning: Pattern match(es) are non-exhaustive
           In an equation for ‘example3’:
               Patterns not matched:
                   F2
                   F3
```

Listing 4.3: Warning Issued For example3

4.3 Extending The Mechanism

4.3.1 Basic Idea

All previous examples indicate that the mechanism used by GHC for detecting unused or missing patterns lacks a type-checking character. Additionally, we know that the mechanism issues *too many*

[1] Unfortnately, there is a tedious way to avoid such warnings. Since the algorithm for the detection of overlapping patterns is also incomplete, an addition of a *catch-all* branch (like: `example3 _ = error "inaccessible code"`) would suppress these messages. Although this seems to "solve" the problem, is actually a temporary solution, because the additional branch is actually inaccessible code and an overlapping pattern.

warnings and never less than actually should. Consequently, our main approaches were based on a simple idea: First, use the current mechanism to detect all unused patterns (possibly more than actual). Afterwards, for each pattern, generate all constraints that its appearance would introduce and, finally, filter them via constraint solving. The outcome of the solver uniquely determines which warning is to be issued and which is not:

Unification Failure If the unification fails, a branch introducing the pattern under consideration is inaccessible. If it appeared, we would have a type-error, so, it is not an actual missing pattern. In this case we should not issue a warning.

Successful Unification In case the solver succeeds to find a unifier, the pattern is actually missing. The appearance of the pattern under consideration at a call site would lead to a run-time error, and so, we should issue a warning.

Our work actually implements the mechanism described, suitably adjusted, not only to take advantage of GHC's strict structure, but also with respect to the restrictions provided. In the next sections we describe our mechanism in detail, including technical information about our implementation.

4.3.2 Approaching The Issue

Although only one approach seemed to work, throughout this procedure, we came up with several solutions and many different approaches that we cite in this section. Even if they seem irrelevant at first sight, I believe that they reveal many important aspects of GHC's structure and will help the reader deeply understand our design choices.

Core Haskell At first, we considered the phase of desugaring one suitable place to add our check: The existing mechanism already lived within the desugarer and many works are available towards this direction, like [Mitc08]. One great advantage of this approach is the fact that, since desugaring comes after type checking, we have type information on every term. Unfortunately, two major problems arose, that made this solution somewhat impossible. Firstly, when the original code is transformed in Core Haskell, a lot of information is lost about the initial appearance of the expressions. This leads to more hazy messages that are not of much use to the programmer. Secondly, and more important, during this phase, constraint solving cannot take place. Even if we ignored the hazy error messaging, in order to implement the check in the desugarer we would need extreme code refactoring.

Moving Backwards As the only phase of GHC's pipeline that supports constraint solving is the type checker, we stopped concerning ourselves with desugaring and all our subsequent approaches involved the phase of type checking. Since the erroneous warnings occured only in the case of GADT Constructors, an –initially tempting– solution was to filter out all constructors that do not introduce local constraints (ADT-Constructors) and handle the rest. The ones without the local constraints can easily be solved by the old mechanism, so, we would have to concern ourselves only with GADTs. Of course, such a solution would be messy, so we aimed at a mechanism that treats uniformly all cases, from `Int` patterns to GADT data constructors.

Need of Types The mechanism currently implemented in GHC is based on the *In and Out Assumption* (as described in Chapter 3). This adds one more restriction: In order for our mechanism to work, we had to place it after typechecking, but before desugaring. Since the typechecker traverses the AST (*Abstract Syntax Tree*), the suitest place seemed immediately after the typecheck of each pattern in the tree, in order to avoid one more traversal that would slow down the phase of type checking. Unfortunately, during the traversal no constraint solving takes place, just all constraints introduced are collected.

Split of the Phases Taking into consideration all the above restrictions lead us to the final solution: Immediately after constraint generation we detect the patterns and the respective type-constraints, and, after constraint solving, we filter them.

4.4 An Algorithm for Detecting Missing Patterns

Here we finally present our solution in detail. In Section 4.4.1 we describe our mechanism in a more structured and step-to-step way. Section 4.4.2 contains technical information, including functions and code fragments we implemented to achieve our objective.

4.4.1 The Mechanism

Our mechanism proceeds as follows:

1. Immediately after typecheching a match (or a group of matches, in case expressions and pattern matching), we call the implemented mechanism to detect all *missing patterns*.
2. For every pattern, we call the respecive type-checking function[1] to get the constraints that the respective pattern would introduce.
3. We store in the global state all patterns, along with the respective constraints, the context we are in (case-expression, let-expression, function binding, etc.) and the current location. The pattern, the context and the location are all needed to pretty print the warning, in case we finally need to.
4. After constraint solving we do the heavy work: For each set of constraints we call the constraint solver on a trial basis (we wouldn't want to accidentally add spurious error-messages in the error-list, about patterns that the programmer didn't even write) and, depending of the outcome, we issue the warning or not.

An oversimplified version of the mechanism as a flowchart can also be seen in Figures 4.1 and 4.2 (Our additions are coloured blue).

[1] All *type-checking* functions actually produce and store the constraints, they are not solving them. The convention is that every type-checking function has the prefix `tc-` (for example, `tcMatch`, `tcPat`, etc.). Our call is on a trial basis, since we only get the constraints, and do not store them, to avoid subverting the type-check process.

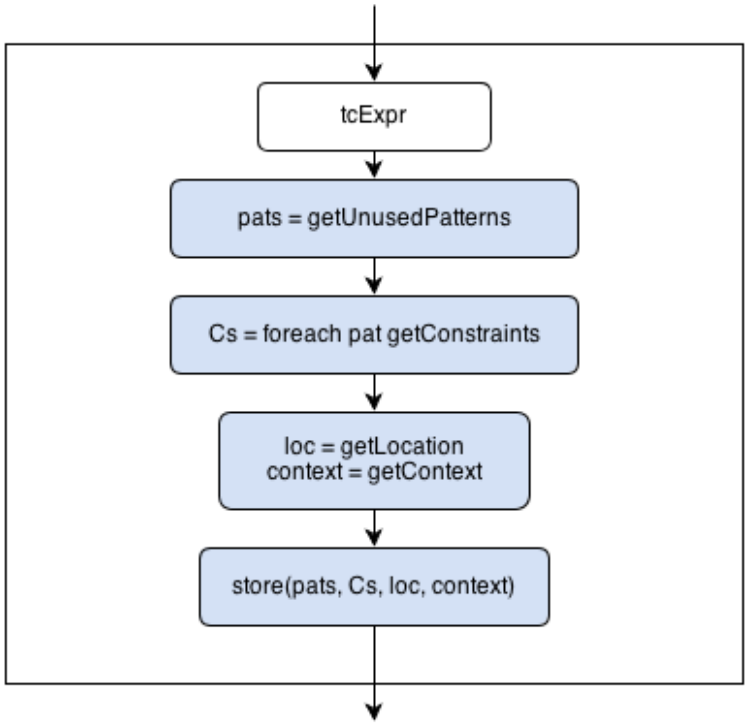

Figure 4.1: Mechanism's Phase 1: Constraint Generation Phase

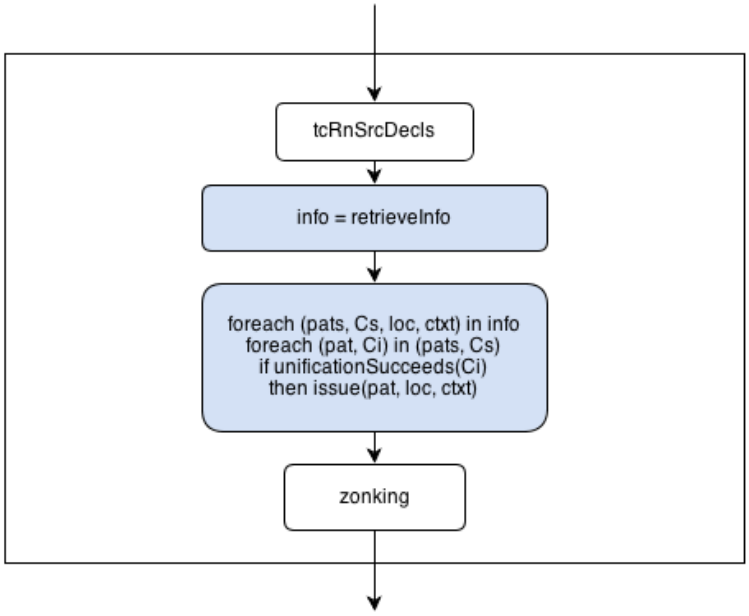

Figure 4.2: Mechanism's Phase 2: Constraint Solving Phase

4.4.2 Implementation

Prerequisites

Before diving into technical details, the reader must first get acquainted with the data types that are used most throughout our work:

Pat id The type of a pattern. It is parameterized by `id`, which, through the various phases of the compilation, may be one of `RdrName`, `Name`, or `TcId`. More specifically, after renaming `id` is `Name` and after type-checking `id` is an `TcId` (types `Name` and `TcId` are explained in section 3.2.1).

Match id body The type of a match. A typical match contains all patterns introduced in a match, along with a right-hand-side, that can make use of guards. Additionally, a match contains one more field, that contains a type signature for the result of the match.

MatchGroup id body To a first approximation, a list of `Located Matches`. Since this struct is used in function pattern bindings (and other structs), it contains also information about the types of the arguments and the result type of the whole match group.

EquationInfo Info about an equation. Contains the patterns for an equation, along with some more information that is quite useful during the phase of desugaring. It is normally produced after the type check of an expression.

ExhaustivePat The patterns to be warned about. Apart from the patterns, it also contains additional information about the set of used literals. For example, in function definition like in the Listing 4.4, we would get a warning like the one shown in Listing 4.5.

```
1 f :: Int -> Int
2 f 10 = 100
```

Listing 4.4: A Non-Exhaustive Match on Literals

```
1     Warning: Pattern match(es) are non-exhaustive
2              In an equation for ‘f’:
3                  Patterns not matched: GHC.Types.I# #x with #x ‘notElem‘ [10#]
```

Listing 4.5: Warning for f's Definition

Additionally, one convention that is extensively used in GHC, is that of `Located`: An `LPat` is simply a `Located Pat`, i.e. a pattern with a location. This naming convention is used for many types, including `Match` and `MatchGroup`.

Additions and Modifications

Although our original intention was to implement a solution that uniformly treats all pattern-matching structs, this was impossible, due to the syntactic difference between some terms. More specifically, case expressions and function pattern bindings use `MatchGroup`, whilst, let-bindings make direct use of `Pat`. Hence, we had to treat `MatchGroups` and let-bindings separately. The rest of the chapter contains a complete description of the functions we implemented or adjusted in GHC, along with the respective type signatures[1].

Extending the Global Environment One of the most important things to do, was to extend the global environment with one additional field. Since the mechanism is split in two phases, we could not pass around the missing patterns, from constraint generation to constraint solving. Instead, we added one more reference field in the global state, to store there all intermediate information while traversing the AST. When we are at the phase of constraint solving, we simply read from the reference the patterns collected from the whole file and process them one at a time. One may also notice that the type of field `back_up_unus` (shown in Listing 4.6) seems a bit clumsy. If we

[1] The whole implementation can be found in the appendix A. Implementation.

take into consideration the type synonyms, the type of it is actually:

`TcRef [(HsMatchContext Name, SrcSpan, [(ExhaustivePat, WantedConstraints)])]` that seems a lot better. Unfortunately, in order to avoid some potential cyclic-imports, we did not import the files with the respective type definitions.

```
back_up_unus :: TcRef [( HsMatchContext Name
                       , SrcSpan
                       , ((([LPat Name], [(Name, [HsLit])]), WantedConstraints)]
                       )]
```

Listing 4.6: Additions in typecheck/TcRnTypes.lhs

Exploitation of Solver's Incremental Nature As we have previously mentioned, GHC's constraint solver[1] makes heavy use of `IO` references, for efficiency reasons. One of the most important references it uses, is an `EvBindsVar` (an `IORef` type synonym), to store all intermediate information, during unification. This reference is initialized when entering the function, is used throughout the whole procedure, but is not returned as a result. In order to call the simplifier more times (more accurately, one time for every missing pattern), we had to adjust it, to grant us access to all intermediate information. To achieve this, we added two variations of function `simplifyTop`: `simplifyTopInit` and `simplifyTopIncr`, where the pass of `EvBindsVar` is done explicitly. Type signatures of all functions described here are shown in Listing 4.7 below:

```
simplifyTop     :: WantedConstraints -> TcM (Bag EvBind)
simplifyTopInit :: WantedConstraints -> TcM (Bag EvBind, EvBindsVar)
simplifyTopIncr :: WantedConstraints -> EvBindsVar -> TcM (Bag EvBind)
```

Listing 4.7: Additions in typecheck/TcSimplify.lhs

Collecting Constraints from Patterns In order to collect the constraints that a pattern would introduce, we had to implement two more functions (Listing 4.8), `getConstraintsFromPat` and `getConstraintsFromPats`. Almost identical, these two functions call typechecking functions `tcPat` and `tcPats` respectively, and collect the constraints generated. Although we call these functions from another modules, we placed them both in `TcPat`, because they are highly connected with the type checking of patterns.

```
getConstraintsFromPat  :: HsMatchContext Name
                       -> LPat Name -> TcSigmaType
                       -> TcM WantedConstraints
getConstraintsFromPats :: HsMatchContext Name
                       -> [LPat Name] -> [TcSigmaType]
                       -> TcM WantedConstraints
```

Listing 4.8: Additions in typecheck/TcPat.lhs

Storing of Patterns and Constraints For the storage of the necessary information, we implemented the function `storeUnusedPatterns`, shown in Listing 4.9. The only thing this function does is to update the global state's reference with the supplied additional information. Although small, this function seemed right to be defined separately, for reusability and clarity. Since its main interaction concerns the `TcRn` monad, we placed it in *TcRnMonad.lhs*, a file that contains functions for working with the typechecker environment.

[1] Constraint Solver's entry point is the function **`simplifyTop`**.

```
storeUnusedPatterns :: HsMatchContext Name
                    -> SrcSpan
                    -> [([LPat Name], [(Name, [HsLit])])]
                    -> [WantedConstraints]
                    -> TcM ()
```

Listing 4.9: Additions in typecheck/TcRnMonad.lhs

Handling of MatchGroups The manipulation of `MatchGroups` is rather straightforward: After the typecheck of the match-group has taken place, we collect all missing patterns via the previous mechanism used by GHC (through the type-checked match-group, due to the reasons discussed in section 3.3.1). For each set of patterns, we collect all constraints using the function `getConstraintsFromPats` we defined earlier and finally store them, using the function `storeUnusedPatterns`. The additional code to implement the above is shown in Listing 4.10. The type signature of function `extractMissingPatterns` that exploits the previous mechanism can be found in Listing 4.11. One important issue that has to be discussed here is the additional code for checking if there are any errors. In case there are errors already, we do not trigger our mechanism, mainly for two reasons:

Efficiency GHC does not issue warnings if there are errors. That means that it is only a waste of time to examine code for non-exhaustive pattern matches since they will never be issued.

Falsums In some cases, when GHC detects an error, there is no value to return, so, some functions may return falsums. If we went on with our mechanism, we would definately force these terms to be evaluated, leading to a compiler crash, that is highly undesirable.

```
  ...
  ; errs_var <- getErrsVar
  ; msgs <- readTcRef errs_var
  ; dflags <- getDynFlags
  ; current_loc <- getSrcSpanM
  ; let errs_flag = errorsFound dflags msgs
        hs_ctxt   = mc_what ctxt
  ; unless errs_flag $
      do { let (exhaust_pats, missing) = extractMissingPatterns match_group'
         ; wanteds <- forM missing $ \pats ->
             getConstraintsFromPats hs_ctxt pats pat_tys
         ; storeUnusedPatterns hs_ctxt current_loc exhaust_pats wanteds
         }
  ...
```

Listing 4.10: Additional code in function tcMatches

```
extractMissingPatterns :: MatchGroup TcId (Located (body TcId))
                       -> ([ExhaustivePat], [[LPat Name]])
```

Listing 4.11: Additions in typecheck/TcMatches.lhs

Handling of Let-Bindings The processing of let-bindings, although similar to the one we previously described, is a bit different. The steps we follow are:

- Filter the type-checked bindings. Function bindings are translated in `MatchGroups` and are handled by the previous mechanism. So, we have to keep only the pattern-bindings, through a simple filtering.
- Collect the respective types for the patterns filtered in the previous step. In order to collect the constraints we want, we need the expected type for each pattern-binding.

- For each pattern, detection of the missing patterns. In order to achieve this, we call function `getUnusedPatsFromLTcPatBind`, shown in Listing 4.14.
- For every missing pattern, we collect all constraints and store them in the global state.

The implementation of the mechanism is shown in Listing 4.12.

```
...
; errs_var <- getErrsVar
; msgs     <- readTcRef errs_var
; dflags   <- getDynFlags
; unless (errorsFound dflags msgs) $
    do { let pat_binds_tc  = filterMonoPatBinds tc_binds
             pat_exp_types = [ty | PatBind {pat_rhs_ty=ty}
                                 <- map unLoc (filterPatBinds binds')]
             zipped = zip pat_binds_tc pat_exp_types
       ; forM_ zipped $ \(l_tc_bind, rhs_ty_to_use) ->
           do { let exhaust_pats = getUnusedPatsFromLTcPatBind l_tc_bind
              ; let lpatss       = map fst exhaust_pats
              ; wcs <- forM lpatss $ \[pat] ->
                  getConstraintsFromPat PatBindRhs pat rhs_ty_to_use
              ; current_loc <- getSrcSpanM
              ; storeUnusedPatterns PatBindRhs current_loc exhaust_pats wcs
              }
        }
...
```

Listing 4.12: Additional code in function tcMonoBinds

```
isPatBind :: HsBind id -> Bool
```

Listing 4.13: Additions in hsSyn/HsBinds.lhs

```
filterPatBinds     :: [LHsBind id] -> [LHsBind id]
filterMonoPatBinds :: [Located TcMonoBind] -> [Located TcMonoBind]

mkEquationInfoFromLTcPatBind :: Located TcMonoBind -> EquationInfo
getUnusedPatsFromLTcPatBind  :: Located TcMonoBind -> [ExhaustivePat]
```

Listing 4.14: Additions in typecheck/TcBinds.lhs

Predicate `isPatBind` shown in Listing 4.13 is used by `filterPatBinds`, but, we defined it in another module. Since type `HsBind id` is defined in *HsBinds.lhs*, we believe that that is the *right place to be*.

Detection of Missing Patterns The final phase of the mechanism lives inside the module *TcRnDriver.lhs*, and, more specifically, function `tcRnSrcDecls`[1]. After constraint solving has taken place –and if there were no errors– our mechanism takes place. First, we retrieve all information from the global state. Then, we process sequentially all sets of patterns. Only if the flags enabled indicate that the current context is of interest, we do further search. In cases where we have patterns qualified with literal constraints, we issue the warning right away (literals are always of the same type, so, no further check is needed). In all other cases, we call the constraint solver (`simplifyTopIncr`) for each set of constraints, and, whether it fails or not to find a unifier, we do not or do issue a warning, respectively. For the pretty printing of the warnings, we use the function `tcIncompleteWarn`, shown in Listing 4.16. All these functions (including `tcIncompleteWarn`) are identical (or slightly adjusted to work within the `TcM` monad) to the respective functions in desugaring, since non-exhaustive pattern matches were previously detected during desugaring. The implementation is displayed in Listing 4.15 below.

[1] The entry point for the renaming and typechecking of a whole module.

```
  ...
  ; dflags <- getDynFlags
  ; tc_gbl_env <- getGblEnv
  ; let mmioref = tcg_back_up_unused tc_gbl_env
  ; all_cases <- readTcRef mmioref

  ; forM_ all_cases $ \(hs_match_context, location, list_of_pats) ->
      when (incomplete_flag dflags hs_match_context) $
        do { actual_pats <- filterM (\(exh_pat, wcs) -> do
               case (null (snd exh_pat)) of
                 True  -> do { (_msgs, mVal) <- tryTcErrs $
                                  simplifyTopIncr wcs ev_binds_var
                             ; return $ case mVal of { Nothing -> False
                                                     ; Just _  -> True}) }
                 False -> return True ) list_of_pats

           ; when (notNull actual_pats) $
               tcIncompleteWarn hs_match_context location (map fst actual_pats)
           }
  ...
```

Listing 4.15: Additional code in function tcRnSrcDecls

```
tcIncompleteWarn    :: HsMatchContext Name -> SrcSpan -> [ExhaustivePat] -> TcM ()
maximum_output      :: Int
pp_context          :: HsMatchContext Name
                    -> SrcSpan
                    -> SDoc
                    -> ((SDoc -> SDoc) -> SDoc)
                    -> SDoc
ppr_pats            :: Outputable a => [a] -> SDoc
ppr_incomplete_pats :: HsMatchContext Name -> ExhaustivePat -> SDoc
ppr_constraint      :: (Name,[HsLit]) -> SDoc
incomplete_flag     :: DynFlags -> HsMatchContext id -> Bool
```

Listing 4.16: Additions in typecheck/TcRnDriver.lhs

Cleaning Up Finally, we had to deactivate the previous mechanism, or else, we would have multiple warnings. The entry point was in file *deSugar/Match.lhs* and, more specifically, the function `matchCheck_really`. Unfortunately, till now, `matchCheck_really` used to handle both overlapping and missing patterns, so we could not remove it. Instead, we only removed the part that had to do with the detection of missing patterns. Additionally, we moved functions `incomplete_flag`, `dsIncompleteWarn`, `ppr_incomplete_pats` and `ppr_constraint` (for the pretty printing of warnings) in *typecheck/TcRnDriver.lhs*. The rest three functions we previously mentioned (`maximum_output`, `pp_context` and `ppr_pats`) had to be copied and not just moved, because they are also important in the issuing of overlapping-pattern-warnings.

Chapter 5

Evaluation of the Mechanism

In this chapter, we present the results we had, when testing our mechanism. Section 5.1 concerns soundness. We show that our mechanism passed almost all the tests, issuing the wanted warnings, in most cases tested. On the other hand, in Section 5.2, we concern ourselves with the efficiency of GHC. We show that, even on extremely large tests, the slowdown due to our additions is insignificant.

5.1 Testing

5.1.1 Proper Interaction With The System

First of all, we had to check that our additions interact properly with the existing system. In order to be sure about that[1], we run the two major sets of tests:

GHC-Build Since GHC uses bootstrapping, if our mechanism caused any undesirable effects, it would probably make the compiler crash. Unfortunately, this was not the most accurate test in our case, because GHC is highly structured and a quite robust application. Hence, not many non-exhaustive pattern-matches existed to check our mehanism. Additionally, GHC does not make heavy use of GADTs, because they were integrated recently into the system[2].

Testsuite The ultimate test for our implementation, was the testsuite of GHC, that includes 3685 tests, which gave rise to 14555 testcases. Due to some missing libraries (we tested GHC only with the basic libraries), the actual number of testcases was 11482, since the 3073 were skipped. More than enough, through these tests we came to conclude that the extended GHC does not do *worse*[3] than the vanilla version. Below we discuss our results in detail.

Figures 5.1 and 5.2 show the results we got when we ran the suite with the extended and the vanilla GHC, respectively. Of course, they are not of much use to anyone in this form, so, in Figure 5.3 we show the actual differences between our outputs.

[1] As sure as anyone can be in such cases. In any case, we are talking about testing, and not a mathematical proof here.
[2] GHC was initially released in 1989, whilst, GADTs are supported by GHC since March 2006.
[3] We had some failures, but not much different from the failures of the vanilla version.

```
       0 caused framework failures
       2 unexpected passes
      72 unexpected failures

Unexpected passes:
   driver  T5313 (ghci,dyn)

Unexpected failures:
   ../../libraries/base/tests         T7653 [bad exit code] (ghci,threaded1,threaded2)
   ../../libraries/base/tests         memo001 [bad stdout] (hpc)
   ../../libraries/base/tests/System  Timeout001 [bad exit code] (ghci,threaded1,threaded2)
   ../../libraries/random/tests       rangeTest [bad exit code] (optasm,threaded2,dyn
                                                                ,optllvm)
   codeGen/should_run                 cgrun015 [bad exit code] (ghci)
   codeGen/should_run                 cgrun068 [bad exit code] (ghci)
   concurrent/should_run              T367_letnoescape [bad exit code] (ghci)
   concurrent/should_run              T5611 [bad exit code] (ghci)
   concurrent/should_run              T7970 [bad stdout] (normal,hpc,optasm,threaded1
                                                         ,threaded2,dyn,optllvm)
   concurrent/should_run              T7970 [bad stdout or stderr] (ghci)
   concurrent/should_run              conc023 [bad exit code] (ghci,threaded1,threaded2)
   concurrent/should_run              conc064 [bad exit code] (ghci)
   concurrent/should_run              foreignInterruptible [bad exit code] (ghci)
   deSugar/should_compile             T5455 [stderr mismatch] (normal)
   dph/nbody                          dph-nbody-vseg-opt [bad exit code] (threaded1
                                                                        ,threaded2)
   driver                             werror [stderr mismatch] (normal)
   ffi/should_run                     T1288_ghci [bad exit code] (ghci)
   ffi/should_run                     T1679 [bad exit code] (ghci)
   ffi/should_run                     ffi003 [bad exit code] (ghci)
   ffi/should_run                     ffi021 [bad exit code] (ghci)
   ghc-api/T7478                      T7478 [bad exit code] (normal)
   ghci.debugger/scripts              dynbrk009 [bad exit code] (ghci)
   ghci.debugger/scripts              print020 [bad exit code] (ghci)
   ghci/linking                       ghcilink002 [bad exit code] (normal)
   ghci/linking                       ghcilink005 [bad exit code] (normal)
   numeric/should_run                 T7689 [exit code non-0] (dyn)
   numeric/should_run                 arith016 [bad exit code] (ghci)
   numeric/should_run                 expfloat [bad exit code] (ghci)
   parser/should_run                  readRun002 [bad exit code] (ghci)
   parser/should_run                  readRun004 [bad exit code] (hpc)
   perf/compiler                      T1969 [stat too good] (normal)
   perf/compiler                      T3294 [stat not good enough] (normal)
   perf/compiler                      T4801 [stat not good enough] (normal)
   perf/haddock                       haddock.Cabal [stat not good enough] (normal)
   perf/haddock                       haddock.base [stat not good enough] (normal)
   perf/haddock                       haddock.compiler [stat too good] (normal)
   perf/should_run                    T7797 [stat too good] (normal)
   perf/should_run                    lazy-bs-alloc [stat too good] (normal)
   programs/barton-mangler-bug        barton-mangler-bug [bad exit code] (hpc,optasm
                                                                        ,threaded2
                                                                        ,dyn,optllvm)
   programs/barton-mangler-bug        barton-mangler-bug [exit code non-0] (threaded1)
   programs/seward-space-leak         seward-space-leak [bad exit code] (ghci)
   rts                                T2047 [bad exit code] (ghci)
   rts                                T6006 [bad exit code] (hpc)
   rts                                T7919 [bad stdout] (dyn)
   rts                                T7919 [exit code non-0] (normal,hpc,optasm,threaded1
                                                             ,threaded2,optllvm)
   rts                                T7919 [bad stdout or stderr] (ghci)
   simplCore/should_compile           T7702 [stderr mismatch] (optllvm)
```

Figure 5.1: Testsuite Results: Extended Mechanism

```
       0 caused framework failures
       2 unexpected passes
      68 unexpected failures

Unexpected passes:
   driver  T5313 (ghci,dyn)

Unexpected failures:
   ../../libraries/base/tests         T7653 [bad exit code] (ghci,threaded1,threaded2)
   ../../libraries/base/tests         memo001 [bad stdout] (hpc)
   ../../libraries/base/tests/System  Timeout001 [bad exit code] (ghci,threaded1,threaded2)
   ../../libraries/random/tests       rangeTest [bad exit code] (optasm,threaded2,dyn
                                                                ,optllvm)
   ../../libraries/unix/tests         executeFile001 [bad exit code] (threaded2)
   array/should_run                   arr016 [exit code non-0] (optllvm)
   codeGen/should_run                 cgrun015 [bad exit code] (ghci)
   codeGen/should_run                 cgrun068 [bad exit code] (ghci)
   concurrent/should_run              T5611 [bad exit code] (ghci)
   concurrent/should_run              T7970 [bad stdout] (normal,hpc,optasm,threaded1
                                                         ,threaded2,dyn,optllvm)
   concurrent/should_run              T7970 [bad stdout or stderr] (ghci)
   concurrent/should_run              conc023 [bad exit code] (ghci,threaded1,threaded2)
   concurrent/should_run              foreignInterruptible [bad exit code] (ghci)
   dph/dotp                           dph-dotp-copy-opt [bad exit code] (threaded2)
   dph/nbody                          dph-nbody-vseg-opt [bad exit code] (normal,threaded1)
   ffi/should_run                     T1288_ghci [bad exit code] (ghci)
   ffi/should_run                     T1679 [bad exit code] (ghci)
   ffi/should_run                     ffi003 [bad exit code] (ghci)
   ffi/should_run                     ffi021 [bad exit code] (ghci)
   ghc-api/T7478                      T7478 [bad exit code] (normal)
   ghci.debugger/scripts              dynbrk009 [bad exit code] (ghci)
   ghci.debugger/scripts              print020 [bad exit code] (ghci)
   ghci/linking                       ghcilink002 [bad exit code] (normal)
   ghci/linking                       ghcilink005 [bad exit code] (normal)
   numeric/should_run                 T7689 [exit code non-0] (dyn)
   numeric/should_run                 arith016 [bad exit code] (ghci)
   numeric/should_run                 expfloat [bad exit code] (ghci)
   parser/should_run                  readRun002 [bad exit code] (ghci)
   parser/should_run                  readRun004 [bad exit code] (hpc)
   perf/compiler                      T1969 [stat not good enough] (normal)
   perf/compiler                      T3294 [stat not good enough] (normal)
   perf/compiler                      T4801 [stat not good enough] (normal)
   perf/haddock                       haddock.Cabal [stat not good enough] (normal)
   perf/haddock                       haddock.base [stat not good enough] (normal)
   perf/haddock                       haddock.compiler [stat too good] (normal)
   perf/should_run                    T7797 [stat too good] (normal)
   perf/should_run                    lazy-bs-alloc [stat too good] (normal)
   programs/barton-mangler-bug        barton-mangler-bug [bad exit code] (hpc,optasm
                                                                        ,threaded2,dyn
                                                                        ,optllvm)
   programs/seward-space-leak         seward-space-leak [bad exit code] (ghci)
   rts                                T7919 [bad stdout] (dyn)
   rts                                T7919 [exit code non-0] (normal,hpc,optasm,threaded1
                                                             ,threaded2,optllvm)
   rts                                T7919 [bad stdout or stderr] (ghci)
   simplCore/should_compile           T7702 [stderr mismatch] (optllvm)
```

Figure 5.2: Testsuite Results: Vanilla GHC

```
1  -    ../../libraries/unix/tests          executeFile001 [bad exit code] (threaded2)
2  -    array/should_run                    arr016 [exit code non-0] (optllvm)
3  -    dph/dotp                            dph-dotp-copy-opt [bad exit code] (threaded2)
4  +    concurrent/should_run               T367_letnoescape [bad exit code] (ghci)
5  +    concurrent/should_run               conc064 [bad exit code] (ghci)
6  +    rts                                 T2047 [bad exit code] (ghci)
7  +    rts                                 T6006 [bad exit code] (hpc)
8
9  +    deSugar/should_compile              T5455 [stderr mismatch] (normal)
10 +    driver                              werror [stderr mismatch] (normal)
11
12 -    dph/nbody                           dph-nbody-vseg-opt [bad exit code] (normal
13 -                                                                           ,threaded1)
14 +    dph/nbody                           dph-nbody-vseg-opt [bad exit code] (threaded1
15 +                                                                           ,threaded2)
16
17 -    perf/compiler                       T1969 [stat not good enough] (normal)
18 +    perf/compiler                       T1969 [stat too good] (normal)
19
20 +    programs/barton-mangler-bug         barton-mangler-bug [exit code non-0] (threaded1)
```

Figure 5.3: Actual Differences Between Testsuite Results

Let's discuss the differences shown in Figure 5.3 in detail:

1. **T5455**: This stderr-mismatch shown in line 9 is maybe the most interesting. The example `T5455` includes the definition of the function of the Listing 5.1. Vanilla GHC used to avoid issuing warnings is such cases, where the pattern bound is not actually demanded in the body of the let. Instead, we believe that warnings is such cases *must* be issued, mainly for two reasons:

   – Due to Haskell's laziness, if the pattern will not be demanded, it will *never* be evaluated. Hence, there is no reason to add such a let-binding in the first place.

   – Even ignoring the fact of lazy evaluation, the pattern binding is non-exhaustive anyway. If we had strict evaluation, such a pattern-binding could lead to a runtime error, so, the warning *should* be issued.

```
1 w :: String -> String
2 w x = let (_:_) = x in "1"
```

Listing 5.1: Function With Non-Demanded Pattern

2. **werror**: This stderr-mismatch shown in line 10 is nothing more than a rearrangement in the order of the issued warnings. The order in the line and column of the warnings is preserved, and the difference in the warnings, concerns only warnings of the same line and column. Hence, this mismatch is just a false alarm.

3. **T1969**: Almost nothing to discuss on this one, test T1969 shows statistics about the performance of the test *T1969.hs* of the suite. More specifically, this test case contains 300 data types (with one data constructor each), and 300 instances of a class (one instance for each data type). Remarkable is the fact that our extension has nothing to do with class instances, so, this improvement is somewhat unexpected. The only thing that could possibly contribute positively, is the way we handle pattern matches (in this file we had 300 exhaustive pattern matches, one for each type and the respective data constructor).

4. **barton-mangler-bug**: This difference shown in line 20 is rather cryptic. Both vanilla and adjusted GHC fail in the cases of hpc, optasm, threaded2, dyn, optllvm with bad exit code. The extended version seems to fail additionally in the case of threaded1 with non-0 exit code. Unfortunately, this difference in the exit codes was not available to further inspection. In any case, the whole program did not have any non-exhaustive match, so, our mechanism would not store any pattern at all, so, we believe that this mismatch is caused by another bug.

5. **dph-nbody-vseg-opt**: Like the previous one, this difference in lines 12-15, is quite irrelevant. Vanilla GHC fails in the cases of normal and threaded1, whereas, the adjusted GHC seems to fail in the cases of threaded1 and threaded2. Suprisingly enough, this test is also completely irrelevant with our work, and does not contain any missing pattern too.

6. **rest differences**: Finally, we have the mismatches shown in lines 1-7. Vanilla GHC used to fail in the cases shown in lines 1-3 while it shouldn't. Instead, the extended mechanism – for completely unknown reasons – passes these tests, but it fails in the tests of lines 4-7. Unfortunately, all 7 cases are completely irrelevant with our work (mostly related with parallelism, concurrency and ffi calls). Additionally, they contain only exhaustive matches, and, the second phase of our mechanism (the only one that could have some side effects) does not even come to work.

Although we had the differences discussed, we believe that the numbers are quite reassuring:

	Adjusted GHC	Vanilla GHC
expected passes	11122	11126
expected failures	141	141
unexpected passes	2	2
unexpected failures	72	68

5.1.2 Proper Non-Exhaustive Warnings

After that, we had to check that our implementation does *what it is supposed to do*: Issue the correct warnings in pattern-match constructs, even in cases that we are pattern matching against GADT constructors. In order to check that, we have written some tests of our own[1], that cover many possible settings. We descibe some of them below:

A Basic Test These are actually the same test, but, in order to check that our mechanism works in all pattern-match constructs, we had to test it on all of them[2]. In this testcase, only constructor `T1` can satisfy the type specified. Although obvious to the reader, GHC used to issue a warning on this one, before implementing our mechanism. The twist in this one is the fact that, if we change the type signature of `c` from `T a a -> Int` to `T Int Int -> Int`, GHC does not issue a warning about missing patterns. The reason for this is the patch described in Section 4.2.1. Since no type variables are introduced, constraint solving is not actually needed and the problem can be solved with a simple equality test.

```
1 {-# OPTIONS -XGADTs -XKindSignatures -Wall #-}
2
3 module GADTmatch01 where
4
5 data T :: * -> * -> * where
6   T1 :: Int  -> Int  -> T Int Int
```

[1] We have also collected some tests that have been issued in the GHC/Haskell community.

[2] Although we do not present it here, all the examples shown below have been tested like this. However, for readability reasons, we presnt only pattern-match constructs and case expressions, since are usually clearer.

```
  T2 :: Char -> Int  -> T Char Int
  T3 :: Int  -> Char -> T Int Char

-- Check on a case expression
c :: T a a -> Int
c x =
  case x of          -- should not issue warning
    T1 i j -> i+j
```

Listing 5.2: Test GADTmatch01case.hs

```
{-# OPTIONS -XGADTs -XKindSignatures -fwarn-incomplete-uni-patterns -Wall #-}

module GADTmatch01 where

data T :: * -> * -> * where
  T1 :: Int  -> Int  -> T Int Int
  T2 :: Char -> Int  -> T Char Int
  T3 :: Int  -> Char -> T Int Char

-- Check on a let-binding
f :: T a a -> Int
f x = let T1 i j = x -- should not issue warning
      in  i+j
```

Listing 5.3: Test GADTmatch01let.hs

```
{-# OPTIONS -XGADTs -XKindSignatures -Wall #-}

module GADTmatch01 where

data T :: * -> * -> * where
  T1 :: Int  -> Int  -> T Int Int
  T2 :: Char -> Int  -> T Char Int
  T3 :: Int  -> Char -> T Int Char

-- Check on a function pattern-matching
w :: T a a -> Int
w (T1 i j) = i+j     -- should not issue warning
```

Listing 5.4: Test GADTmatch01pm.hs

```
{-# OPTIONS -XGADTs -XKindSignatures -fwarn-incomplete-uni-patterns -Wall #-}

module GADTmatch01 where

data T :: * -> * -> * where
  T1 :: Int  -> Int  -> T Int Int
  T2 :: Char -> Int  -> T Char Int
  T3 :: Int  -> Char -> T Int Char

-- Check on a where-binding (syntactic sugar for let)
g :: T a a -> Int
g x = i+j
  where T1 i j = x   -- should not issue warning
```

Listing 5.5: Test GADTmatch01where.hs

Nested GADT Patterns The reason we have included this test, is to show that our mechanism behaves well, even in nested patterns. There is no actual missing pattern here, although the vanilla GHC issues one: `X2`.

```haskell
{-# OPTIONS_GHC -Wall -XGADTs -XKindSignatures #-}

module Nested where

data X :: * -> * -> * where
  X1 :: X Char Char
  X2 :: X Int  Char

data Y :: * -> * -> * where
  Y1 :: Int  -> Char -> Y Int  Char
  Y2 :: Char -> Char -> Y Char Char
  Y3 :: a    -> b    -> Y a    b

fxy :: Y (X a a) (X a a) -> a
fxy value = case value of
              Y3 X1 X1 -> 'a'
```

Listing 5.6: Test GADTmatch02nested.hs

A Simple Test This test, although embarrassingly simple, is left in our testsuite for historical reasons, since it's the first one I came across and my main challenge for a long time.

```haskell
{-# OPTIONS_GHC -Wall -XGADTs -XKindSignatures #-}

module Simple where

data Foo a b where
  F1 :: Foo Int  Bool
  F2 :: Foo Bool Int
  F3 :: Foo Int  Int
  F4 :: Foo Char Char

simple :: Foo a a -> Int
simple F3 = 42
```

Listing 5.7: Test GADTmatch03simple.hs

GADTs and Type Classes The next two examples show a behaviour of the mechanism that is quite interesting. Although our mechanism managed to avoid issuing a warning for the function `add` (as it is supposed to do) in Listing 5.1.2, it issued a warning for the function `ipow`. Although it seems wrong at a first glance (since the class constraints are unsatisfiable), it it's actually absolutely correct! This is well illustrated in the next program (Listing 5.1.2) where GHC issued no error for the unreachable branch. The reason for this behaviour is Haskell's *Open World Assumption*. Since new instances can be added at any time (and, more importantly **in any file**), the satifiability of the class constraint is required only at the call site. Hence, the type checker does not reject the program. Since our mechanism preserves the properties of the type system, if the missing pattern is type-correct, we *have* to issue the warning.

```haskell
{-# OPTIONS_GHC -Wall -XGADTs -XKindSignatures #-}

module Numerical where

data NumT :: * -> * where
  NumI :: Int    -> NumT Int
  NumF :: Float  -> NumT Float
  NumD :: Double -> NumT Double

-- No Warning Should Be Issued Here
add :: (Num a) => NumT a -> NumT a -> NumT a
```

```
add (NumI x) (NumI y) = NumI $ x + y
add (NumF x) (NumF y) = NumF $ x + y
add (NumD x) (NumD y) = NumD $ x + y

ipow :: (Integral b, Num a) => NumT a -> NumT b -> NumT a
ipow (NumI x) (NumI y) = NumI $ x ^ y
ipow (NumF x) (NumI y) = NumF $ x ^ y
ipow (NumD x) (NumI y) = NumD $ x ^ y
```

Listing 5.8: Test GADTmatch04numerical.hs

```
{-# OPTIONS_GHC -Wall -XGADTs -XKindSignatures #-}

module OpenW1 where

data Gen :: * -> * where
  GenS :: (Num a) => a -> Gen [a]
  GenT :: (Integral a, Integral b) => a -> b -> Gen [(a,b)]

list :: (Num a) => [Gen [a]] -> [a]
list [] = []
list ((GenS   x):rest) = x : list rest
list ((GenT _ _):rest) = 1 : list rest -- (a,b) NOT an instance of Num (not
                                       -- exported from the prelude. Hence,
                                       -- currently unreachable.
```

Listing 5.9: Test GADTmatch05openw.hs

GHC Tickets The next three programs are more or less covered by the previous examples but I believe that had to be included in this work. All of them are official bugs stated in the GHC Community and, since our mechanism is handling all cases uniformly, all of them are covered.

```
{-# OPTIONS_GHC -Wall -XGADTs #-}

module GADTmatch07trac2006 where

data Expr a vs where
    EPrim   :: String -> a -> Expr a vs
    EVar    :: Expr a (a,vs)

interpret :: Expr a () -> a
interpret (EPrim _ a) = a
-- interpret EVar = error "unreachable"
```

Listing 5.10: Test GADTmatch07trac2006.hs

```
{-# OPTIONS_GHC -Wall -XGADTs #-}

module GADTmatch08trac366 where

data T a where
    C1 :: T Char
    C2 :: T Float

exhaustive :: T Char -> Char
exhaustive C1 = ' '
```

Listing 5.11: Test GADTmatch08trac366.hs

```haskell
{-# OPTIONS_GHC -Wall -XGADTs #-}

module GADTmatch09trac3927 where

data T a where
  T1 :: T Int
  T2 :: T Bool

-- f1 should be exhaustive
f1 :: T a -> T a -> Bool
f1 T1 T1 = True
f1 T2 T2 = False

-- f2 is exhaustive too, even more obviously
f2 :: T a -> T a -> Bool
f2 T1 (T1 :: T Int) = True
f2 T2 (T2 :: T Bool) = False
```

Listing 5.12: Test GADTmatch09trac3927.hs

Data Kinds Finally, we show that our mechanism behaves as expected in the presence of promoted types.

```haskell
{-# LANGUAGE DataKinds, GADTs, TypeOperators #-}

module GADTmatch10dkinds where

data Vect v a where
    Nil :: Vect '[] a
    Vec :: a -> Vect v a -> Vect (() ': v) a

instance Eq a => Eq (Vect v a) where
    (==) Nil Nil               = True
    (Vec e0 v0) == (Vec e1 v1) = e0 == e1 && v0 == v1
```

Listing 5.13: Test GADTmatch10dkinds.hs

```haskell
{-# OPTIONS_GHC -Wall -XDataKinds -XKindSignatures -XGADTs #-}

module GADTmatch11dksimple where

data Foo :: Bool -> * where
    A :: Foo False
    B :: Foo True

hmm :: Foo b -> Foo b -> Bool
hmm A A = False
hmm B B = True
```

Listing 5.14: Test GADTmatch11dksimple.hs

One Ticket Missing The following example is a case our implementation cannot handle at the moment. The commented `GBool` branch is actually inaccessible but we issue a warning about a missing pattern. Interesting is the fact that if we annotate the scrutinee `baz a` with the type `G Int`, the warning is not issued. The reason our extension behaves this way is quite simple: Our current implementation cannot exploit the type constraints that are extracted from the rest of the program. In our case, the constraint from the as-pattern is not used by our extension and the constraints introduced by the pattern `GBool` seem to be solvable while they are not. Hopefully, this is only an implementation issue and not a problem of the mechanism. Additionally,

although incomplete, the current implementation can handle all annotated top level definitions, so, the programmer can avoid such warnings with a type signature. In any case, in the near future, our implementation will follow precisely the mechanism we described, and such cases will be handled too.

```
{-# OPTIONS_GHC -Wall -XGADTs #-}

module Trac4139 where

data F a where
  FInt  :: F Int
  FBool :: F Bool

data G a where
  GInt  :: G Int
  GBool :: G Bool

class Baz a where
  baz :: F a -> G a

instance Baz Int  where { baz _ = GInt  }
instance Baz Bool where { baz _ = GBool }

bar :: Baz a => F a -> ()
bar a@(FInt) =
  case baz a of
    GInt -> ()
    -- GBool -> ()
bar _ = ()
```

Listing 5.15: Test GADTmatch06trac4139.hs

5.2 Performance

5.2.1 Trivial Tests

The least accurate tests –which we inevitably run– were the build of GHC and the testsuite. None of them is closely related to our mechanism, and sparsely use GADTs, or non-exhaustive matches. Therefore, these tests were not of much use in the measurement of our mechanism's time performance. The main reasons we ran these tests are:

- GHC is a large and well-written project. Although the cases we handle do not seem to appear very often in GHC, many different constructs and methods are used in it. Hence, through this performance test, we could check how the extended GHC performs in a general setting.
- The build of GHC is already time consuming. We definately wouldn't want to make it last even more.
- The performance test on testsuite, of even less use, was really simple to do. Since we would anyway test our mechanism on the testsuite, why not measure our time consumption on this one too?

Time results we got from the GHC and testsuite build are shown in Figure 5.4 below. Rather unexpected is the fact that GHC's build in both cases took about 1 hour and 47 minutes, the extended GHC

was 22 seconds faster that the original[1].

	Adjusted GHC	Vanilla GHC
GHC Build (ms)	6400179	6422403
Testsuite Build (ms)	17878182	17860428

Figure 5.4: Time Results of GHC & Testsuite Build

5.2.2 General Tests

Roughly speaking, our mechanism is slower than the original, for three reasons: *a*) For each potentially missing pattern, we call the typechecker to detect the respective type constraints. *b*) For each match, we store the potentially missing patterns along with the respective constraints in the global reference field. In contrast, the previous mechanism used to issue the warning as soon as the patterns were detected. *c*) Finally, for each pattern we call the constraint solver, on a trial basis. Specifically, if the pattern is not actually missing, the constraint solving may fail (literally or due to errors), so, we have the cost of the recovery (no negligible at all).

In order to measure such delays, we created large tests[2] like the `example1` of Listing 4.2. In these cases, the warnings issued by the vanilla GHC are identical to the ones the extended mechanism issues. Hence, through this test, we can approximately calculate the additional time cost per pattern, or even per match. The only factor we can't estimate through these tests is the potential failure and recovery of the constraint solver (all patterns may appear in the context, so, constraint solving always succeeds), but, we talk about that in the next section.

Number of Data Constructors	Adjusted GHC (ms)	Vanilla GHC (ms)	Delay (ms)
27	348	345	3
256	447	364	83
625	607	511	96
1296	1004	656	348
2041	1536	969	567
3125	2122	1298	824

Figure 5.5: Non-Exhaustive Performance Test Results

5.2.3 GADT-Specific Tests

Finally, we measure our performance in cases where vanilla GHC issues warnings while it shouldn't. We have created large tests[3] , where our functions are actually exhaustive. In order for our mechanism to ensure that there are actually no missing patterns, it has to run through all potential patterns, collect the respective constraints, try to solve them and *fail*. Hence, this is the best test to check the worst-case delay caused by our *trial-and-error* method.

[1] We believe that this speed-up can be attributed only to caching or other irrelevant reasons, since our mechanism performs at least as many operations as the original.

[2] These tests can be found in the appendix B. Performance Tests.

[3] These tests can also be found in the appendix B. Performance Tests.

Number of Data Constructors	Adjusted GHC (ms)	Vanilla GHC (ms)	Delay (ms)
27	321	318	3
256	454	376	78
625	712	513	199
1296	1146	656	490
2041	1818	983	835
3125	2527	1314	1213

Figure 5.6: Exhaustive Performance Test Results

5.2.4 Sum Up

In Figures 5.7 and 5.8 below are presented the previous results graphically. The blue line represents the exhaustive definitions and the green line represents the non-exhaustive ones. As expected, the behaviour of the extended mechanism is almost linear in the number of data constructors, since each data constructor introduces (most of the times) only a few constraints. Additionally, in Figure 5.8 one can see that if we have less than 50 data constructors (which is the most common case), the compilation delay does not exceed 5%, a really decent percentage, especially if we consider how much time it takes to compile a file with GHC.

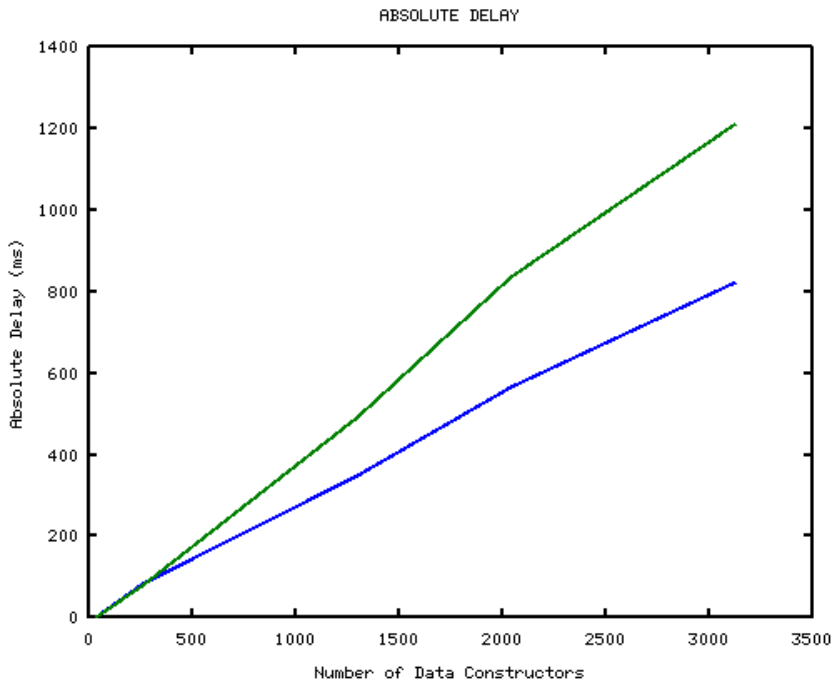

Figure 5.7: Absolute Delay

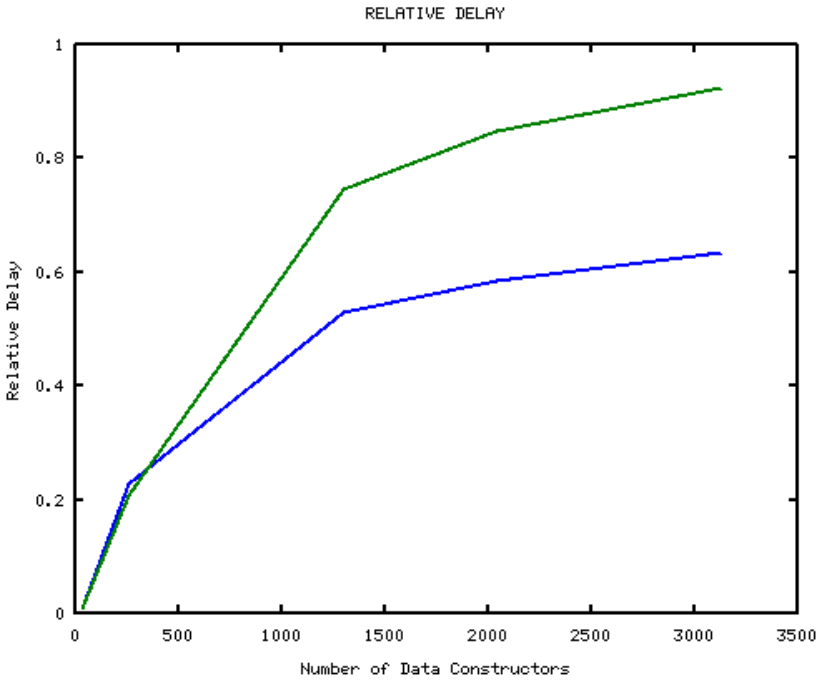

Figure 5.8: Relative Delay

Chapter 6

Conclusion

6.1 Contribution

Although small, we believe that this misbehaviour has contributed negatively to the usage of GADTs. Hence, through our work, we hope to make the setting more friendly for Haskell programmers, and motivate them to use this significant feature in a larger extent.

Furthermore, our mechanism enjoys the highly desirable properties of efficiency and simplicity. Due to this, we believe that it would be worth it to be implemented in other languages that support GADTs too. It is based on very simple and principal mechanisms (type-checking and exhaustiveness check on ADT matches) that are already performed by all compilers. So, such an extension would be relatively easy to implement.

6.2 Future Work

The exhaustiveness of pattern-match is a subject long studied and efficiently solved. We believe that, from the theoretical point of view, our work extends the previous mechanism in a rather general way, and, its soundness is somewhat self-evident. Hence, we believe that not much work could be done on this aspect of the subject.

On the other hand, there is much work that can be done, regarding the implementation. Some of the adjustments we intend to do in the near future include:

- Fully implement our extension. The extension suggests that cases like the one shown in Listing 5.1.2 are handled too, hence, we intend to extend our code to implement the exact procedure we have described.
- Complete separation of exhaustiveness and overlapping test. Till now, GHC provided a function named `check` as the entry point for both the detection of missing and overlapping patterns. Since these two tests are now separate, so must be their entry points. Although this could have caused some efficiency problems, Haskell's laziness helped us overcome such a situation. Since overlapping patterns are not demanded in our implementation, they are never actually computed and the potential slowdown is avoided.
- Shift of all respective functions (and files in general) from the desugarer to the folder that concerns the phase of type-checking. Henceforth, the exhaustiveness of pattern-match has nothing to do with the phase of desugaring, so, there is no reason for it to be there.

Bibliography

[Chen03] James Cheney and Ralf Hinze, "First-Class Phantom Types", Technical Report CUSIS TR2003-1901, Cornell University, 2003.

[Gira89] Jean-Yves Girard, *Proofs and Types*, Cambridge University Press, 1989.

[Hope] "Hope", http://www.soi.city.ac.uk/~ross/Hope/.

[Mitc08] Neil Mitchell and Colin Runciman, "Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching", in *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pp. 49–60, New York, NY, USA, 2008, ACM.

[Peyt06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich and Geoffrey Washburn, "Simple unification-based type inference for GADTs", in *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pp. 50–61, New York, NY, USA, 2006, ACM.

[Schr08] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty and Martin Sulzmann, "Type checking with open type functions", in *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pp. 51–62, New York, NY, USA, 2008, ACM.

[Sulz07] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones and Kevin Donnelly, "System F with type equality coercions", in *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '07, pp. 53–66, New York, NY, USA, 2007, ACM.

[Vyti11] Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers and Martin Sulzmann, "Outsidein(x) modular type inference with local assumptions", *J. Funct. Program.*, vol. 21, no. 4-5, pp. 333–412, September 2011.

[Xi03] Hongwei Xi, Chiyan Chen and Gang Chen, "Guarded recursive datatype constructors", in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pp. 224–235, New York, NY, USA, 2003, ACM.

[Yorg12] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis and José Pedro Magalhães, "Giving Haskell a promotion", in *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '12, pp. 53–66, New York, NY, USA, 2012, ACM.

Appendices

A. Implementation

In Chapter 4, we cited only the type signatures of the functions we implemented in GHC. The reader who wants to take a closer look at our work can find here the whole implementation.

```
-- | Additional Functions in HsBinds.lhs
isPatBind :: HsBind id -> Bool
isPatBind (PatBind {})   = True
isPatBind _other_pat_bnd = False

-- | Additional Functions in TcBinds.lhs
filterPatBinds :: [LHsBind id] -> [LHsBind id]
filterPatBinds binds = [b | b <- binds, isPatBind (unLoc b)]

filterMonoPatBinds :: [Located TcMonoBind] -> [Located TcMonoBind]
filterMonoPatBinds binds = [b | b <- binds, isTcPatBind (unLoc b)]
  where isTcPatBind (TcPatBind _ _ _ _) = True
        isTcPatBind _other_bind         = False

mkEquationInfoFromLTcPatBind :: Located TcMonoBind -> EquationInfo
mkEquationInfoFromLTcPatBind (L _ (TcPatBind _ pat _ _))
  = EqnInfo { eqn_pats = [unLoc pat]
            , eqn_rhs  = cantFailMatchResult undefined }
mkEquationInfoFromLTcPatBind (L _ _other_bind)
  = error "mkEquationInfoFromLTcPatBind called with TcFunBind"

getUnusedPatsFromLTcPatBind :: Located TcMonoBind -> [ExhaustivePat]
getUnusedPatsFromLTcPatBind mono_bind
  = fst $ check [mkEquationInfoFromLTcPatBind mono_bind]

-- | Additional Functions in TcMatches.lhs
extractMissingPatterns :: MatchGroup TcId (Located (body TcId))
                       -> ([ExhaustivePat], [[LPat Name]])
extractMissingPatterns match_group = (exhaust_pats, missing_pats)
  where exhaust_pats = fst (check (mkEqInfoPure match_group))
        missing_pats = map fst exhaust_pats

-- | Additional Functions in TcPat.lhs
getConstraintsFromPat :: HsMatchContext Name
                      -> LPat Name -> TcSigmaType
                      -> TcM WantedConstraints
getConstraintsFromPat hs_ctxt pat sig_ty
  = do { let tc = tcPat hs_ctxt pat sig_ty (return ())
       ; liftM snd (captureConstraints tc) }

getConstraintsFromPats :: HsMatchContext Name
                       -> [LPat Name] -> [TcSigmaType]
                       -> TcM WantedConstraints
getConstraintsFromPats hs_ctxt pats pat_tys
  = do { let tc = tcPats hs_ctxt pats pat_tys (return ())
       ; liftM snd (captureConstraints tc) }

-- | Additional Function in DsGRHSs.lhs
```

```
mkEqInfoPure :: MatchGroup Id (Located (body Id)) -> [EquationInfo]
mkEqInfoPure (MG { mg_alts = matches }) = map mkEqInfoPure1 matches
  where
    mkEqInfoPure1 :: LMatch Id (Located (body Id)) -> EquationInfo
    mkEqInfoPure1 (L _ (Match pats _ (GRHSs grhss _)))
      = EqnInfo { eqn_pats = map unLoc pats
                , eqn_rhs = foldr1 combineMatchResults (map lgrhsToMR grhss) }

    lgrhsToMR :: LGRHS Id (Located (body Id)) -> MatchResult
    lgrhsToMR (L _ (GRHS guards _)) = guardsToMR (map unLoc guards)

    guardsToMR :: [GuardStmt Id] -> MatchResult
    guardsToMR []                          = cantFailMatchResult undefined
    guardsToMR (BodyStmt e _ _ _ : stmts) | Just addTicks <- lhsexprToMR2 e
                                           = guardsToMR stmts
    guardsToMR (BodyStmt _ _ _ _ : stmts) = MatchResult CanFail undefined
    guardsToMR (LetStmt  _       : stmts) = guardsToMR stmts
    guardsToMR (BindStmt _ _ _ _ : stmts) = guardsToMR stmts
    guardsToMR _                           = panic "Guards: Non Exhaustiveness failure"

    lhsexprToMR2 :: LHsExpr Id -> Maybe (CoreExpr -> DsM CoreExpr)
    lhsexprToMR2 (L _ (HsVar v))
      | v `hasKey` otherwiseIdKey || v `hasKey` getUnique trueDataConId   = Just return
    lhsexprToMR2 (L _ (HsTick tickish  e)) | Just ticks <- lhsexprToMR2 e = Just ticks
    lhsexprToMR2 (L _ (HsBinTick ixT _ e)) | Just ticks <- lhsexprToMR2 e = Just ticks
    lhsexprToMR2 (L _ (HsPar e))           = lhsexprToMR2 e
    lhsexprToMR2 _                         = Nothing
```

Listing 1: Additional Functions (Part 1)

```
-- | Additional Functions in TcRnDriver.lhs
-- All of them are copies (or slighly adjusted) of the respective
-- ones in the desugarer. All useful for pretty-printing if warnings.

tcIncompleteWarn :: HsMatchContext Name -> SrcSpan -> [ExhaustivePat] -> TcM ()
tcIncompleteWarn kind loc pats
  = addWarnAt loc warn
      where
        warn = pp_context kind loc (ptext (sLit "are non-exhaustive"))
                          (\_ -> hang (ptext (sLit "Patterns not matched:"))
                                4 ((vcat $ map (ppr_incomplete_pats kind)
                                               (take maximum_output pats))
                                   $$ dots))

        dots | pats `lengthExceeds` maximum_output = ptext (sLit "...")
             | otherwise                           = empty

maximum_output :: Int
maximum_output = 4

pp_context :: HsMatchContext Name -> SrcSpan -> SDoc -> ((SDoc -> SDoc) -> SDoc) -> SDoc
pp_context kind _loc msg rest_of_msg_fun
  = vcat [ptext (sLit "Pattern match(es)") <+> msg,
          sep [ptext (sLit "In") <+> ppr_match <> char ':', nest 4 (rest_of_msg_fun pref)]]
  where
    (ppr_match, pref)
       = case kind of
            FunRhs fun _ -> (pprMatchContext kind, \ pp -> ppr fun <+> pp)
             _           -> (pprMatchContext kind, \ pp -> pp)

ppr_pats :: Outputable a => [a] -> SDoc
ppr_pats pats = sep (map ppr pats)

```

```
ppr_incomplete_pats :: HsMatchContext Name -> ExhaustivePat -> SDoc
ppr_incomplete_pats _ (pats,[]) = ppr_pats pats
ppr_incomplete_pats _ (pats,constraints) =
                        sep [ppr_pats pats, ptext (sLit "with"),
                             sep (map ppr_constraint constraints)]

ppr_constraint :: (Name,[HsLit]) -> SDoc
ppr_constraint (var,pats) = sep [ppr var, ptext (sLit "`notElem`"), ppr pats]

incomplete_flag :: DynFlags -> HsMatchContext id -> Bool
incomplete_flag dflags (FunRhs {})   = wopt Opt_WarnIncompletePatterns dflags
incomplete_flag dflags CaseAlt       = wopt Opt_WarnIncompletePatterns dflags
incomplete_flag _flags IfAlt         = False
incomplete_flag dflags LambdaExpr    = wopt Opt_WarnIncompleteUniPatterns dflags
incomplete_flag dflags PatBindRhs    = wopt Opt_WarnIncompleteUniPatterns dflags
incomplete_flag dflags ProcExpr      = wopt Opt_WarnIncompleteUniPatterns dflags
incomplete_flag dflags RecUpd        = wopt Opt_WarnIncompletePatternsRecUpd dflags
incomplete_flag _flags ThPatQuote    = False
incomplete_flag _flags (StmtCtxt {}) = False
```

Listing 2: Additional Functions (Part 2)

```
-- | Additional Function in TcRnMonad.lhs

storeUnusedPatterns :: HsMatchContext Name
                    -> SrcSpan
                    -> [([LPat Name], [(Name, [HsLit])])]
                    -> [WantedConstraints]
                    -> TcM ()
storeUnusedPatterns match_ctxt src_span ex_pats wcs
  = do { let pats_and_constraints = zip ex_pats wcs
       ; let info_to_store = (match_ctxt, src_span, pats_and_constraints)
       ; unused_ref <- liftM tcg_back_up_unused getGblEnv
       ; updTcRef unused_ref (info_to_store:) }
```

Listing 3: Additional Functions (Part 3)

```
-- | Additional functions in TcSimplify.lhs

simplifyTop :: WantedConstraints -> TcM (Bag EvBind)
-- Simplify top-level constraints
-- Usually these will be implications,
-- but when there is nothing to quantify we don't wrap
-- in a degenerate implication, so we do that here instead
simplifyTop wanteds
  = do { traceTc "simplifyTop {" $ text "wanted = " <+> ppr wanteds
       ; ev_binds_var <- newTcEvBinds
       ; binds <- simplifyTopIncr wanteds ev_binds_var
       ; traceTc "End simplifyTop }" empty
       ; return binds }

simplifyTopInit :: WantedConstraints -> TcM (Bag EvBind, EvBindsVar)
-- Simplify top-level constraints
-- Initializer for incremental version
simplifyTopInit wanteds
  = do { traceTc "simplifyTopInit {" $ text "wanted = " <+> ppr wanteds
       ; ev_binds_var <- newTcEvBinds
       ; binds <- simplifyTopIncr wanteds ev_binds_var
       ; traceTc "End simplifyTopInit }" empty
       ; return (binds, ev_binds_var) }
```

Listing 4: Additional Functions (Part 4)

```
-- | Adjustments in function tcMatches, in file TcMatches.lhs

tcMatches :: (Outputable (body Name)) => TcMatchCtxt body
          -> [TcSigmaType]      -- Expected pattern types
          -> TcRhoType          -- Expected result-type of the Match.
          -> MatchGroup Name (Located (body Name))
          -> TcM (MatchGroup TcId (Located (body TcId)))

tcMatches ctxt pat_tys rhs_ty (MG { mg_alts = matches })
  = ASSERT( not (null matches) )        -- Ensure that rhs_ty is filled in
    do  { matches' <- mapM (tcMatch ctxt pat_tys rhs_ty) matches
        ; let match_group' = MG { mg_alts = matches', mg_arg_tys = pat_tys, mg_res_ty =
            rhs_ty }

        -- Detection of Missing Patterns
        ; errs_var <- getErrsVar
        ; msgs <- readTcRef errs_var
        ; dflags <- getDynFlags
        ; current_loc <- getSrcSpanM
        ; let errs_flag = errorsFound dflags msgs
              hs_ctxt   = mc_what ctxt
        ; unless errs_flag $
            do { let (exhaust_pats, missing) = extractMissingPatterns match_group'
               ; wanteds <- forM missing $ \pats ->
                   getConstraintsFromPats hs_ctxt pats pat_tys
               ; storeUnusedPatterns hs_ctxt current_loc exhaust_pats wanteds
               }

        ; return match_group' }
```

Listing 5: Adjusted Functions (Part 1)

```
-- | Adjustments in function tcMonoBinds, in file TcBinds.lhs
tcMonoBinds :: TopLevelFlag
            -> RecFlag  -- Whether the binding is recursive for typechecking purposes
                        -- i.e. the binders are mentioned in their RHSs, and
                        --      we are not rescued by a type signature
            -> TcSigFun -> LetBndrSpec
            -> [LHsBind Name]
            -> TcM (LHsBinds TcId, [MonoBindInfo])

tcMonoBinds top_lvl is_rec sig_fn no_gen
           [ L b_loc (FunBind { fun_id = L nm_loc name, fun_infix = inf,
                                fun_matches = matches, bind_fvs = fvs })]
                             -- Single function binding,
  | NonRecursive <- is_rec   -- ...binder isn't mentioned in RHS
  , Nothing <- sig_fn name   -- ...with no type signature
  =     -- In this very special case we infer the type of the
        -- right hand side first (it may have a higher-rank type)
        -- and *then* make the monomorphic Id for the LHS
        -- e.g.         f = \(x::forall a. a->a) -> <body>
        --      We want to infer a higher-rank type for f
    setSrcSpan b_loc    $
    do  { rhs_ty  <- newFlexiTyVarTy openTypeKind
        ; mono_id <- newNoSigLetBndr no_gen name rhs_ty
        ; (co_fn, matches') <- tcExtendIdBndrs [TcIdBndr mono_id top_lvl] $
                               tcMatchesFun name inf matches rhs_ty

        ; return (unitBag (L b_loc (FunBind { fun_id = L nm_loc mono_id, fun_infix = inf,
                                              fun_matches = matches', bind_fvs = fvs,
                                              fun_co_fn = co_fn, fun_tick = Nothing })),
                  [(name, Nothing, mono_id)]) }
```

```
31
32 tcMonoBinds top_lvl _ sig_fn no_gen binds
33   = do  { tc_binds <- mapM (wrapLocM (tcLhs sig_fn no_gen)) binds
34
35         -- Bring the monomorphic Ids, into scope for the RHSs
36         ; let mono_info  = getMonoBindInfo tc_binds
37               rhs_id_env = [(name,mono_id) | (name, Nothing, mono_id) <- mono_info]
38                     -- A monomorphic binding for each term variable that lacks
39                     -- a type sig.  (Ones with a sig are already in scope.)
40
41         ; traceTc "tcMonoBinds" $ vcat [ ppr n <+> ppr id <+> ppr (idType id)
42                                        | (n,id) <- rhs_id_env]
43         ; binds' <- tcExtendIdEnv2 rhs_id_env $
44                     mapM (wrapLocM (tcRhs top_lvl)) tc_binds
45
46         ; errs_var <- getErrsVar
47         ; msgs     <- readTcRef errs_var
48         ; dflags   <- getDynFlags
49         ; unless (errorsFound dflags msgs) $
50             do { let pat_binds_tc  = filterMonoPatBinds tc_binds
51                      pat_exp_types = [ty | PatBind {pat_rhs_ty=ty}
52                                          <- map unLoc (filterPatBinds binds')]
53                      zipped = zip pat_binds_tc pat_exp_types
54                ; forM_ zipped $ \(l_tc_bind, rhs_ty_to_use) ->
55                    do { let exhaust_pats = getUnusedPatsFromLTcPatBind l_tc_bind
56                       ; let lpatss       = map fst exhaust_pats
57                       ; wcs <- forM lpatss $ \[pat] ->
58                           getConstraintsFromPat PatBindRhs pat rhs_ty_to_use
59                       ; current_loc <- getSrcSpanM
60                       ; storeUnusedPatterns PatBindRhs current_loc exhaust_pats wcs
61                       }
62                }
63
64         ; return (listToBag binds', mono_info) }
```

Listing 6: Adjusted Functions (Part 2)

```
1  -- | Adjustments in function tcSimplify (now renamed to tcSimplifyTopIncr),
2  -- in file TcSimplify.lhs
3
4  simplifyTopIncr :: WantedConstraints -> EvBindsVar -> TcM (Bag EvBind)
5  -- Simplify top-level constraints incrementally
6  simplifyTopIncr wanteds ev_binds_var
7    = do { traceTc "simplifyTopIncr {" $ text "wanted = " <+> ppr wanteds
8         ; zonked_final_wc <- solveWantedsTcMWithEvBinds ev_binds_var wanteds simpl_top
9         ; binds1 <- TcRnMonad.getTcEvBinds ev_binds_var
10        ; traceTc "End simplifyTopIncr }" empty
11
12        ; traceTc "reportUnsolved {" empty
13        ; binds2 <- reportUnsolved zonked_final_wc
14        ; traceTc "reportUnsolved }" empty
15
16        ; return (binds1 `unionBags` binds2) }
17
18   where
19     -- See Note [Top-level Defaulting Plan]
20     simpl_top :: WantedConstraints -> TcS WantedConstraints
21     simpl_top wanteds
22       = do { wc_first_go <- nestTcS (solve_wanteds_and_drop wanteds)
23                             -- This is where the main work happens
24            ; try_tyvar_defaulting wc_first_go }
25
26     try_tyvar_defaulting :: WantedConstraints -> TcS WantedConstraints
```

```
27      try_tyvar_defaulting wc
28        | isEmptyWC wc
29        = return wc
30        | otherwise
31        = do { free_tvs <- TcS.zonkTyVarsAndFV (tyVarsOfWC wc)
32             ; let meta_tvs = varSetElems (filterVarSet isMetaTyVar free_tvs)
33                     -- zonkTyVarsAndFV: the wc_first_go is not yet zonked
34                     -- filter isMetaTyVar: we might have runtime-skolems in GHCi,
35                     -- and we definitely don't want to try to assign to those!
36
37             ; meta_tvs' <- mapM defaultTyVar meta_tvs   -- Has unification side effects
38             ; if meta_tvs' == meta_tvs   -- No defaulting took place;
39                                          -- (defaulting returns fresh vars)
40               then try_class_defaulting wc
41               else do { wc_residual <- nestTcS (solve_wanteds_and_drop wc)
42                                    -- See Note [Must simplify after defaulting]
43                       ; try_class_defaulting wc_residual } }
44
45      try_class_defaulting :: WantedConstraints -> TcS WantedConstraints
46      try_class_defaulting wc
47        | isEmptyWC wc || insolubleWC wc
48        = return wc  -- Don't do type-class defaulting if there are insolubles
49                     -- Doing so is not going to solve the insolubles
50        | otherwise
51        = do { something_happened <- applyDefaultingRules (approximateWC wc)
52                                     -- See Note [Top-level Defaulting Plan]
53             ; if something_happened
54               then do { wc_residual <- nestTcS (solve_wanteds_and_drop wc)
55                       ; try_class_defaulting wc_residual }
56               else return wc }
```

Listing 7: Adjusted Functions (Part 3)

B. Performance Tests

All the tests described in both sections 5.2.2 and 5.2.3 use the same GADT data types. In order to get as accurate results as possible, we have first separately compiled the files containing the definitions and, after that, we have kept time of the compilation ***only for the main programs***. The files containing the definitions are the following:

```
{-# OPTIONS_GHC -Wall -XKindSignatures -XGADTs #-}

module Size0027F (F(..)) where

data F :: * -> * -> * -> * where
  MkF1  :: Int  -> Int  -> Int  -> F Int  Int  Int
  MkF2  :: Int  -> Int  -> Char -> F Int  Int  Char
  MkF3  :: Int  -> Int  -> Bool -> F Int  Int  Bool
  ...
  MkF27 :: Bool -> Bool -> Bool -> F Bool Bool Bool
```

Listing 1: File Size0027F.hs

```
{-# OPTIONS_GHC -Wall -XKindSignatures -XGADTs #-}

module Size0256H (H(..)) where

data H :: * -> * -> * -> * -> * where
  MkH1 :: Int -> Int -> Int -> Int -> H Int Int Int Int
  MkH2 :: Int -> Int -> Int -> Char -> H Int Int Int Char
  MkH3 :: Int -> Int -> Int -> Bool -> H Int Int Int Bool
  MkH4 :: Int -> Int -> Int -> String -> H Int Int Int String
  ...
  MkH256 :: String -> String -> String -> String -> H String String String String
```

Listing 2: File Size0256H.hs

```
{-# OPTIONS_GHC -Wall -XKindSignatures -XGADTs #-}

module Size0625X (X(..)) where

data X :: * -> * -> * -> * -> * where
  MkX1 :: Int -> Int -> Int -> Int -> X Int Int Int Int
  MkX2 :: Int -> Int -> Int -> Char -> X Int Int Int Char
  MkX3 :: Int -> Int -> Int -> Bool -> X Int Int Int Bool
  MkX4 :: Int -> Int -> Int -> String -> X Int Int Int String
  MkX5 :: Int -> Int -> Int -> Integer -> X Int Int Int Integer
  ...
  MkX625 :: Integer -> Integer -> Integer -> Integer -> X Integer Integer Integer Integer
```

Listing 3: File Size0625X.hs

```
{-# OPTIONS_GHC -Wall -XKindSignatures -XGADTs #-}
```

```haskell

module Size1296Y (Y(..)) where

data Y :: * -> * -> * -> * -> * where
  MkY1 :: Int -> Int -> Int -> Int -> Y Int Int Int Int
  MkY2 :: Int -> Int -> Int -> Char -> Y Int Int Int Char
  MkY3 :: Int -> Int -> Int -> Bool -> Y Int Int Int Bool
  MkY4 :: Int -> Int -> Int -> String -> Y Int Int Int String
  MkY5 :: Int -> Int -> Int -> Integer -> Y Int Int Int Integer
  MkY6 :: Int -> Int -> Int -> () -> Y Int Int Int ()
  ...
  MkY1296 :: () -> () -> () -> () -> Y () () () ()
```

Listing 4: File Size1296Y.hs

```haskell
{-# OPTIONS_GHC -Wall -XKindSignatures -XGADTs #-}

module Size2041Z (Z(..)) where

data Z :: * -> * -> * -> * -> * where
  MkZ1 :: Int -> Int -> Int -> Int -> Z Int Int Int Int
  MkZ2 :: Int -> Int -> Int -> Char -> Z Int Int Int Char
  MkZ3 :: Int -> Int -> Int -> Bool -> Z Int Int Int Bool
  MkZ4 :: Int -> Int -> Int -> String -> Z Int Int Int String
  MkZ5 :: Int -> Int -> Int -> Integer -> Z Int Int Int Integer
  MkZ6 :: Int -> Int -> Int -> () -> Z Int Int Int ()
  MkZ7 :: Int -> Int -> Int -> [Int] -> Z Int Int Int [Int]
  ...
  MkZ2401 :: [Int] -> [Int] -> [Int] -> [Int] -> Z [Int] [Int] [Int] [Int]
```

Listing 5: File Size2041Z.hs

```haskell
{-# OPTIONS_GHC -Wall -XKindSignatures -XGADTs #-}

module Size3125G (G(..)) where

data G :: * -> * -> * -> * -> * -> * where
  MkG1 :: Int -> Int -> Int -> Int -> Int -> G Int Int Int Int Int
  MkG2 :: Int -> Int -> Int -> Int -> Char -> G Int Int Int Int Char
  MkG3 :: Int -> Int -> Int -> Int -> Bool -> G Int Int Int Int Bool
  MkG4 :: Int -> Int -> Int -> Int -> String -> G Int Int Int Int String
  MkG5 :: Int -> Int -> Int -> Int -> Integer -> G Int Int Int Int Integer
  ...
  MkG3125 :: Integer -> Integer -> Integer -> Integer -> Integer
          -> G Integer Integer Integer Integer Integer
```

Listing 6: File Size3125G.hs

General

In this section we present the performance tests we have described in Section 5.2.2.

```haskell
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func1 where

import Size0027F (F(..))

```

```haskell
func1 :: F a b c -> Int
func1 (MkF1 _ _ _) = 1

main :: IO ()
main = print (42 :: Int)
```

Listing 1: File NonExhaustive/Func1.hs

```haskell
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func2 where

import Size0256H (H(..))

func2 :: H a b c d -> Int
func2 (MkH1 _ _ _ _) = 1

main :: IO ()
main = print (42 :: Int)
```

Listing 2: File NonExhaustive/Func2.hs

```haskell
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func3 where

import Size0625X (X(..))

func3 :: X a b c d -> Int
func3 (MkX1 _ _ _ _) = 1

main :: IO ()
main = print (42 :: Int)
```

Listing 3: File NonExhaustive/Func3.hs

```haskell
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func4 where

import Size1296Y (Y(..))

func4 :: Y a b c d -> Int
func4 (MkY1 _ _ _ _) = 1

main :: IO ()
main = print (42 :: Int)
```

Listing 4: File NonExhaustive/Func4.hs

```haskell
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func5 where

import Size2041Z (Z(..))

func5 :: Z a b c d -> Int
func5 (MkZ1 _ _ _ _) = 1

main :: IO ()
```

```
main = print (42 :: Int)
```

Listing 5: File NonExhaustive/Func5.hs

```
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func6 where

import Size3125G (G(..))

func6 :: G a b c d e -> Int
func6 (MkG1 _ _ _ _ _) = 1

main :: IO ()
main = print (42 :: Int)
```

Listing 6: File NonExhaustive/Func6.hs

GADT-Specific

In this section we present the performance tests we have described in Section 5.2.3.

```
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func1 where

import Size0027F (F(..))

func1 :: F a a a -> Int
func1 (MkF1  _ _ _) = 1 -- Int
func1 (MkF14 _ _ _) = 2 -- Char
func1 (MkF27 _ _ _) = 3 -- Bool

main :: IO ()
main = print (42 :: Int)
```

Listing 1: File Exhaustive/Func1.hs

```
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func2 where

import Size0256H (H(..))

func2 :: H a a a a -> Int
func2 (MkH1   _ _ _ _) = 1 -- Int
func2 (MkH86  _ _ _ _) = 2 -- Char
func2 (MkH171 _ _ _ _) = 3 -- Bool
func2 (MkH256 _ _ _ _) = 4 -- String

main :: IO ()
main = print (42 :: Int)
```

Listing 2: File Exhaustive/Func2.hs

```
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

```

```haskell
module Func3 where

import Size0625X (X(..))

func3 :: X a a a a -> Int
func3 (MkX1   _ _ _ _) = 1 -- Int
func3 (MkX157 _ _ _ _) = 2 -- Char
func3 (MkX313 _ _ _ _) = 3 -- Bool
func3 (MkX469 _ _ _ _) = 4 -- String
func3 (MkX625 _ _ _ _) = 5 -- Integer

main :: IO ()
main = print (42 :: Int)
```

Listing 3: File Exhaustive/Func3.hs

```haskell
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func4 where

import Size1296Y (Y(..))

func4 :: Y a a a a -> Int
func4 (MkY1    _ _ _ _) = 1 -- Int
func4 (MkY260  _ _ _ _) = 2 -- Char
func4 (MkY519  _ _ _ _) = 3 -- Bool
func4 (MkY778  _ _ _ _) = 4 -- String
func4 (MkY1037 _ _ _ _) = 5 -- Integer
func4 (MkY1296 _ _ _ _) = 6 -- ()

main :: IO ()
main = print (42 :: Int)
```

Listing 4: File Exhaustive/Func4.hs

```haskell
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func5 where

import Size2041Z (Z(..))

func5 :: Z a a a a -> Int
func5 (MkZ1    _ _ _ _) = 1 -- Int
func5 (MkZ401  _ _ _ _) = 2 -- Char
func5 (MkZ801  _ _ _ _) = 3 -- Bool
func5 (MkZ1201 _ _ _ _) = 4 -- String
func5 (MkZ1601 _ _ _ _) = 5 -- Integer
func5 (MkZ2001 _ _ _ _) = 6 -- ()
func5 (MkZ2401 _ _ _ _) = 7 -- [Int]

main :: IO ()
main = print (42 :: Int)
```

Listing 5: File Exhaustive/Func5.hs

```haskell
{-# OPTIONS_GHC -XScopedTypeVariables -Wall -XGADTs #-}

module Func6 where

import Size3125G (G(..))

```

```
7  func6 :: G a a a a a -> Int
8  func6 (MkG1    _ _ _ _ _) = 1 -- Int
9  func6 (MkG782  _ _ _ _ _) = 2 -- Char
10 func6 (MkG1563 _ _ _ _ _) = 3 -- Bool
11 func6 (MkG2344 _ _ _ _ _) = 4 -- String
12 func6 (MkG3125 _ _ _ _ _) = 5 -- Integer
13
14 main :: IO ()
15 main = print (42 :: Int)
```

Listing 6: File Exhaustive/Func6.hs