KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Implicit Type Conversions

Francisco Gabriel

Thesis submitted for the degree of
Master of Science in Engineering:
Computer Science

Thesis supervisor:
Prof. dr. ir. Tom Schrijvers

Assessors:
Dr. Clement Gautrais
Dr. Amin Timany

Mentor:
Ir. Gert-Jan Bottu

Academic year 2018 – 2019

Preface

This thesis would have been impossible without Gert-Jan Bottu. I must express my deepest gratitude for his ideas, constructive feedback, enthusiasm and dedication. He has truly been a mentor to me and has put (many) more hours into this work than I could ever expect.

I would also like to express my gratitude to my supervisor, Professor Tom Schrijvers, for his contributions, interest and availability, and to George Karachalias for his input and help in order to tackle difficulties that arose.

I must also thank Laura, for her humor, wit and company, as it has been a great pleasure working and living along side her these past two years.

Tenho ainda que agradecer aos meus pais, pelo apoio e por uma genética formidável; aos meus quatro avós pela força que me deram para dar o meu melhor; à minha avó Dulce, em particular, por ter tido a paciência de me ensinar coisas novas todos os dias durante anos; ao Zé e à Isabel, por nenhuma razão em especial, mas para que me mencionem nas suas teses/dissertações/romances/ráio-que-os-parta.

In memory of Daphne Caruana Galizia, Marielle Franco, Jamal Khashoggi and all others murdered for exposing the truth.

In memory of those who drowned (and keep drowning) trying to escape someone else's war; and of those who were bombed before/while trying to escape.

Francisco Gabriel

Contents

Abstract

This text presents TrIC (Transitive Implicit Conversions), a calculus with (user-defined) implicit type conversions (ITC). TrIC was designed as a Haskell language feature, so type inference and compatibility with type classes [25] were taken into account. In general, any Haskell program that type-checks without ITC should not be affected in any way. A further goal is that the language is as clear and simple as possible so that the user is able to easily predict its behavior. To the best of our knowledge, only two mainstream languages already support user-defined implicit type conversions: Scala [16] and C# [6]. TrIC allows transitivity to be used in the implicit type conversions, a feature that is not supported by either of the fore mentioned languages. Other novelties are the local scoping of the conversion axioms and the fact that the user is able to write polymorphic axioms with multiple conditions on other implicit conversions.

In a TrIC program, the programmer is free to introduce conversion axioms (a triple of the type the axiom converts from, the type it converts to and a term capable of performing said conversion) that will, when necessary, be composed and inserted in the source program, in order to translate it into a System F [19] program.

This translation occurs simultaneously to type inference, as is common for Hindley-Milner (HM) based systems: the program is traversed and partially elaborated (a process similar to the elaboration of a HM term into System F) and both equality and the novel conversion constraints are generated (place-holding variables are introduced in the program, each associated with a conversion constraint in a bijective mapping).

Conversion constraints are generated while typing applications and CASE expressions. They relax HM's equality constraints: it is no longer required that the types present in an application match literally, but rather that it is possible to implicitly convert between the appropriate types, and similarly for CASE expressions. If it is possible to entail all the generated constraints, the program is accepted, a substitution mapping type variables to ground types constructed and applied to it in order to obtain the final type; and converting expressions substitute the place-holders in order to obtain the elaborated expression. Solving the conversion constraints involves a novel graph-based approach. An extension of TrIC with support for type classes (TrICx) is also presented.

Chapter 1

Introduction

We build tools to make our life easier. The goal is to minimize the work needed to solve some problem, provided that we retain full control on how our problem is being solved. Washing machines wash our dirty laundry but we choose the temperature, we do not let them guess it. However, if we were sure that the machines could infer the right temperature by the laundry it has been fed, we would be glad that it is now easier to have clean clothes.

Programming languages are tools. They have been evolving over the past decades allowing us to solve more problems and/or solve them in a better way. "A better way" is not restricted to the final solution alone: the process of writing the code should also be taken into account. Languages with static typing allow the detection of erroneous code sooner, at compile-time. However, static typing can lead to a reduction in the expressivity of a language and to the need of explicitly writing the types in the programs.

The Hindley-Milner type system [3, 14, 7] is a perfect example of a breakthrough in the programming language research. While being a powerful enough language in which a great variety of programs can be expressed, Hindley-Milner improves on System F by eliminating a burden for the user: the need for explicitly annotating the programs (at the cost of restricting System F's expressiveness). In turn, System F improves on the less sophisticated Simply Typed Lambda Calculus since it enables code re-use by allowing functions to be defined once and applicable to arguments of any type.

Implicit programming mechanisms "infer values by a type-directed resolution process, making programs more compact and easier to read"([20]). Implicit programming features have surfaced in a number of languages: from Haskell [25] and Coq's [21] type classes, to Agda's implicit arguments [2] and to Rust's traits [15], to name a few.

Scala [16] is a widely used programming language that enables the use of implicit parameters and implicit conversions. This means that a parameter can be omitted from a function and Scala will fetch it (under certain conditions); and that functions can be applied to arguments of a different type, as long as there is an implicit conversion from the type of the provided argument to the type of the expected

argument. As an example, if an implicit conversion from the type Euro to the type Dollar is in scope, a program that expects Dollars will behave seamlessly in cases where an argument of type Euro is provided, *i.e.*, as if its argument was the corresponding amount in Dollars.

Nonetheless, there is room for improvement: as the technology stands now, given an implicit conversion from Euro to Dollar and one from Dollar to Pounds, Scala's implicit conversions will not be able to take the two steps and implicitly convert from Euro to Pound. This text presents TrIC, a language that supports transitivity in its implicit type conversions, thus allowing the programmer to avoid writing tedious conversions and thus also possibly avoiding some annoying bugs.

Furthermore, TrIC supports novel features regarding implicit conversions, from local scoping to multiple constrained conversions. User-friendliness is taken into account so that this extension behaves in a way the user can easily predict. This is not currently the case: Chapter 4 provides an example of implicit conversions leading the execution of a Scala program down unintended paths.

The structure of this text is as follows:

- Chapters 2 and 3 provide the necessary background for the rest of this thesis;
- Chapter 4 introduces implicit type conversions (ITC), presents the state of the art discusses the desirable aspects of a language with support for ITC;
- Chapter 5 presents TrIC, a small core language with user-defined ITC, and the generation of constraints required for compiling TrIC programs;
- Chapter 6 explains how the aforementioned constraints are entailed and how the program is compiled;
- In Chapter 7, a version of TrIC with support for type classes (TrICx) is studied.
- Finally, Chapter 8 discusses the related work and Chapter 9 concludes.

This thesis resulted in the extension of the KU Leuven Haskell compiler (KHC) to support the features discusses in the text. The implementation of the extended KHC is available at https://github.com/mesqg/thesis. Furthermore, this text has been summarized in an extended abstract (Appendix C) that has been accepted to appear in the International Conference on Functional Programming Student Research Competition 2019.

Chapter 2

STLC and System F

Programming languages should have desirable properties programmers can take advantage of when using them. But before relying on those properties, they should be rigorously proved. For that, it is very useful to have a small core language (to minimize what needs to be taken into account when proving such results), capable of expressing the full power of the language. This chapter is based on Chapters 5, 6, 9 and 23 of Types and Programming Languages [17].

Section 2.1 introduces the (untyped) Lambda-Calculus and Section 2.2 discusses its typed version. In Section 2.3 System F, a language with support for parametric polymorphism, is presented and the chapter concludes with some popular extensions to System F (Section 2.4).

2.1 The Untyped Lambda-Calculus

The simple or untyped lambda-calculus [1], introduced in the 1920s by Alonzo Church, is one such (very) small core language, based on the essential notion of functions. Functions are expressions abstracted over some variables; and can be given arguments to replace these variables. The application of arguments to a function yields a new expression which may (or may not) then be further evaluated.

In the pure lambda-calculus, functions are called abstractions (even though throughout this thesis *function* and *abstraction* are used interchangeably). Whereas mathematics or programming languages such as Java use the notation `f(x)={...}` for functions, lambda calculus uses the lambda notation: $\lambda x.\ \{...\}$. For example, the function `square(x)=x*x` translates to $\lambda x.x * x$. Another difference is that while applying a function to a given argument is usually written as `f(argument)`, lambda calculus syntax is $((\lambda x.\ \{...\})\ argument)$. This means that `square(2)` becomes $(\lambda x.x * x)\ 2$. Figure 2.1 presents the syntax of the untyped lambda-calculus, where the vertical bars can be read as "or". Its terms e can only be variables x, abstractions of some term e over a variable x: $\lambda x.e$; or the application of a term to another: $e_1\ e_2$. In $\lambda x.e$, e is sometimes referred to as the body of the abstraction.

Abstractions can be nested: $\lambda x_1.\lambda x_2.x_1$ is a valid term that, when applied one argument and then a second, returns the first. However, had it been written as

$e ::= x \mid \lambda x.e \mid e_1\ e_2$ *terms*

Figure 2.1: Syntax for the untyped lambda-calculus

$e ::= x \mid \lambda x : \tau .e \mid e_1\ e_2 \mid \{0, 1, ...\}$ *terms*
$v ::= \lambda x : \tau.e \mid \{0, 1, ...\}$ *values*
$\tau ::= \tau \to \tau \mid Nat$ *types*
$\Gamma ::= \bullet \mid \Gamma, x : \tau$ *environment*

Figure 2.2: STLC's syntax (extended with Naturals)

$\lambda x.\lambda x.x$, it would behave differently (it is effectively another term) and return the second argument. The variable x in the inner abstraction is said to be shadowing the x in the outer. This is easily solved with De Bruijn's indices by assigning numbers, that reflect to which λ they refer, to the variables. The term $\lambda x_1.\lambda x_2.x_1$ would translate to $\lambda.\lambda.1$. You can think of this term as $\lambda 1.\lambda 0.1$. More information on De Bruijn's indices can be found in chapter 6 of Types and Programming Languages[17]. This notion that the names of the variables are irrelevant for the meaning of the term is called α equivalence.

In case the calculus is enriched with naturals and addition, terms such as "$(\lambda x.x + 1)\, 15$" (applying 15 to the successor function) or more convoluted terms like $((\lambda x_1.\lambda x_2.3 + (x_1\ x_2)\ )\, (\lambda x_3.x_3 + 1))\, 15$ can be expressed. The latter would then simplify to $3 + (15 + 1)$. Despite being Turing complete (which in essence means that it can emulate any programming language), its simple syntax and evaluation rules also allow for meaningless terms (for which no evaluation rules exist, and thus get stuck while evaluating) such as $x_1\ x_2$ or $x_1\ (\lambda x_2.5)$.

2.2 Simply Typed Lambda Calculus

In order to prevent terms from getting *stucked* during evaluation, types were introduced, resulting in the simply typed lambda-calculus (STLC). Types ensure that *well-typed* terms share some highly desirable properties. Erroneous code will not be well-typed (although well-typedness does not guarantee that the program will behave as intended by the programmer) and consequently certain kind of errors are shifted from runtime to compile time.

2.2.1 Syntax

The syntax of the STLC (extended with natural numbers) is illustrated in Figure 2.2, with new syntax highlighted in grey. There is a novel notion: a value; values are special terms that are maximally evaluated. In the pure STLC, only abstractions are defined to be values. Extensions of the STLC introduce new syntax, some of which may be defined to be values as well.

Abstractions now require the type of its argument to be explicitly given (in an annotation immediately after the variable). Having this requisite qualifies a language as explicitly typed.

To make STLC interesting, some base type need to be predefined. Otherwise, no term would be well-typed (as will be explained shortly). As such, in this section, the pure STLC is extended with natural numbers. This extension defines natural numbers to be values too (which implies they are also terms).

2.2.2 Typing

Types reflect the fundamental difference between abstractions (a mathematical function) and other terms. Abstractions are given a *function type* ($\tau_1 \to \tau_2$). This type informs about the type of the argument the abstraction can be applied to (τ_1) and about the resulting type of that operation, τ_2. In the current extension, we have added the *base type Nat* and decided to assign it to natural numbers. A *base type* is a type that is not constructed from other types. This is opposed to *function types*, which are constructed from two types: the type of the argument and the type of the result. If there are no base types, there are no types at all and thus no term can ever have a type.

Well-typed terms are defined as being the terms we can assign some type to (the remaining terms are known as *ill-typed*).

The typing rules for STLC (the combination of the pure STLC's typing rules and the ones we introduced for natural numbers) are shown in Figure 2.3. They reflect our choice to assign natural numbers the type *Nat*, implying that natural numbers are well-typed terms. Infinitely many other terms are also well-typed, as will be clear once we take a closer look at these rules.

Before diving into the typing rules, it is helpful to keep in mind the behavior of functions: applying a function that takes elements of a set X to elements of Y, to an argument from X results in an element of Y. Conversely, if by assuming the argument given to the function f is in some set X it will surely return an element of Y, then f is a function from X to Y.

As such, we expect the type of a function that receives a natural and adds 3 to it ($\lambda x : Nat.x + 3$) to be $Nat \to Nat$. Note that a given abstraction could have several types if its variable was not explicitly annotated. Consider for example $\lambda x :?.3$; it has type $? \to Nat$, for any type $?$.

Likewise, the type of the identity function applied to a natural number ($(\lambda x : Nat.x)\, 4$) is expected to be *Nat*.

Because abstractions can be nested, all assumptions made about the types of variables must be tracked. These are kept in what is known as a *typing environment* (Γ is used to denote such environments). Therefore, typing is a relation in the product of environments, terms and types. The assumption that x has type τ is stored as $x : \tau$; x is said to be *bound* to τ. The relation "$\Gamma \vdash_{tm} e : \tau$" should be read as "under the typing environment Γ, the term e has type τ.

The T-Var rule simply states that, if some variable x is bound to some given type τ in an environment Γ, then, under Γ, that x has type τ. The rules T-Abs

$\boxed{\Gamma \vdash_{tm} e : \tau}$ STLC Typing

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_{tm} x : \tau}\ \text{T-Var}$$

$$\frac{\Gamma \vdash_{tm} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{tm} e_2 : \tau_1}{\Gamma \vdash_{tm} e_1\ e_2 : \tau_2}\ \text{T-App} \qquad \frac{\Gamma, x : \tau_1 \vdash_{tm} e : \tau_2}{\Gamma \vdash_{tm} \lambda x : \tau_1 . e : \tau_1 \to \tau_2}\ \text{T-Abs}$$

$$\frac{}{\Gamma \vdash_{tm} 0 : Nat}\ \text{T-NatZero} \qquad \frac{\Gamma \vdash_{tm} e : Nat}{\Gamma \vdash_{tm} e + 1 : Nat}\ \text{T-NatSuc}$$

Figure 2.3: Typing rules for STLC

and T-App reflect our intuition about functions. Note that '$\Gamma, x : \tau_1$' in the rule T-Abs represents the environment Γ to which the assumption "x has type a" has been added.

The rules T-NatZero and T-NatSuc inductively type natural numbers: they state, respectively, that zero is a natural and that any term is a natural if its predecessor is.

Typing derivations

A typing derivation is a constructive proof that some term e as a given type τ. It shows how to use the typing rules to infer that conclusion. As we type closed terms (terms in which all variables appear for the first time as the argument of an abstraction), a typing derivation for a STLC's term starts with $\Gamma = \bullet$, *i.e.*, it starts with no assumptions. Note that the typing rules hold for any environment Γ, and in particular to the empty environment $\bullet$.

In STLC, a well-typed term has exactly one type (ill-typed have none), and, because the rules are syntax directed, it can be proven that the typing derivations are unique: there is exactly one way to prove a term has its type.

Example of a typing derivation

Figure 2.4 shows an example of a typing derivation for $((\lambda x : Nat \to Nat.x)\,(\lambda x : Nat.x))\,15)$. This derivation is a proof that the term has type Nat and it holds for any environment Γ, in particular for the empty environment $\bullet$.

The easiest way to understand it is from bottom to top. Because the term is an application the rule T-App is considered. To prove the term has type Nat it must now be proved that, for some τ, $((\lambda x : Nat \to Nat.x)\,(\lambda x : Nat.x))$ has type $\tau \to Nat$ and that 15 as type τ. We stick to this approach and keep simplifying what we need to prove until a trivial case is reached. By trivial cases we mean either a typing rule with no premises (T-NatZero) or a true statement (using T-Var).

Note that while trying to prove that the fore mentioned term has type $Nat \to Nat$, we would reach a point in which no typing rules would apply, so the derivation would be incomplete and thus not a proof.

2.2.3 Evaluation

Regarding the evaluation of a STLC term, it should first be noted that STLC is a language with only variables, abstractions, and applications. Since abstractions are already values they are not evaluated further and it would not make sense to evaluate a variable. It would also not make sense to evaluate an application in which the left hand side is a variable $(x\ e)$ or another application: $((e_1\ e_2)\ e_3)$. In the latter case it could be reasonable to evaluate $e_1\ e_2$ further but not the top-level application.

As such, only applications in which the left side is an abstraction can be further evaluated. A pair of a function and a argument of the correct type is known as a *reducible expression*, or *redex* for short.

Given a redex (a function with type $a_1 \to a_2$ and an argument of type a_1) the evaluation consists in replacing the variable bound by the function everywhere in the body of the function by the argument given. The evaluation of the term then proceeds on another redex.

Happiness is defined in different ways by different people and this applies to evaluating STLC terms as well: there are several ways to define the evaluation rules, *i.e.* there are several ways to decide which redexes get evaluated first and even whether they get evaluated at all. This section presents the evaluation rules of call-by-name: this is similar to Haskell's strategy, except for the fact that Haskell memorizes which terms are the same in order to avoid computing the same multiple times (Haskell's evaluation strategy is known as call-by-need).

The evaluation rules are presented in Figure 2.5. The formula $[x \mapsto e_2]e_1$ represents e_1 in which all occurrences of the variable x were replaced by the term e_2.

Since Haskell is a functional language, it seems fitting that it stops whenever it gets a function: evaluation stops when we get to a top-level abstraction. In other words, redexes inside the body of an abstraction are not evaluated, because none of the evaluation rules can then be applied. The rule E-App states that it is possible to further evaluate the left side of an application, the whole application evaluates to the new left side applied to the same right side. In a way, this evaluation strategy looks ahead to figure out what will happen with the argument when it is part of a redex. If it turns out the argument is never used, it is never evaluated. For this reason, call-by-name and Haskell's call-by-need are said to be lazy evaluation strategies.

In most of the traditional programming languages, the arguments must always be evaluated: this is called a strict or eager evaluation strategy. One example of such evaluation strategy is call-by-value. In call-by-value, only the outermost redex for which the right side is already a value can be further evaluated, possibly resulting in useless work.

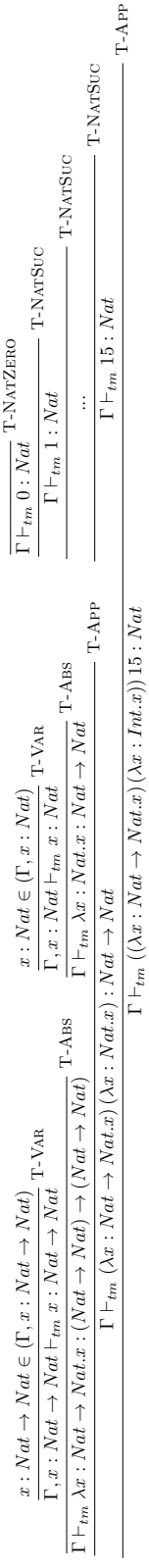


Figure 2.4: Typing Derivation for $((\lambda x : Nat \to Nat.x)\,(\lambda x : Nat.x))\,15)$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \text{E-App} \qquad \frac{}{(\lambda x : \tau.e_1)\ e_2 \to [x \mapsto e_2]e_1}\ \text{E-AppAbs}$$

Figure 2.5: Call by name evaluation rules for STLC

$((\lambda x_1 : (N \to N) \to N.\lambda x_2 : N.\lambda x_3 : N.(x_1\ x_2)\ x_3)(\lambda x_4 : N.\lambda x_5 : N.x_4 + x_5))\ 3$
$\to_{E-AppAbs \text{ followed by } E-App} (\lambda x_2 : N.\lambda x_3 : N.((\lambda x_4 : N.\lambda x_5 : N.x_4 + x_5)\ x_2)\ x_3)\ 3$
$\to_{E-AppAbs} \lambda x_3 : N.((\lambda x_4 : N.\lambda x_5 : N.x_4 + x_5)\ 3)\ x_3$

Figure 2.6: Example of an evaluation of a STLC's term

An example of a call-by-name evaluation of a term

Figure 2.6 (where N represents *Nat* for brevity), illustrates how the evaluation of a non-trivial STLC's term proceeds. Keep in mind that evaluating a term means replacing the top-level expression by some other term.

First, note the left side of the application can be further evaluated: E-AppAbs is applied to $(\lambda x_1 : (N \to N) \to N.\lambda x_2 : N.\lambda x_3 : N.(x_1\ x_2)\ x_3)(\lambda x_4 : N.\lambda x_5 : N.x_4 + x_5)$, resulting in the replacement of x_1 by '$\lambda x_4 : N.\lambda x_5 : N.x_4 + x_5$' everywhere in '$\lambda x_2 : N.\lambda x_3 : N.(x_1\ x_2)\ x_3$'. The rule E-App then allows us to replace our old top-level expression by one in which the left side has been evaluated.

Applying E-AppAbs to what is now a simple redex replaces the variable x_2 everywhere in the body of the abstraction by the number 3. At that point, no evaluation rules apply to the top-level expression, so the evaluation stops.

Note how the redex $(\lambda x_4 : N.\lambda x_5 : N.x_4 + x_5)\ 3$ is not further simplified due to being within a lambda abstraction $(\lambda x_3 : N.((\lambda x_4 : N.\lambda x_5 : N.x_4 + x_5)\ 3)\ x_3)$.

2.2.4 Properties

The assurance that well-typed terms do not get stuck is known as type safety [17]. In order to prove it, it is common to prove *progress* and *preservation*. Progress states that "a term is either a value or it can take a step according to the evaluation rules"; preservation is the property that "if a well-typed term takes a step of evaluation, then the resulting term is also well-typed". Both these theorems hold for the STLC.

Furthermore, STLC's terms are strongly normalizing: "every reduction path starting from a well-typed term is guaranteed to terminate".

2.3 System F

In the pure lambda-calculus it is possible to express the term "$id = \lambda x.x$" which, when applied to a term, returns it. Functions in untyped lambda calculus do not care about their argument: they are defined for any and behave equally for all of them. This behavior is close to what is called, in a slightly more evolved setting where types are present, parametric polymorphism. Parametric polymorphism is

$$
\begin{array}{ll}
e ::= x \mid \lambda x : \sigma^F . e \mid e_1\ e_2 \mid \{0, 1, ...\} \mid \Lambda a.e \mid e\ \sigma^F & \textit{terms} \\
v ::= \lambda x : \sigma^F . e \mid \{0, 1, ...\} \mid \Lambda a.e & \textit{values} \\
\sigma^F ::= a \mid \sigma^F \to \sigma^F \mid Nat \mid \forall a.\sigma^F & \textit{types} \\
\Gamma ::= \bullet \mid \Gamma, x : \sigma^F \mid \Gamma, a & \textit{typing environments}
\end{array}
$$

Figure 2.7: Syntax for System F

defined as occurring "when a function is defined over a range of types, acting in the same way for each type" ([25]).

In STLC, it is not possible to write one single identity function and have it defined over a range of types: because the lambda abstractions are type annotated, a different function is needed for each type. Applying an abstraction $\lambda x : \tau_1$ to a term e of type τ_2 is not well-typed and thus not part of the language. In general, a function is defined for only one type. This lack of parametric polymorphism is a severe limitation of STLC.

System F [19, 4] extends STLC with parametric polymorphism while maintaining type safety.

2.3.1 Syntax

In order to enable functions to apply to terms of multiple types, it is necessary to abstract terms over types. System F supports this behavior in a similar manner to STLC's abstraction over terms. Its syntax is shown is Figure 2.7 and it is an extension of STLC's. In this text, System F types are denoted by σ^F.

For the purpose of abstracting terms over types, it is necessary to add type variables (a) and a language construct that represents the abstraction of e over the type variable a: $\Lambda a.e$. The *id* function can now be re-written as $\Lambda a.\lambda x_1 : a.x_1$. Terms of the form $\Lambda a.e$ are called type abstractions. Because the newly defined *id* is, in essence, still a function, it should also be considered a value. In general, type abstractions are defined to be values too.

Conversely, types τ must be applied to type abstractions in order to have a "normal" function that can be applied to arguments of type τ. This way, the syntax is further extended to include the application of a type to a term. If *Nat* is applied to the newly defined *id* function, the result is a function that has type $Nat \to Nat$. This is completely analogous to term application and is known as type application.

Note that, opposed to term abstractions, which specify the type of the terms that can be applied, type abstractions in System F have no way to restrict "the type of the type" (known as the *kind*) that can be applied.

2.3.2 Typing

Drawing another parallelism with STLC, whereas before the assumptions made about the term variables were stored and it was only possible to reason about variables in the environment, now the type variables encountered must also be considered.

$\boxed{\Gamma \vdash_{tm} e : \sigma^F}$ System F Typing

$$\frac{}{\Gamma \vdash_{tm} 0 : Nat}\ \text{T-NatZero} \qquad \frac{\Gamma \vdash_{tm} e : Nat}{\Gamma \vdash_{tm} e + 1 : Nat}\ \text{T-NatSuc} \qquad \frac{x : \sigma^F \in \Gamma}{\Gamma \vdash_{tm} x : \sigma^F}\ \text{T-Var}$$

$$\frac{\Gamma \vdash_{tm} e_1 : {\sigma^F}_1 \to {\sigma^F}_2 \qquad \Gamma \vdash_{tm} e_2 : {\sigma^F}_1}{\Gamma \vdash_{tm} e_1\ e_2 : {\sigma^F}_2}\ \text{T-App} \qquad \frac{\Gamma, x : {\sigma^F}_1 \vdash_{tm} e : {\sigma^F}_2 \qquad \Gamma \vdash_{ty} {\sigma^F}_1}{\Gamma \vdash_{tm} \lambda x : {\sigma^F}_1 . e : {\sigma^F}_1 \to {\sigma^F}_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash_{tm} e : \forall a.{\sigma^F}_1 \qquad \Gamma \vdash_{ty} {\sigma^F}_2}{\Gamma \vdash_{tm} e\ {\sigma^F}_2 : [a \mapsto {\sigma^F}_2]{\sigma^F}_1}\ \text{T-TApp} \qquad \frac{\Gamma, a \vdash_{tm} e : {\sigma^F}_1 \qquad a \notin \Gamma}{\Gamma \vdash_{tm} \Lambda a.e : \forall a.{\sigma^F}_1}\ \text{T-TAbs}$$

Figure 2.8: Typing rules for System F

The typing environment can now be extended with type variables: "Γ, a". These are introduced to the environment when type-checking type abstractions.

The same way a term can only be well-typed if all its variables are in the environment, a type can only be *well-formed* if all its type variables are in Γ. This is actually the only condition for type well-formedness for this simple system. The rules for type well-formedness are shown in Figure 2.11 (the highlighted rule can safely be ignored for now). The judgment "$\Gamma \vdash_{ty} \tau''$ can be read as "under the typing environment Γ, the type τ is well-formed".

Unsurprisingly, the System F's typing rules extend STLC's. The complete typing rules for System F are shown in Figure 2.8, were the two highlighted rules were added. The T-TApp rule types type application. If a term e has some type of the form $\forall a.\tau_1$, then the application of τ_2 to e has a type that results from substituting the type variable a in τ_1 in by τ_2. The rule T-TAbs is concerned with type abstraction and state that if, by adding the type variable a to the typing environment, term e can be typed as having type τ; then, abstracting e over that type variable has type $\forall a.\tau$.

A parametrically polymorphic function *twice* that, given a function and a term, applies the function to the term and then applies it again on the result of the first computation can be expressed in System F as: $\Lambda a.\lambda x_1 : a \to a.\lambda x_2 : a.x_1(x_1 x_2)$. The type of *twice* is $\forall a.(a \to a) \to a \to a$.

2.3.3 Evaluation

The operational semantics of System F are an extension from STLC's. System F has the same E-App and E-AppAbs rules for term abstraction and application; it also has their type counterpart: rules E-TyApp and E-TyAppAbs. These are completely analogous. All of them are shown in Figure 2.9.

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \text{E-App} \qquad \frac{}{(\lambda x : \tau.e_1)\ e_2 \to [x \mapsto e_2]e_1}\ \text{E-AppAbs}$$

$$\frac{e_1 \to e_1'}{e_1\ \tau \to e_1'\ \tau}\ \text{E-TyApp} \qquad \frac{}{(\Lambda a.e_1)\ \tau \to [a \mapsto \tau]e_1}\ \text{E-TyAppAbs}$$

Figure 2.9: Call by name operational semantics for pure System F

2.4 System F Extended

This section is concerned with extending System F with data constructors, let bindings and case expressions, making it closer to a real world programming language. This will also the the language that TrIC, presented in Chapter 5, compiles to.

A major advantage of this extension is that instead of needing pre-defined base types as in the pure System F, it is now possible to introduce types in the programs through the use of data declarations. Although it is not obvious, datatypes have been shown to be encodable using polymorphism.

2.4.1 Syntax

As can be seen in Figure 2.10, terms can now also be:

- Data constructors, denoted by K. These are functions that, when applied to terms of the correct type, construct a term of the type T explicitly mentioned in the datatype declaration.
- A `let`-expression that binds a (type annotated) variable x to some expression e. In this thesis we have opted for a recursive let: it allows for the expression e to already refer to the variable x.
- A `case`-expression, which allows the programmer to pattern match terms against the data constructor K used to generate them, in order to retrieve information on the arguments provided to K.

This extension of System F introduces the notion of a program: a sequence of declarations and a term. In this setting, declarations can only be datatype declarations. These introduce a new type T (a System F type, where a type variable can be present) as well as data constructors, which are the only way of constructing a term of type T. The typing environment is extended to be able to handle data declarations.

2.4.2 Typing

Declarations are not typed; programs are assigned the type σ^F assigned to their term under the *appropriate environment* Γ.

The declarations present in the program before the term set the *appropriate environment* under which the term is typed. A datatype declaration '`data` $T\ a = \overline{K_i\ \overline{\sigma^F}_{ij}}$'

$$
\begin{array}{lr}
pgm ::= \overline{datadecl}; e & \textit{program} \\
datadecl ::= \texttt{data}\ T\ a = \overline{K_i\ \overline{\sigma^F_{i_j}}} & \textit{datatype} \\
e ::= x \mid \lambda x : \sigma^F.e \mid e_1\ e_2 \mid \forall a.e \mid e\ \sigma^F & \textit{term} \\
\quad \mid K \mid \texttt{let}\ x : \sigma^F = e_1\ \texttt{in}\ e_2 \mid \texttt{case}\ e_1\ \texttt{of}\ \overline{K_i\ \overline{x_{i_j}} \to e_{i_2}} & \\
v ::= \lambda x : \sigma^F.e \mid \Lambda a.e & \textit{value} \\
\sigma^F ::= a \mid \sigma^F \to \sigma^F \mid \forall a.\sigma^F \mid T\ \sigma^F & \textit{type} \\
\Gamma ::= \bullet \mid \Gamma, x : \sigma^F \mid \Gamma, a \mid \Gamma, K : \sigma^F \mid \Gamma, T & \textit{environment}
\end{array}
$$

Figure 2.10: Syntax for the extended System F

$\boxed{\Gamma \vdash_{ty} \sigma}$

$$\frac{a \in \Gamma}{\Gamma \vdash_{ty} a}\ \textsc{Ty-Var} \qquad \frac{T \in \Gamma}{\Gamma \vdash_{ty} T}\ \textsc{Ty-Cons}$$

$$\frac{\Gamma \vdash_{ty} \tau_1 \qquad \Gamma \vdash_{ty} \tau_2}{\Gamma \vdash_{ty} \tau_1 \to \tau_2}\ \textsc{Ty-Arr} \qquad \frac{\Gamma, a \vdash_{ty} \sigma \qquad a \notin \Gamma}{\Gamma \vdash_{ty} \forall a.\sigma}\ \textsc{Ty-All}$$

Figure 2.11: Type well-formedness

adds both the type constructor T and the data constructors K_i bound to their type to the typing environment. To construct a term of type $T\ {\sigma^F}_1$, first the type ${\sigma^F}_1$ and then terms of the corresponding types $[a \mapsto {\sigma^F}_1]\overline{{\sigma^F}_{i_j}}$ need to be applied to some K_i.

There are two new typing rules: T-Let and T-Case. The former illustrates the recursive nature of the '$\texttt{let}\ x : {\sigma^F}_1 = e_1\ \texttt{in}\ e_2$' terms: both e_1 and e_2 are typed in the extended environment in which x is assumed to have type ${\sigma^F}_1$. This recursion allows functions as the one in Example 1 to be expressed in the language.

As for the T-Case, it states that a case expression '$\texttt{case}\ e_1\ \texttt{of}\ \overline{K_i\ \overline{x_{i_j}} \to e_{i_2}}$' has a type ${\sigma^F}_2$ if all the branches (e_{i_2}) have type ${\sigma^F}_2$ under the extended environment where the variables $\overline{x_{i_j}}$ are bound the corresponding types $\overline{{\sigma^F}_{i_j}}$, where the type variable a has been substituted in conformity with the type ($T\ {\sigma^F}_1$) of e_1: $[a \mapsto {\sigma^F}_1]$. Both of these typing rules are highlighted in Figure 2.12.

2.4.3 Properties

System F achieves what it sets out to be: a type-safe language (well-typed terms either evaluate to a value or evaluate forever) which, by allowing parametric polymorphism, eliminates unpleasant STLC's restrictions.

It would be convenient to drop System F's type annotations. However, that would make typing undecidable for System F's terms.

$$\frac{(x : \sigma^F) \in \Gamma}{\Gamma \vdash_{tm} x : \sigma^F}\ \text{T-Var} \qquad \frac{(K : \sigma^F) \in \Gamma}{\Gamma \vdash_{tm} K : \sigma^F}\ \text{T-Con}$$

$$\frac{\Gamma \vdash_{tm} e_1 : {\sigma^F}_1 \to {\sigma^F}_2 \qquad \Gamma \vdash_{tm} e_2 : {\sigma^F}_1}{\Gamma \vdash_{tm} e_1\ e_2 : {\sigma^F}_2}\ \text{T-App} \qquad \frac{\Gamma, x : {\sigma^F}_1 \vdash_{tm} e : {\sigma^F}_2 \qquad \Gamma \vdash_{ty} {\sigma^F}_1}{\Gamma \vdash_{tm} \lambda x : {\sigma^F}_1 . e : {\sigma^F}_1 \to {\sigma^F}_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash_{tm} e : \forall a.{\sigma^F}_1 \qquad \Gamma \vdash_{ty} {\sigma^F}_2}{\Gamma \vdash_{tm} e\ {\sigma^F}_2 : [a \mapsto {\sigma^F}_2]{\sigma^F}_1}\ \text{T-TApp} \qquad \frac{\Gamma, a \vdash_{tm} e : {\sigma^F}_1 \qquad a \notin \Gamma}{\Gamma \vdash_{tm} \Lambda a.e : \forall a.{\sigma^F}_1}\ \text{T-TAbs}$$

$$\frac{\begin{array}{c}\Gamma, x : {\sigma^F}_1 \vdash_{tm} e_1 : {\sigma^F}_1 \\ \Gamma, x : {\sigma^F}_1 \vdash_{ty} e_2 : {\sigma^F}_2 \qquad \Gamma \vdash_{ty} {\sigma^F}_1 \qquad x \notin dom(\Gamma)\end{array}}{\Gamma \vdash_{tm} (\texttt{let}\ \ x : {\sigma^F}_1 = e\ \ \texttt{in}\ \ e_2) : {\sigma^F}_2}\ \text{T-Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{tm} e_1 : T\ \sigma_1^F \qquad \overline{(K_i : \forall a.\overline{\sigma_{i_j}^F} \to T\ a) \in \Gamma} \\ \Gamma, \overline{x_{i_j} : [a \mapsto \sigma_1^F]\sigma_{i_j}^F} \vdash_{ty} e_{i_2} : \sigma_2^F \qquad x \notin dom(\Gamma)\end{array}}{\Gamma \vdash_{tm} (\texttt{case}\ \ e_1\ \ \texttt{of}\ \ \overline{K_i\ \overline{x_{i_j}} \to e_{i_2}}) : \sigma_2^F}\ \text{T-Case}$$

Figure 2.12: Typing rules for the extended System F's terms

A definition is said to be impredicative if "it envolves a quantifier whose domain includes the very thing being defined" [17]. System F is said to be impredicative, because of its definition for types: a type variable a in the type $\forall a.a \to a$ can refer to any System F type, including types that contain type abstraction (like $\forall a.a \to a$ itself). The identity function, for instance, has type $\forall a.a \to a$. Note that it is possible to apply any type to the identity function: $id\ (\forall a.a \to a)$ is a perfectly valid System F term, with type $(\forall a.a \to a) \to (\forall a.a \to a)$. In cases like this, where the universal operator ($\forall$) is in a nested position of a more complex type σ^F, σ^F is known as a higher-rank type.

Without type annotations the types of System F's expressions would have to be inferred. However, it has been proven that there is no algorithm capable of, in general, inferring the type of a language with higher-rank polymorphic types [10]. In other words, type inference is undecidable for System F. Further, under the call-by-name operational semantics, System F's terms are not strongly normalizing (although they are strongly normalizing under full beta reduction [17]).

2.4.4 An example

Assuming we have declared the datatype *List* with two data constructors, [ ] and *Cons*, it is now possible to write the function foldlr as shown in Example 1.

This is a well-known function in the context of functional programming. In languages with type inference, like Haskell, the type abstractions would be absent. Suppose the datatype declaration for the type constructor List to be:
`data` $List\ a = [\,]\ a \mid Cons\ a\ (List\ a)$

According to the typing rules, $foldr$ would then be typed as:
$\forall a_1.\forall a_2.((a_1 \to a_2 \to a_2) \to (a_2 \to (List\ a_1 \to a_2)))$.

$$
\begin{aligned}
foldr = \Lambda a_1.\Lambda a_2.\lambda x_1 : a_1 \to a_2 \to a_2.\lambda x_2 : a_2.\lambda x_3 : List\, a_1.\\
\quad \texttt{case}\ x_3\ \texttt{of}\\
\quad\quad [\,] \qquad\quad \to x_2\\
\quad\quad Cons\ y\ z \to x_1\ y\ (foldr\ a_1\ a_2\ x_1\ x_2\ b)
\end{aligned}
$$

Example 1: Function $foldr$ in System F

Chapter 3

Hindley-Milner

This chapter introduces the Hindley-Milner type system (HM) [3], a predicative restriction on System F. This type system omits type annotations and has no explicit abstraction of terms over types nor application of types to terms. Nonetheless, there is an algorithm that is capable of inferring the type of HM's expressions.

Any HM expression can be translated to System F but the converse does not hold, *i.e.*, HM is not as expressive as System F. However, it is expressive enough to support parametric polymorphism.

Section 3.1 introduces HM's syntax, Section 3.2 its typing rules, Section 3.3 presents HM's type inference mechanism, Section 3.4 its operational semantics and, finally, Section 3.5 is concerned with the elaboration of HM programs to System F.

3.1 HM syntax

The syntax for HM is shown in Figure 3.1, including the counterparts of the extensions presented in 2.4. It is worth mentioning that the *let* terms are the only language construct that enables polymorphic polymorphism. Overall, the syntax is simpler than System F's: there are no type annotations, explicit types, type abstraction nor type application.

What is new in HM is the further specialization of types. Types can now be monotypes or polytypes (also known as type schemes). Monotypes are either type variables (type variables only ceased to be explicit in the syntax of the terms, they are still needed), function types or base types. Polytypes are either a monotype or a type abstraction. This distinction is used to restrict type expressiveness (when compared to System F) by placing all the "$\forall$'s" upfront, in order to achieve decidable type inference. Regarding the environment, it is same as the one for System F.

3.2 Typing

A HM type is well-formed "$\Gamma \vdash_{ty} \sigma$" if all its type variables are in the typing environment as the type constructors are in scope, similarly to System F's type

$$
\begin{array}{lr}
pgm ::= \overline{data\ decl};\ e & programs \\
data\ decl ::= data\ T\ a = \overline{K_i\ \overline{\tau_{i_j}}} & data\ declarations \\
e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid K \mid case\ e_1\ of\ \overline{(K_i\ \overline{x_{i_j}}) \to e_2} \mid let\ x\ =\ e_1\ in\ e_2 & terms \\
v ::= \lambda x.e & values \\
\tau ::= a \mid \tau \to \tau \mid T & monotypes \\
\sigma ::= \tau \mid \forall a.\sigma & polytypes \\
\Gamma ::= \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, K : \sigma \mid \Gamma, T & environments
\end{array}
$$

Figure 3.1: HM's syntax

well-formedness (Figure 2.11). The typing rules (shown in Figure 3.2) are very similar to System F's, except that:

- the rules T-Abs, T-App, T-Let and T-Case can only assign monotypes to terms.
- Unlike their System F's counterparts, rules T-Tabs, T-Tapp have the same term e in the premises and in the conclusion. Type abstraction and type application are transparent to the syntax of the term.
- the typing rules now assign polytypes to terms, instead of System F's more general types. These typing rules effectively enforce that all the "$\forall$"'s are placed upfront. Note that term variables and data constructors assigned a universally quantified type will first have to go through rule T-TApp before any of the rules T-Abs, T-App, T-Let or T-Case is applicable.

3.3 Type inference

Type inference means that the types of the expressions will be automatically inferred by the compiler. Algorithm W (to be described in this section) is responsible for this feature in HM. It infers the types in two separate steps: first it generates a type (with unresolved type variables) and a set of constraints of the form $\tau_1 \sim \tau_2$, specifying that the types τ_1 and τ_2 must be unifiable; afterwards, the constraints are solved, a substitution is generated and applied to the previously inferred types, instantiating the necessary type variables.

3.3.1 Constraint Generation

The first part of algorithm W is concerned with the generation of equality constraints and is shown in Figure 3.4. The highlighted part is important for the translation of HM terms to System F (Section 3.5), but can be safely ignored for now. The relation "$\Gamma \vdash_{tm} e : \tau;\ E$", which should be read as "under the environment Γ, e gets assigned type τ assuming the equality constraints E are satisfied", reflects the behavior of the first part of the algorithm.

$\boxed{\Gamma \vdash_{tm} e : \sigma}$

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash_{tm} x : \sigma}\ \text{T-Var} \qquad \frac{(K : \sigma) \in \Gamma}{\Gamma \vdash_{tm} K : \sigma}\ \text{T-Con}$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma, x : \tau_1 \vdash_{tm} e : \tau_2 \qquad \Gamma \vdash_{ty} \tau_1}{\Gamma \vdash_{tm} \lambda x : \tau_1 . e : \tau_1 \to \tau_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash_{tm} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{tm} e_2 : \tau_1}{\Gamma \vdash_{tm} (e_1\ e_2) : \tau_2}\ \text{T-App}$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma, x : \sigma \vdash_{tm} e_1 : \sigma \qquad \Gamma, x : \sigma \vdash_{ty} e_2 : \tau}{\Gamma \vdash_{tm} (let\ x : \sigma = e_1\ in\ e_2) : \tau}\ \text{T-Let}$$

$$\frac{I; \Gamma \vdash_{tm} e_1 : T\ \tau_1 \qquad \dfrac{\overline{(K_i : \forall a . \overline{\tau_{i_j}} \to T\ a) \in \Gamma}}{I; \Gamma, \overline{x_{i_j} : [a \mapsto \tau_1]\tau_{i_j}} \vdash_{tm} e_i : \tau_2} \qquad \overline{x_{i_j}} \notin dom(\Gamma)}{I; \Gamma \vdash_{tm} case\ e_1\ of\ \overline{(K_i\ \overline{x_{i_j}}) \to e_i} : \tau_2}\ \text{T-Case}$$

$$\frac{\Gamma, a \vdash_{tm} e : \sigma \qquad a \notin \Gamma}{\Gamma \vdash_{tm} e : \forall a . \sigma}\ \text{T-TAbs} \qquad \frac{\Gamma \vdash_{tm} e : \forall a . \sigma \qquad \Gamma \vdash_{ty} \tau}{\Gamma \vdash_{tm} e : [a \mapsto \tau]\sigma}\ \text{T-TApp}$$

Figure 3.2: HM's typing rules

The T-Var rule looks up the variable x in the typing environment. If the variable is bound to a polytype $\forall \overline{a} . \tau$, fresh type variables $\overline{b}$ are created to substitute the variables $\overline{a}$ in τ. This assures that different occurrences of x will be assigned different types. If this were not the case, the step of solving the equality constraints could fail due to the unification of types that do not need to be equal. Suppose the polymorphic function $id = \lambda x . x$ is applied, in different places, to 3 and to 'c'. The type of the variable x needs to be unified with the type of 3 in one case and with the type of 'c' in the other, but not to both simultaneously (which would be the meaning of the equality constraints if the variables were not refreshed).

As for lambda abstractions, T-Abs, since there is no type annotation on the bound variable, a fresh type variable is used both to type the abstraction and to be bound to the variable for the typing of the body of the abstraction. This type variable will be substituted by the correct type after solving the constraints.

For the case of an application $e_1\ e_2$, T-App first performs the first step of type inference on both its subterms. For $e_1\ e_2$ to be well-typed, the type τ_1 of e_1 is necessarily a function type from the type τ_2 of e_2 to some type unknown for now. As such, the constraint $\tau_1 \sim (\tau_2 \to a)$ is generated, where the new fresh type variable a represents the result type.

$\boxed{\Gamma \vdash_{tm} e : \sigma \,;\, e^F}$ Top-Level Type Inference with Elaboration

$$\frac{\Gamma \vdash_{tm} e : \tau; E \,;\, e^F \qquad \theta = unify(\bullet; E) \qquad \overline{a} = fv(\theta(\tau))}{\Gamma \vdash_{tm} e : \forall \overline{a}.\theta(\tau) \,;\, \Lambda \overline{a}.\theta(e^F)}$$

Figure 3.3: HM top level type inference with elaboration (highlighted)

$\boxed{\Gamma \vdash_{tm} e : \tau; E \,;\, e^F}$ Type Inference with Elaboration

$$\frac{(x : \forall \overline{a}.\tau) \in \Gamma \qquad \overline{b}\ fresh}{\Gamma \vdash_{tm} x : [\overline{a} \mapsto \overline{b}]\tau \,;\, \bullet \,;\, x^F\ \overline{b^F}}\ \text{T-Var} \qquad \frac{(K : \forall a.\tau) \in \Gamma \qquad b\ fresh}{\Gamma \vdash_{tm} K : [a \mapsto b]\tau \,;\, \bullet \,;\, K^F\ b^F}\ \text{T-Cons}$$

$$\frac{a\ fresh \qquad \Gamma, x : a \vdash_{tm} e : \tau_1 \,;\, E \,;\, e^F}{\Gamma \vdash_{tm} \lambda x.e : a \,;\, E \,;\, \lambda(x^F : a^F).e^F}\ \text{T-Abs}$$

$$\frac{a\ fresh \qquad \Gamma \vdash_{tm} e_1 : \tau_1 \,;\, E_1 \,;\, e_1^F \qquad \Gamma \vdash_{tm} e_2 : \tau_2 \,;\, E_2 \,;\, e_2^F}{\Gamma \vdash_{tm} e_1\ e_2 : a \,;\, E_1, E_2, \tau_1 \sim (\tau_2 \to a) \,;\, e_1^F\ e_2^F}\ \text{T-App}$$

$$\frac{fresh\ a \qquad \Gamma, x : \tau_1 \vdash_{tm} e_1 : \tau_1; E_1 \,;\, e_1^F \qquad \Gamma, x : \tau_1 \vdash_{tm} e_2 : \tau_2; E_2 \,;\, e_2^F}{\Gamma \vdash_{tm} let\ x = e_1\ in\ e_2 : \tau_2; E_1, E_2 \,;\, let\ x^F : \tau_1^F = e_1^F\ in\ e_2^F}\ \text{T-Let}$$

$$\frac{\begin{array}{c}\overline{(K_i : \forall a.\overline{\tau_{i_j}} \to T\ a) \in \Gamma} \\ \Gamma \vdash_{tm} e_1 : \tau_1; E_1\ e_1^F \qquad \overline{\Gamma, \overline{x_{i_j} : [a \mapsto b]\tau_{i_j}} \vdash_{tm} e_{2i} : \tau_i; E_i \,;\, e_{2i}^F} \\ fresh\ b,\ d \qquad E = E_1, \overline{E_i}, \tau_1 \sim T\ b, \overline{\tau_i \sim d}\end{array}}{\Gamma \vdash_{tm} case\ e_1\ of\ \overline{(K_i\ \overline{x_{i_j}}) \to e_{2i}} : d; E \,;\, case\ e_1^F\ of\ \overline{(K_i^F\ \overline{x_{i_j}^F}) \to e_{2i}^F}}\ \text{T-Case}$$

Figure 3.4: HM type inference with elaboration (highlighted)

Let expressions simply accumulate the equality constraints of its subterms. Finally, *case* expressions constrain its scrutinee to be of the form $T\ \tau$, for the appropriate type constructor T with respect to the data constructors K_i on the patterns; and all its alternatives e_i to have the same type.

$\boxed{unify(\overline{a}; E) = \theta}$ Type Unification Algorithm

$$unify(\overline{a}; \bullet) = \bullet \tag{3.1}$$
$$unify(\overline{a}; E, b \sim b) = unify(E) \tag{3.2}$$
$$unify(\overline{a}; E, \tau \sim b) = unify(\overline{a}; E)) \circ [b \mapsto \tau], \text{ if } b \notin (\overline{a} \cup fv(\tau)) \tag{3.3}$$
$$unify(\overline{a}; E, b \sim \tau) = unify(\overline{a}; E)) \circ [b \mapsto \tau], \text{ if } b \notin (\overline{a} \cup fv(\tau)) \tag{3.4}$$
$$unify(\overline{a}; E, (\tau_1 \to \tau_2) \sim (\tau_3 \to \tau_4)) = unify(\overline{a}; E, \tau_1 \sim \tau_3, \tau_2 \sim \tau_4) \tag{3.5}$$

Figure 3.5: Type Unification Algorithm

3.3.2 Constraint Solving

Regarding solving the constraints, Martelli and Montanari's unification algorithm [13] (Figure 3.5) is used to compute a type substitution θ such that all the equality constraints are trivially entailed (*i.e.*, such that the types on both sides of the constraint are literally the same: $\tau \sim \tau$) after applying θ to the set of generated equality constraints. If the term is ill-typed, no substitution exists and the unification algorithm fails (*i.e.* there are still constraints to unify but no rule applies). The first argument of *unify* represents the type variables that must not be substituted during unification (and plays an important role in the TrIC and TrICx).

The fore mentioned type substitution is recursively constructed by sequentially considering the equality constraints. If both types are the same type variable $a \sim a$ the constraint is already trivially entailed and is disregarded. If only one of the types is a type variable a, then θ must map a to the other type τ. However, a can not be a free variable of τ; otherwise, θ would cause τ to be an infinite type (*eg.* $a \sim [a]$ would lead to substituting a by the infinite type $[...[a]...]$).

Equality constraints between two function types are destructed into two simpler constraints (one for the argument type and another for the result type) that should be solved sequentially. These will not become independent as the type substitution resulting from unifying the first will be applied to the second before its unification. Otherwise, the equality constraint $a \to a \sim B \to C$ would be entailable for any predefined types B and C, which is clearly undesirable.

3.4 Operational Semantics

The evaluation rules for HM are completely identical to the ones of STLC (Section 2.2.3).

3.5 Elaboration

The elaboration of a HM to System F is presented highlighted in Figure 3.4. Notice the rule T-Var: because System F has explicit type abstraction (and application), a variable that is bound to a polytype is applied type variables so that these are later substituted by the appropriate types.

For lambda abstractions, its variable needs to be annotated with the correct System F type. Because in a HM abstraction this variable is necessarily assigned a monotype (otherwise the type of the abstraction would not be well formed), and there is bijection between HM monotypes and System F monotypes, the elaboration of the type is not made explicit. The elaboration of an application is simply the application of the elaborations of its subterms. A similar reasoning applies for *let* and *case* expressions.

Chapter 4

Implicit Type Conversions

This chapter starts by introducing implicit type conversions (ITC) and showcasing them with a simple example (Example 4.1). Afterwards, the state of the art is discussed in Section 4.2; in Section 4.3, the problem statement of this thesis is precisely stated and its main ideas clarified. Finally, in Sections 4.4 and 4.5, some immediate consequences of the problem statement are addressed.

4.1 The Basics

This text defines implicit type conversions to be an programming feature that allows the programmer to write conversions and then have them automatically inserted where needed.

The support of implicit conversions requires two basic aspects: a set of expressions intended to be used to convert terms of some type τ_1 to terms of another type τ_2 and a way to figure out where, which and how these expressions should be inserted in the source program.

A *conversion axiom* consists of one such converting expression e, the type (τ_s) e converts from and the type (τ_t) e converts to. Even though τ_s and τ_t could be inferred in a language with type inference (as the one presented in chapter 5), these inferred types could be more general than intended by the programmer and thus applicable in undesired circumstances. As such, they should be explicitly written by the user. The set of conversion axioms is know as the *implicit environment*.

As for the "where, which and how" to use these conversing expressions, the main idea is to perform type inference and, upon encountering an inconsistency, flag it and associate it with a *conversion constraint*, stating the type τ_1 inferred and the type τ_2 needed.

From this point onwards, "$\rightsquigarrow$" will be used to denote the implicit conversion from the type τ_s on its left to the type τ_t on its right, as in $\tau_s \rightsquigarrow \tau_t$. In this chapter, to distinguish between the possibility of converting from τ_s to τ_t (due to the existent conversion axioms) and the necessity of a conversion from τ_s to τ_t (a conversion constraint), these arrows will be indexed by, respectively, "ax" ($\tau_s \rightsquigarrow_{ax} \tau_t$) and "ct" ($\tau_s \rightsquigarrow_{ct} \tau_t$).

As a running example, suppose Maria sells olives in Lisbon. She wrote some accounting software and has defined a type EUR (for Euro). She then decides to export her olives to Switzerland, so she defines a type CHF (for Swiss Franc). If the programming language supports ITC, instead of duplicating her code or manually introducing a converting function every time a value in CHF is introduced to the system, she will only need to define an implicit conversion from CFH to EUR once in order to have the software running seamlessly.

As a more concrete example, consider Maria's updated program to compute the taxes she has to pay:

$$
\begin{aligned}
&\mathit{data}\ \text{EUR} = K_{\text{EUR}}\ \mathit{Float}\\
&\mathit{data}\ \text{CHF} = K_{\text{CHF}}\ \mathit{Float}\\
&...\\
&\mathit{conversion\ axiom} :: \mathit{CHF} \rightsquigarrow_{ax} \mathit{EUR} = \lambda x.\mathit{case}\ x\ \mathit{of}\ (K_{\text{CHF}}\ y) \rightarrow (K_{\text{EUR}}\ 0.9 * y)\\
&\mathit{taxes} :: \mathit{EUR} \rightarrow \mathit{EUR} = ...\\
&...\\
&\mathit{taxes}(K_{\mathit{CHF}}\ 30.0)
\end{aligned}
$$

The conversion constraint in this case is CHF $\rightsquigarrow_{ct}$ EUR, because *taxes* expects an argument of type EUR but gets one of type CHF. Because of the conversion axiom, the final expression will be elaborated to $\mathit{taxes}((\lambda x.\mathit{case}\ x\ \mathit{of}\ (K_{\text{CHF}}\ y) \rightarrow (K_{\text{EUR}}\ 0.9 * y))\ (K_{\mathit{CHF}}\ 30.0))$, which can then be evaluated to a EUR value.

4.2 State of the Art

In the literature, implicit type conversions can also refer to a widely used language feature that implicitly converts between some of the language's fundamental types. In C or C++, for example, if an arithmetic expression has operands of two different types (suppose we try to sum a *long double* and a *float*) the compiler will implicitly convert the type with lowest rank (*float*) to the type of highest rank (*long double*) and use the '+' operator as defined for type long double.

However, as apparent from the last section, this thesis is concerned with user-defined implicit type conversions. To the best of our knowledge, only two mainstream languages already support user-defined implicit type conversions: Scala and C#.

4.2.1 Scala

Scala [16] is a widely used language with support for Implicit Type Conversions. In Scala, a programmer is free to mark declarations with the *implicit* keyword. These marked declarations can later be used to fix type errors.

To be available for use, an implicit conversion must be in scope under a single identifier or be defined in the "companion object of the source or expected target types of the conversion".

```
implicit def impl_conv (x:Char):Int = 3
def function(x:Char):String =
x match {
  case 3 => ``Int 3''
  case 97 => ``Int 97''
  case 'a' => ``Char 'a'''
  case _ => ``Other''
}
println(function('a'))

> Int 97
```

Figure 4.1: Execution of a Scala program

Scala allows for polymorphic methods in general, including in implicit conversions. Furthermore, the use of Scala's implicit parameters enables writing implicit type conversions only applicable when another implicit conversion is in scope.

Scala will not try to compose conversions in order to fix type errors. One could be tempted to bypass this restriction by using the fore mentioned features in order to write a rule that would introduce transitivity. The idea would be to write this implicit as "it is possible to convert type variable a to type variable c if there are two implicit parameters: one able to convert from a to another type variable b and a second capable of converting from b to c". However, this is not supported in Scala, since only one implicit parameter can be used.

According to its documentation, Scala will not introduce conversions unless necessary, enforces a "non-ambiguity rule" and "overloading resolution is applied if there are several possible candidates" for implicit conversions. As will be shown, this overloading resolution can cause the language to become hard to predict from the user's point of view. This is aggravated by the fact that Scala pre-defines several high-priority implicit type conversions which will shadow the user's defined conversions.

Figure 4.1 presents a Scala program that showcases the difficult nature of understanding the use of Scala's implicit type conversions and contradicts Scala's claim that it will not attempt to insert implicit conversions if there is no need for them.

Note that Scala ignores the user-defined conversion and then applies a conversion predefined in the language (there is the option of not loading any predefined conversions), and that there was no need for any conversion to be used in the first place, since *function* is applied to an argument of the expected type.

4.2.2 C#

C# [6] is another well-known language that supports user-defined implicit type conversions. Conversions are declared on "classes or structs so that classes or structs can be converted to and/or from other classes or structs, or basic types. Conversions are defined like operators and are named for the type to which they convert. Either

the type of the argument to be converted, or the type of the result of the conversion, but not both, must be the containing type."

As in Scala, only one user-defined conversion can be used at a time. Under some conditions it is possible to write conversions with generic types. If there exists a predefined conversion for the same types of the user-defined conversion, the latter will be ignored.

4.3 Problem Statement

The aim of this thesis is to design a calculus with (user-defined) implicit type conversions and provide an implementation for this feature in the KU Leuven Haskell Compiler (KHC). This calculus improves on the state of the art by allowing transitivity to be used in the implicit type conversions. Other novelties are the local scoping of the conversion axioms and the fact that the user is able to write polymorphic axioms with possibly multiple conditions on the implicit environment. Since it is designed as a Haskell language feature, type inference and compatibility with type classes are central concerns. In general, any Haskell program that type-checks without ITC should not be affected in any way. Further, the user should be able to easily predict the behavior of the calculus and the syntax of the language should be clear and as simple as possible.

This section elaborates on the advanced features supported by TrIC, the language defined in Chapter 5.

4.3.1 Transitivity

Supporting transitivity in implicit type conversions means that the language automatically tries to compose the available conversion axioms in order to convert a term to the appropriate type, instead of being limited to applying one user-provided conversion axiom.

Consider the following example: Maria decides to also sell olives in Copenhagen. She will also need to deal with Danish Krones (DKK). For reasons we must not disclose, she defined functions that expect CHF that are useful in Denmark too. Nonetheless, much of the code is written for arguments to be of type EUR, so she also needs a conversion from DKK to EUR. Having previously defined a conversion from CHF to EUR, she now needs only to define an implicit conversion from DKK to CHF and gets an implicit conversion from DKK to EUR for free.

This is clearly a useful feature, as it allows the programmer to write less conversions. In general, if she defines a chain of n conversions she will have an extra $(n^2 - n)/2$ converting functions available.

As might be expected, this feature does not come without costs. Section 4.4 discusses one effect on the language of the increase in the number of ways a source program can be corrected, due (in part) to the presence of transitivity.

4.3.2 Local Scoping

The language presented in this thesis enables an extensible implicit environment throughout the code. Adjusting the implicit environment makes it possible for conflicting conversions to co-exist in the code, as long as their scope does not overlap. As a corollary, the same function, applied to the same argument, can yield different results due to that fact that different conversions are being used in distinct parts of the code, as specified by the programmer.

As an example of how the implicit environment affects the value of some expression, suppose a complex function *happiness* that computes the happiness of Maria by taking two arguments: the money (in CHF) she has and her location (the name of a city, from a pre determined set of cities). Given these two, the function proceeds by case analysis on her location. Depending on which city she is at, it calls a myriad of functions, some of which require EUR as input. Suppose the converting function CHF to EUR also depends on which city Maria is (*i.e.* the exchange rate of CHF to EUR is higher in Zurich than in Lugano). This program can be simplified by writing *local* conversion axioms whose scope is restricted to the part of *happiness* that implements the behavior for each city. Another example in which the diversified behavior of a function (due to the local scoping of implicit conversions) would be useful is in the generation of boilerplate code.

As will be shown in Section 4.4, altering the implicit environment has much more profound consequences than just locally affecting the values returned by functions: it determines the types of the terms (inclusively whether they are well-typed) and thus which conversion axioms should be used to correct a program. As such, allowing the programmer to write locally scoped conversion axioms enables her to retain full control of the code while providing the convenience of implicit conversions.

4.3.3 Parametric Polymorphism and Constrained Conversions

A conversion axiom that is only applicable if some conditions regarding the existence of other conversion axioms in the implicit environment are met is know as a *constrained conversion*. If a conversion axiom ax_1 is conditional on the existence of another (ax_2), it is likely that the converting expression ax_1 introduces depends on the converting expression associated with ax_2. TrIC takes measures to support this connection between the converting expressions. The fact that TrIC supports multiple conditions on the conversion axioms can be especially useful for converting between types abstracted over multiple type variables: it enables a conversion axiom to state that "it is possible to implicitly convert from `Pair a b` to `Pair c d` if there are implicit conversions from `a` to `c` and `b` to `d`". In the absence of support for multiple type variables, we could be tempted to resort to transitivity in order to achieve the same conversion. However, the languages presented in this text consider such conversions to be ambiguous, as will be discussed.

Another programming feature that enjoys widespread support by programming languages, since it enables code re-use, is parametric polymorphism. Allowing it in implicit conversions brings its benefits to this language extension.

Parametric polymorphism by itself has limited applicability in implicit conversions: not only it may be hard to think of a case in which implicit conversions from and/or to *anything* are useful; but it is also impossible to write an expression capable of converting, for example, from an Integer to some type variable a. However, the constrained conversions and parametric polymorphism can also be used together. If, for example, a list datatype is present and a mapping function defined, the user can easily write an implicit conversion from a list of any type a to a list of any other type b ([a]$\rightsquigarrow_{ax}$[b]) provided it is possible to implicitly convert type a to type b. This feature is further discussed in Section 5.1.1.

4.4 Ambiguity

Implicit type conversions are, in essence, a language feature that automatically corrects (typing) errors. For languages with such correcting features, an immediate concern is to specify its behavior when multiple ways to fix the source program would, *a priori*, be possible.

One aspect to keep in mind is that both parametric polymorphism and transitivity lead to a broader applicability of user-defined conversions. Consequently, the chance of multiple possible "fixes" is higher.

Since predictability is a crucial requirement of any programming language, its behavior must be uniquely defined: for every source program, at most one "corrected" program can exist. This leads to a natural tension between simplicity and expressivity. A finer-grained definition of the language places more of a burden on the user in order to understand the functioning of implicit conversions, while a coarser one maims the expressivity of the language, since more programs will be rejected.

In order to decide on the desired behavior for a language with implicit conversions, consider the program (in Haskell-like syntax) presented in Example 2, where, in addition to the previously mentioned data types (EUR, CHF and DKK), USD (for US Dollar) and Wallet are also defined. The data constructor for type Wallet is polymorphic and takes two arguments of the same type. We can intuitively think of these arguments as amounts of some currency in notes and in coins, even though there is no way to restrict the arguments to currency amounts. Assume the existence of the following implicit conversions: DKK $\rightsquigarrow_{ax}$ EUR and CHF $\rightsquigarrow_{ax}$ EUR (Figure 4.2a).

$$
\begin{aligned}
&\mathit{data}\ \mathrm{EUR} = K_{\mathrm{EUR}}\ \mathit{Float}\\
&\mathit{data}\ \mathrm{CHF} = K_{\mathrm{CHF}}\ \mathit{Float}\\
&\mathit{data}\ \mathrm{DKK} = K_{\mathrm{DKK}}\ \mathit{Float}\\
&\mathit{data}\ \mathrm{USD} = K_{\mathrm{USD}}\ \mathit{Float}\\
&\mathit{data}\ \mathrm{Wallet}\ a = K_{\mathrm{Wallet}}\ a\ a\\
&\\
&K_{\mathrm{Wallet}}\ (K_{\mathrm{DKK}}\ 10.1)\ (K_{\mathrm{CHF}}\ 30.3)
\end{aligned}
$$

Example 2

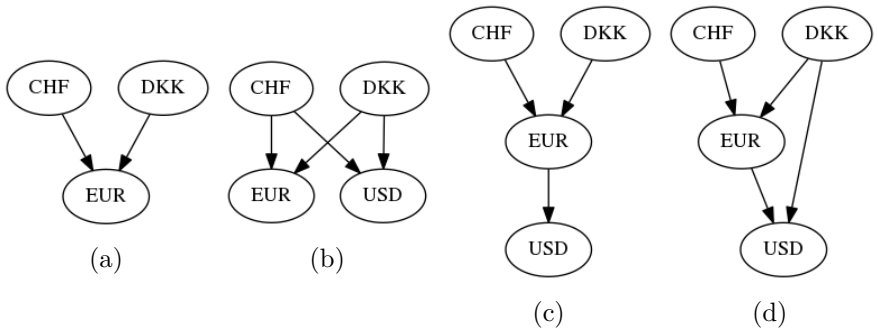


Figure 4.2: Possible implicit environments - resolving type variables

The first aspect to ponder on is whether this program should be accepted: there is no conversion from DKK to CHF nor the other way around. However, it is possible to convert, under the current implicit environment, both arguments of K_{Wallet} to a common type: EUR.

If, in addition to the implicit conversions already defined, the axioms DKK $\rightsquigarrow_{ax}$ USD and CHF $\rightsquigarrow_{ax}$ USD were also available (Figure 4.2b), the same choice would now be arbitrary: both EUR and USD are possible and there is no obvious criteria under which one is better than the other.

For a more intricate case, suppose the implicit conversions defined are: DKK $\rightsquigarrow_{ax}$ EUR, CHF $\rightsquigarrow_{ax}$ EUR and EUR $\rightsquigarrow_{ax}$ USD (Figure 4.2c). Despite still being possible to convert both DKK and CHF to both EUR and USD, a case could be made that program should be corrected (by converting both to EUR) rather than rejected: both DKK and CHF need to be converted to EUR before they can then be converted to USD, meaning that a clear "minimal" solution exists. Finally, how should a fourth conversion, from DKK to USD (Figure 4.2d), be handled?

So far we have considered which type (if any) a given type should be converted to. Another issue to address is what should the behavior be when there are multiple ways to convert between two (ground) types. Suppose we must convert from DKK to EUR and there are two applicable conversion axioms available: DKK $\rightsquigarrow_{ax}$ EUR and $a \rightsquigarrow_{ax}$ EUR (Figure 4.3a), where a is universally quantified. Or suppose there are three axioms, the first two of which can be composed (due to transitivity), to convert from DKK to EUR: DKK $\rightsquigarrow_{ax}$ USD, USD $\rightsquigarrow_{ax}$ EUR, and DKK $\rightsquigarrow_{ax}$ EUR (Figure 4.3b). What should then be the specified behavior?

TrIC considers a conversion constraint between two ground types to be ambiguous if there are multiple ways in which to compose the appropriate axioms. Furthermore, if any of the axioms has conditions that can be entailed in more than one ways, the conversion is considered ambiguous, as will be explained in Chapter 6. One such constraint results in the rejection of the program.

As for deciding to what type τ_t some given types $\overline{\tau_{s_i}}$ should be converted to, a first obvious approach would be to reject any program in which any choices are possible. However, this would mean the program presented in this section, with DKK $\rightsquigarrow_{ax}$ EUR, CHF $\rightsquigarrow_{ax}$ EUR and EUR $\rightsquigarrow_{ax}$ USD (Figure 4.2c) as available

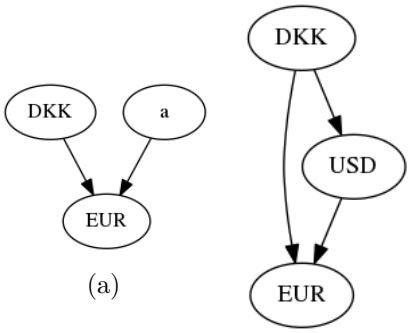


Figure 4.3: Possible implicit environments - constraint with ground types

axioms would be rejected. In order to avoid this severe and unnecessary restriction, TrIC relies on the definition of dominator (Section 4.4.1) and considers a program ambiguous when, in at least one situation, no dominator exists.

4.4.1 Dominator

The notion of *dominator* (from the field of graph theory [18]) was introduced in this text as the answer to the question "to what type should some terms be implicitly converted to?". This question arises under a set of conversion constraints and its answer is computed with respect to the set of conversion axioms available in the implicit environment.

The *dominator* of a set of conversion constraints $S = \overline{\tau_{s_i} \rightsquigarrow_{ct} \tau_{t_i}}$, with $\overline{\tau_{s_i}}$ ground types, is defined as being the type $\tau_{dominator}$ to which all τ_{s_i}'s can convert to and such that, for any other type τ_d such that all τ_{s_i}'s can convert to τ_d, any conversion from any τ_{s_i} to τ_d can be decomposed into a conversion from τ_{s_i} to $\tau_{dominator}$ and another from $\tau_{dominator}$ to τ_d. This text defines implicit type conversion as a reflexive relation.

The generation of constraints will be discussed in detail in Section 5.3. Nonetheless, we can already intuitively infer that the conversion constraints generated under the program presented in Example 2 are that DKK and CHF need to be convertible to some common type: DKK $\rightsquigarrow_{ct} a_1$ and CHF $\rightsquigarrow_{ct} a_1$, for some type variable a_1. For ease of visualization, Figure 4.4 plots multiple conversion axioms in a graph, which will be used to construct multiple implicit environments under which the dominator of these two constraints is computed.

The first set of conversion axioms discussed in this section is $\{ax_1, ax_2\}$. Since there is only one type into which both DKK and CHF are convertible when these are the available axioms, EUR, this is the dominator.

If we now extend the set of available axioms to $\{ax_1, ax_2, ax_3\}$, both DKK and CHF are convertible to both EUR and USD. However, (all) the conversion(s) from DKK to USD can be decomposed into a conversion from DKK to EUR and another

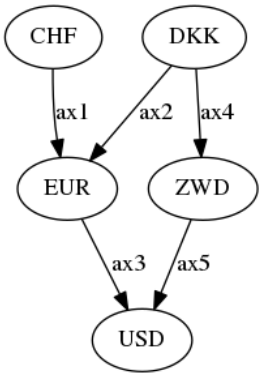


Figure 4.4: Possibilities for Conversion Axioms

from EUR to USD and likewise for the conversion from CHF to USD, so EUR is the dominator of the constraints, reflecting the intended behavior.

Extending the implicit environment to $\{ax_1, ax_2, ax_3, ax_4\}$ does not affect the set of types CHF and DKK can both convert to, nor the paths to these types, so it does not affect the dominator (it is still EUR). Under the assumption that the implicit environment is $\{ax_1, ax_3, ax_4, ax_5\}$, there is once again a single type both CHF and DKK can convert to: USD. As such, USD is the dominator.

Finally, if all the axioms in Figure 4.4 are taken into account, no dominator exists and the program is rejected. In this setting, both EUR and USD are reachable from both CHF and DKK. USD is clearly not the dominator as ax_1 is a conversion from CHF to EUR that can not be decomposed into a conversion from CHF to USD and a conversion from USD to EUR; neither is EUR: the composition of axioms ax_4 and ax_5 is a conversion from DKK into USD that can not be decomposed into a conversion from DKK to EUR and a conversion from EUR to USD.

4.5 Complex Types

Consider again the data type Wallet a, whose constructor K_{WALLET} takes two terms of type a. Section 4.4 discussed the term K_{WALLET} (K_{DKK} 10.1) (K_{CHF} 30.3). Under the implicit environments presented during the discussion of the dominator, this term was either ill-typed or had type Wallet EUR or Wallet USD.

Slightly modifying the term to $\lambda x.(K_{\text{WALLET}}\ (K_{\text{DKK}}\ 10.1)\ x)$ causes the situation to change dramatically. Under standard HM its type would be DKK $\to$ Wallet DKK, which is clearly too restrictive in this setting.

Having understood the idea behind implicit type conversions, we could be inclined to type $\lambda x.(K_{\text{WALLET}}\ (K_{\text{DKK}}\ 10.1)\ x)$ as "$a \to$ Wallet DKK, as long as a can be implicitly converted to DKK". A second thought about this would lead to the realization that this too is overly restrictive: Section 4.4.1 showed that, depending on the implicit environment, both subterms may be converted to some common type. We have now gotten a better intuition about the type of $\lambda x.(K_{\text{WALLET}}\ (K_{\text{DKK}}\ 10)\ x)$: it should be something like "$a \to$ Wallet b, as long as both a and DKK can be

implicitly converted into b". As programs grow, this could lead to unreasonably complex terms to be presented to the user, which is incompatible with the user-friendliness requirement of the problem statement. This will be discussed in detail in Chapter 5.

Chapter 5

TrIC

This chapter presents TrIC (TRansitive Implicit Conversions), a calculus that showcases the features mentioned in Chapter 4, namely type inference and implicit conversions with support for transitivity, local scoping, constrained and polymorphic conversions.

The chapter starts by presenting TrIC's syntax (Section 5.1) and typing rules (Section 5.2). Then, type inference and constraint generation are discussed (Section 5.3); the translation of a TrIC program is addressed by Section 5.4 and the chapter concludes with an example (Section 5.5).

5.1 Syntax

TrIC extends HM with implicit type conversions. As such, the focus of this section is on the syntax of this novel feature of the language, namely on the terms that introduce implicit conversions into the language, *locimp* (short for local implicit), and on the implicit environment they determine, henceforth denoted by I. As will be explained in Chapter 6, the implicit environment is crucial for solving the conversion constraints generated by a program.

A *locimp* term ($locimp\ e_1 : \sigma_{\rightsquigarrow}\ in\ e_2$) is composed of a conversion axiom ($e_1 : \sigma_{\rightsquigarrow}$) and any (sub)term e_2. Similarly to how a *let* expression extends the typing environment Γ for its right hand side term, a *locimp* expression extends the implicit environment under which its expression e_2 is typed.

Conversion axioms introduce a term and specify its conversion type. A conversion type is either a conversion monotype or a conversion polytype. Conversion monotypes ($\tau_{\rightsquigarrow} ::= \tau_s \rightsquigarrow \tau_t$) are simply a pair of monotypes (as defined in HM); τ_s represent the source type and the τ_t the target type of the conversion. A conversion polytype ($\sigma_{\rightsquigarrow} ::= \forall \overline{a}.\overline{(j : \tau_{\rightsquigarrow})} \Rightarrow \tau_{\rightsquigarrow}$) specifies a set of type variables $\overline{a}$, a set of conditionals $\overline{(j : \tau_{\rightsquigarrow})}$ (identifiers bound to conversion monotypes) and a final conversion monotype $\tau_{\rightsquigarrow}$. An implicit environment I is simply the set of available conversions axioms. Intuitively, having an implicit conversion $conv : \tau_1 \rightsquigarrow \tau_2$ in I states that, under I, a term e of τ_1 is implicitly convertible to another of type τ_2, and that $conv$ should be used to convert terms from τ_1 to τ_2. In particular, ($conv\ e$) has type τ_2.

Programs consist of data declarations (responsible for the introduction of data and type constructors to the typing environment), a single expression and a type annotation (the type of the whole program). In this text, the type annotation is required as local type inference is performed (see Chapter 6). However, it is not a necessity of a language with implicits but rather a design choice to be discussed in Chapter 6. The full syntax for TrIC is given in Figure 5.1, where the syntax concerned with implicit conversions is highlighted.

5.1.1 Polymorphic Conversions

As previously stated and deducible from the syntax, TrIC allows the programmer to write conversion axioms ($e : \forall \overline{a}.\overline{j_i : \tau_{\rightsquigarrow i}} \Rightarrow \tau_{\rightsquigarrow}$) parameterized over type variables $\overline{a}$ and constrained by the conditions ($\overline{j_i : \tau_{\rightsquigarrow i}}$) imposed. These are satisfied if, for each required conversion monotype $\tau_{\rightsquigarrow i}$, an implicit conversion of type $\tau_{\rightsquigarrow i}$ can be constructed from the conversion axioms in scope (*i.e.* in the implicit environment).

TrIC allows the identifiers j_i's to be used in the converting expression e. This way the user is able to write a conversion axiom that itself behaves differently in distinct places in the code (it behaves according to the implicits that satisfy its conditions), with the added benefit of using a single identifier to stand for a conversion expression. Each identifier will be substituted, during compile time, by the appropriate converting expression constructed from the implicit environment (as will be seen in detail in Chapter 6).

The example mentioned in Section 4.3.3, in which the goal was to define an implicit conversion from a list of any type a to a list of any other type b provided an implicit conversion from a to b is available, can easily be written (if a list datatype is present and a mapping function has previously been defined) as follows: $(\lambda x.map\ j\ x) : ((j : a \rightsquigarrow b) \Rightarrow [a] \rightsquigarrow [b])$.

5.1.2 Constraints in the Conversion Axioms

In order for type inference to terminate, some requirements are enforced on the size of the types present in the conversion axioms. Further, type variables in the axioms must abide by some rules in order to avoid ambiguity and facilitate the solving process (see Chapter 6). The complete set of requirements for the conversion axioms $e : \forall \overline{a}.\overline{(j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i})} \Rightarrow \tau_s \rightsquigarrow \tau_t$ is:

- in the final monotype of an axiom, the target type τ_t must not be larger than the source type τ_s, *i.e.*, conversions to larger types are not allowed;
- the size of the conversion monotype in each condition $\tau_{s_i} \rightsquigarrow \tau_{t_i}$ (the sum of the sizes of its source and target types) must be strictly inferior to the size of the final conversion monotype $\tau_s \rightsquigarrow \tau_t$;
- all type variables must be well-scoped;
- any type variable present in the source τ_{s_i} of some condition must appear in the source type of the final conversion monotype τ_s;

$$
\begin{array}{lr}
pgm ::= \overline{data};\, e : \sigma & \textit{Program} \\
data ::= data\ T\ a = \overline{K_i\ \overline{\tau_{i_j}}} & \textit{Data Declaration} \\
\\
\tau ::= a \mid \tau_1 \to \tau_2 \mid T\,\overline{\tau} & \textit{Monotypes} \\
\sigma ::= \tau \mid \forall a.\sigma & \textit{Polytypes} \\
\\
\tau_{\rightsquigarrow} ::= \tau \rightsquigarrow \tau & \textit{Conversion Monotypes} \\
\sigma_{\rightsquigarrow} ::= \forall \overline{a}.\overline{(j : \tau_{\rightsquigarrow})} \Rightarrow \tau_{\rightsquigarrow} & \textit{Conversion Polytypes} \\
\\
e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid K \mid case\ e_1\ of\ \overline{(K_i \overline{x_{i_j}}) \to e_2} & \textit{Terms} \\
\quad \mid let\ x : \sigma\ =\ e_1\ in\ e_2 \mid locimp\ e : \sigma_{\rightsquigarrow}\ in\ e_2 & \\
\\
I ::= \overline{e : \sigma_{\rightsquigarrow}} & \textit{Implicit Environment} \\
\\
\Gamma ::= \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, K : \sigma \mid \Gamma, T & \textit{Typing Environment}
\end{array}
$$

Figure 5.1: TrIC's syntax

- the set of all the type variables in the conditions is a subset of the set of the type variables in the final conversion monotype;
- each variable in the target of the final monotype τ_t must be present either on the source type τ_s of the final conversion monotype or in the target type of some condition τ_{t_i}.

5.2 Typing

The top-level program typing rules are standard for languages with data types. The type annotation will be taken into account when performing type inference on the program's term. Program typing rules and data declaration typing are shown respectively in Figure 5.2 and Figure 5.3.

As for term typing, shown in Figure 5.4 (where the "$\Gamma \vdash_{ty} \sigma$" judgment is the same as in Figure 2.11), the main differences from standard HM are the typing rules for application and case expressions. The intuition for rule TmApp is that it is no longer required that the type of the operand (τ_s) matches the left side of the (function) type of the operator ($\tau_{arg} \to \tau_{res}$) literally, but rather that there is exactly one way to to implicitly convert from τ_s to τ_{arg} under the current implicit environment (henceforth denoted by $I \vDash_{th} \tau_s \rightsquigarrow \tau_{arg}$), as will be seen in Chapter 6.

Similarly, TmCase requires the weaker condition that the type of the scrutinee is convertible to the (unique) type of the pattern(s). Further, instead of imposing the same type to all of the alternatives, it merely imposes that every alternative is convertible to a common type.

$\boxed{I;\Gamma \vdash_p pgm : \sigma}$ Program Typing

$$\frac{\Gamma \vdash_{data} \overline{data} : \Gamma' \qquad I;\Gamma' \vdash_p pgm : \sigma}{I;\Gamma \vdash_p \overline{data}; pgm : \sigma}\ \textsc{Data}$$

$$\frac{I;\Gamma \vdash_{tm} e : \sigma}{I;\Gamma \vdash_p e : \sigma}\ \textsc{Expression}$$

Figure 5.2: Program Typing

$\boxed{\Gamma \vdash_{data} data : \Gamma'}$ Data Declaration Typing

$$\frac{}{\Gamma \vdash_{data} (data\ T\ a = K\ \overline{\tau}) : \Gamma, T, (K : \forall a.\overline{\tau} \to T\ a)}\ \textsc{Decl}$$

Figure 5.3: Declaration Typing

When typing a *locimp*, (TmLocImp rule) the converting expression e_1 introduced by the conversion axiom $(e_1 : \overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_s \rightsquigarrow \tau_t)$ of *locimp* must have type $\tau_s \to \tau_t$, under an extended typing environment where the j_i's have type $\tau_{s_i} \to \tau_{t_i}$, since the j_i's will later be substituted in e_1 by terms of those types. The subterm e_2 of *locimp* is then typed under an implicit environment extended with the conversion axiom $e_1 : \overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_s \rightsquigarrow \tau_t$. As explained in Chapter 4, this means that the newly defined axiom can be used to "fix" e_2.

Note that the local scope means that if e_2 has some sub-terms whose types do not match but can be made to match by application of $e_1 : \overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_s \rightsquigarrow \tau_t$, then the axiom may be used. If, however, the entire *locimp* term is a sub-term of some term e_{super} and its type does not conform to expected, the conversion it introduces will not be considered in order to fix the program. If that is the intended behavior, the conversion axiom needs to be available in a broader scope: it should be defined in a *locimp* that has e_{super} as a subterm.

5.3 Constraint Generation and Type Inference

To infer the type of a program, a standard two-phased approach is used. First, the typing environment is built from the data declarations and the term is traversed to collect both the equality constraints (identical to HM) and the novel conversion constraints. The second phase consists of solving all the constraints.

Since the implicit environment is extended (according to the scope of the *locimp* terms) during the traversal, constraints must include the implicit environment under which they were generated in order to know which conversions axioms are available at each point in the program. A conversion constraint with respect to the conversion monotype $\tau_{source} \rightsquigarrow \tau_{target}$ and generated under the implicit environment I is

$\boxed{I;\Gamma \vdash_{tm} e : \sigma}$ Term Typing

$$\frac{(x:\sigma) \in \Gamma}{I;\Gamma \vdash_{tm} x : \sigma}\ \textsc{Var} \qquad \frac{(K:\sigma) \in \Gamma}{I;\Gamma \vdash_{tm} K : \sigma}\ \textsc{Constr}$$

$$\frac{I;\Gamma \vdash_{tm} e_1 : \tau_{arg} \to \tau_{res} \qquad I;\Gamma \vdash_{tm} e_2 : \tau_s \qquad I \vDash_{th} \tau_s \rightsquigarrow \tau_{arg}}{I;\Gamma \vdash_{tm} e_1\ e_2 : \tau_{res}}\ \textsc{TmApp}$$

$$\frac{I;\Gamma, x:\tau_1 \vdash_{tm} e : \tau_2}{I;\Gamma \vdash_{tm} \lambda x.e : \tau_1 \to \tau_2}\ \textsc{TmAbs}$$

$$\frac{I;\Gamma, x:\sigma_1 \vdash_{tm} e_1 : \sigma_1 \qquad I;\Gamma, x:\sigma_1 \vdash_{tm} e_2 : \tau_2}{I;\Gamma \vdash_{tm} let\ x : \sigma_1 = e_1\ in\ e_2 : \tau_2}\ \textsc{TmLet}$$

$$\frac{\overline{(K_i : \forall a.\overline{\tau_{i_k}} \to T\ a) \in \Gamma} \qquad I;\Gamma \vdash_{tm} e_1 : \tau_1 \qquad \overline{I;\Gamma, \overline{x_{i_j}} : [a \mapsto \tau]\overline{\tau_{i_k}} \vdash_{tm} e_i : \tau_i} \qquad I \vDash_{th} \tau_1 \rightsquigarrow T\ \tau \qquad \overline{I \vDash_{th} \tau_i \rightsquigarrow \tau_2}}{I;\Gamma \vdash_{tm} case\ e_1\ of\ \overline{(K_i\ \overline{x_{i_j}}) \to e_i} : \tau_2}\ \textsc{TmCase}$$

$$\frac{\begin{array}{c} I;\Gamma, \overline{a}, \overline{j_i : \tau_{s_i} \to \tau_{t_i}} \vdash_{tm} e_1 : \tau_s \to \tau_t \\ I, (e_1 : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_s \rightsquigarrow \tau_t); \Gamma \vdash_{tm} e_2 : \tau_2 \end{array}}{I;\Gamma \vdash_{tm} (locimp\ e_1 : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_s \to \tau_t\ in\ e_2) : \tau_2}\ \textsc{TmLocimp}$$

$$\frac{I;\Gamma, a \vdash_{tm} e : \sigma}{I;\Gamma \vdash_{tm} e : \forall a.\sigma}\ \textsc{Forall-Intro}$$

$$\frac{I;\Gamma \vdash_{tm} e : \forall a.\sigma \qquad \Gamma \vdash_{ty} \tau}{I;\Gamma \vdash_{tm} e : [a \mapsto \tau]\sigma}\ \textsc{Forall-Elim}$$

Figure 5.4: TrIC's Term Typing

henceforth denoted by $\tau_{source} \rightsquigarrow_I \tau_{target}$. It states that, under I it must be possible to implicitly convert from τ_{source} to τ_{target}. The process of entailing conversion constraints is explained in detail in Chapter 6.

As shown in the top-level inference rule (Figure 5.5), TrIC assigns type schemes to its programs. The first premise is closely related to this chapter, as it is concerned with the collection of equality and conversion constraints, denoted by E and Y respectively. The second and third premises determine type substitutions (Θ_E and Θ_Y) that shall be discussed in Chapter 6 as they regard solving the constraints (in particular, Θ_Y is one of the main contributions of this thesis and is discussed in Section 6.1). The fourth determines a term substitution φ to be discussed in Section 6.2. Note how the type variables in the user-provided annotation can not be substituted in the unification step (as substituting them would result in a type less general than intented).

Figure 5.6 shows the rules for type inference and constraint generation (the highlighted part regards term elaboration and can be ignored for now). The generated constraints are easy to understand if we keep in mind HM's generation of constraints as well as the intuition provided when explaining the typing rules. Upon encountering an application $(e_1\ e_2)$, constraints are generated as follows: if the operator e_1 has type τ_1 and the operand e_2 type τ_2, $e_1\ e_2$ is typed as having type a and an equality constraint specifying that τ_1 is of the form $b \to a$, for fresh type variables a and b is generated. A conversion constraint specifying the need to be able to implicitly convert from τ_2 to b, under the current implicit environment I is also generated. In essence, these constraints state that neither the type of the application nor the type accepted by the operator are known at this time, but it must be possible to implicitly convert from the type of the operand τ_2 to the type expected by the operator (otherwise the application would be ill-typed).

The idea for Case expressions $(case\ e_1\ of\ \overline{(K_i\ \overline{x_i}) \to e_i})$ is that it must be feasible to implicitly convert from the type of the scrutinee to (an instantiation of) the type specified by the data constructor on the patterns; and that all the types of the alternatives are convertible to one common type. Concretely, for a scrutinee e_1 with type τ_1, a conversion constraint from τ_1 to $T\ b$, with T the appropriate type constructor and b a fresh type variable, is generated. Another type variable d is generated and, likewise, it is constrained to be equal to the type τ_i of each resulting expression in the alternatives, computed under the typing environment extended by binding the variables in the patterns to the appropriate types. Other terms in TrIC merely collect the constraints of their sub-expressions.

The first two premises of the LocImp rule state that the type inferred for the converting expression is allowed to be more general than the type provided by the user (allowing, for example for cases in which the argument of the converting expression is discarded).

After collecting the constraints generated by the term, one last equality constraint is added: the type resulting from type inference on the term must be equal to the type in the type annotation. Section 5.5 discusses some of the rationale for this last equality constraint.

Note that the first premise of both Let and LocImp use the top-level judgment.

$$\boxed{I; \Gamma \vdash_{tm} (e : \forall \overline{a}.\tau) : \sigma;\ e'}$$

$$\frac{\begin{array}{c} I; \Gamma, \overline{a} \vdash_{tm} e : \tau'; E; Y;\ e' \\ \Theta_E = unify(\overline{a}; E, \tau' \sim \tau) \qquad \Theta_Y = dominator(\Theta_E\ Y) \\ \varphi = conversions(\Theta_Y(\Theta_E Y)) \qquad fv(\Theta_Y(\Theta_E(\tau'))) \subseteq \overline{a} \qquad (\Theta_Y(\Theta_E(\tau'))) = \tau \end{array}}{I; \Gamma \vdash_{tm} e : \forall \overline{a}.\tau;\ \varphi(e')}$$

Figure 5.5: TrIC's top-level inference judgment

This means that the respective terms e_1 are typed locally and do not generate constraints as far a their "superterms" are concerned. The local typing enables both of these terms to be assigned type schemes, allowing for polymorphic LET and polymorphic conversion axioms, respectively.

5.4 Translation to a ITC-free language

Implicit type conversions require type information that is no longer present at runtime. In order to overcome this hurdle, a TrIC program is first translated to a language without implicit conversions and for which there is an interpreter.

In the implementation provided with this thesis, the target language is System F, which is a proper subset of the intermediate language used by GHC (System F_C [23]), so that it can later be adapted to extend GHC. However, for the sake of exposure, the main text is concerned with the translation of a TrIC program into the non-highlighted subset of the language as show in Figure 5.1. This (subset) language is also a subset of the KHC, which means that, after the translation, the KHC could be used to run the program. The translation to System F is shown in the Appendix B.

This translation involves using the converting expressions introduced with each conversion axiom to effectively transform an ill-typed program into a well-typed program of the target language.

The idea is to introduce place-holding terms (usually denoted by j) in the program to act as hooks onto which the appropriate converting expressions can then be inserted. These converting expressions act as bridges between the subterms of applications and case expressions. Whenever a conversion constraint $\tau_s \rightsquigarrow_I \tau_t$ is generated, it is stored together with its corresponding place-holder, as is specified in Figure 5.6: $(j, \tau_s \rightsquigarrow_I \tau_t)$. This approach ensures that the converting expression (computed as explained in Chapter 6) is inserted in the correct place in the code: it will substitute the place-holder j. An advantage of using with place-holders is that it allows the collection of all the constraints generated by the program before the solving process starts, providing a clear separation of the two phases.

Terms/programs with place-holders as known as *partially elaborated*. Note that in an application $e_1\, e_2$ a place holding variable j is put between the partially elaborated operator e_1' and operand e_2'. The converting expression that will substitute j will be

$\boxed{I; \Gamma \vdash_{tm} e : \tau; E; Y; e'}$

$$\frac{(x : \forall \overline{a}.\tau) \in \Gamma \qquad freshb}{I; \Gamma \vdash_{tm} x : [a \mapsto b]\tau; \bullet; \bullet; x'} \textsc{Var} \qquad \frac{(K : \forall a.\tau) \in \Gamma \qquad \overline{b}\ fresh}{I; \Gamma \vdash_{tm} K : [a \mapsto \overline{b}]\tau; \bullet; \bullet; K'} \textsc{Constr}$$

$$\frac{I; \Gamma, x : a \vdash_{tm} e : \tau; E; Y; e' \qquad fresh\ a}{I; \Gamma \vdash_{tm} \lambda x.e : a \to \tau; E; Y; e'} \textsc{Abstraction}$$

$$\frac{\begin{array}{c} I; \Gamma \vdash_{tm} e_1 : \tau_1; E_1; Y_1; e_1' \\ I; \Gamma \vdash_{tm} e_2 : \tau_2; E_2; Y_2; e_2' \qquad fresh\ a,\ b,\ j \end{array}}{I; \Gamma \vdash_{tm} e_1 e_2 : a; E_1, E_2, (\tau_1 \sim b \to a); Y_1, Y_2, (j, \tau_2 \rightsquigarrow_I b); e_1'(j\ e_2')} \textsc{Application}$$

$$\frac{\begin{array}{c} \overline{(K_i : \forall a.\overline{\tau_{i_k}} \to T\ a) \in \Gamma} \qquad I; \Gamma \vdash_{tm} e_1 : \tau_1; E_1; Y_1; e_1' \\ I; \Gamma, \overline{x_{i_j}} : [a \mapsto b]\overline{\tau_{i_k}} \vdash_{tm} e_{2i} : \tau_i; E_i; Y_i; e_{2i}' \qquad fresh\ b,\ d,\ j_1,\ \overline{j_i} \\ E = E_1, \overline{E_i} \qquad Y = Y_1, (j_1, \tau_1 \rightsquigarrow_I T\ b), \overline{Y_i}, \overline{(j_i, \tau_i \rightsquigarrow_I d)} \end{array}}{I; \Gamma \vdash_{tm} case\ e_1\ of\ \overline{(K_i\ \overline{x_{i_j}}) \to e_{2i}} : d; E; Y; case\ (j_1\ e_1')\ of\ \overline{(K_i'\ \overline{x_{i_j}'}) \to j_i\ e_{2i}'}} \textsc{Case}$$

$$\frac{I; \Gamma \vdash_{tm} e_1 : \sigma_1; e_1' \qquad I; \Gamma, x : \sigma_1 \vdash_{tm} e_2 : \tau_2; E_2; Y_2; e_2'}{I; \Gamma \vdash_{tm} let\ x : \sigma_1 = e_1\ in\ e_2 : a; E_1, E_2; Y_1, Y_2; let\ x : \sigma_1 = e_1'\ in\ e_2'} \textsc{Let}$$

$$\frac{\begin{array}{c} I; \Gamma, \overline{j_i : \tau_{s_i} \to \tau_{t_i}} \vdash_{tm} e_1 : \forall \overline{a}.\tau; e_1' \\ unify(fv(\tau_{source} \to \tau_{target}); \tau \sim \tau_{source} \to \tau_{target}) = \theta \\ I, e_1 : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_{source} \rightsquigarrow \tau_{target}; \Gamma \vdash_{tm} e_2 : \tau; E; Y; e_2' \end{array}}{I; \Gamma \vdash_{tm} (locimp\ e_1 : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_{source} \rightsquigarrow \tau_{target}\ in\ e_2) : \tau; E; Y; e_2'} \textsc{LocImp}$$

Figure 5.6: Type Inference and Partial Translation

a lambda abstraction, whose type is a function type from the type of the operand to the type expected by the operator. In case the types already match, it will simply be substituted by the identity function.

Similarly, for case expressions $case\ e_1\ of\ \overline{(K_i\ \overline{x}) \to e_{2i}}$, place-holders will be put to the left of the scrutinee and of the terms returned from the alternatives so that they can be later replaced by the appropriate conversions. Since no other terms generate conversion constraints, no other terms introduce place-holders.

5.5 Examples

Two example programs in TrIC are now presented. Besides showcasing the syntax, these illustrate the constraint generation process.

5.5.1 A Simple Example

The first program, in Example 3, defines one data type (assume *Float* and *Int* to be previously defined base types), a simple conversion axiom and a final term in conspicuous need of that implicit conversion. All axioms that appear in the program before any (non-*locimp*) term can be considered as having a *global* scope which is the case for the axiom "$\lambda x.3.1415 : \text{Int} \to \text{Float}$" defined in this example. Such axioms may be used anywhere in the program, except in the converting expressions introduced by prior axioms.

We will now collect the constraints generated by this program, as described in Section 5.3. Declarations construct the typing environment, without generating any constraint. The top-level expression of the program is a *locimpl* term: its first part introduces a new conversion axiom into the implicit environment; its second is an application.

Both subterms of the application are typed without generating constraints. The application itself generates one equality constraint ($\text{Float} \to \text{DKK} \sim b \to a$) and one conversion constraint: $\text{Int} \rightsquigarrow_I b$, with $I = (\lambda x.3.1415) : (\text{Int} \to \text{Float})$. Another equality constraint is generated from the type annotation: $a \sim \text{DKK}$.

$$
\begin{array}{l}
\mathit{data}\ \text{DKK} = K_{\text{DKK}}\ \mathit{Float};\\
(\mathit{locimp}\ \lambda x.3.1415 : (\mathit{Int} \to \mathit{Float})\ \mathit{in}\ (K_{\text{DKK}}\ 1))\\
: \text{DKK}
\end{array}
$$

Example 3: A simple TrIC program.

5.5.2 A More Involved Example

Example 4 presents a slightly more complex program: it defines five data types, one of which contains type variables, and five conversion axioms, one of which with a restricted scope.

By design, neither data declarations nor the declarations of axioms in the first part of *locimp* terms generate constraints. Before generating the constraints from the top-level application $(K_{\text{Wallet}}\ (K_{\text{CHF}}\ 5.25))\ (\mathit{locimp}\ \lambda x.3.1415 : (\mathit{Int} \to \mathit{Float})\ \mathit{in}\ (K_{\text{DKK}}\ 1))$, the constraints must be generated for its subterms. The $K_{\text{Wallet}}\ (K_{\text{CHF}}\ 5.25)$ generates two equality and two conversion constraints: one of each for each application. The right hand side is the same term as presented in Example 3, and as such generates similar constraints, but under a new implicit environment.

Note that, due to being the second part of a *locimp* term, the constraints generated by '$K_{\text{DKK}}\ 1$' specify a richer implicit environment (I_2, consisting of all the five conversion axioms defined in the program) than any of the other constraints, generated under I_1, which consists only of the "global" conversion axioms.

The top-level application generates an additional equality and conversion constraints, resulting in the set of equality constraints $\{\mathit{Float} \to \text{CHF} \sim b_1 \to b_2, a \to a \to (\text{Wallet}\ a) \sim b_3 \to b_4,\ \mathit{Float} \to \text{DKK} \sim b_5 \to b_6,\ b_4 \sim b_7 \to b_8\}$

and in the set of conversion constraints $\{Float \rightsquigarrow_{I_1} b_1,\ b_2 \rightsquigarrow_{I_1} b_3,\ Int \rightsquigarrow_{I_2} b_5,\ b_6 \rightsquigarrow_{I_1} b_7\ \}$. A final equality constraint resulting from the user-given type is then stored: $\{\text{Wallet USD} \sim b_8\}$.

It should be noted that the type annotation also enables elaborating programs that would otherwise be considered ambiguous due to polymorphic functions (introduced in *let* expressions) and polymorphic data constructors.

This program would still be well-type in TrIC without the type annotation (but its type would then be Wallet EUR). However, a minor modification to the program to include a conversion axiom from DKK to USD would cause this program to be rejected if no annotation was present (see Section 6.1.3).

$$
\begin{array}{l}
data\ \text{EUR} = K_{\text{EUR}}\ Float; \\
data\ \text{CHF} = K_{\text{CHF}}\ Float; \\
data\ \text{DKK} = K_{\text{DKK}}\ Float; \\
data\ \text{USD} = K_{\text{USD}}\ Float; \\
data\ \text{Wallet}\ a = K_{\text{Wallet}}\ a\ a \\
\\
locimp\ \lambda x.K_{\text{EUR}}\ 28.0 : (\text{DKK} \rightarrow \text{EUR})\ in \\
locimp\ \lambda x.K_{\text{EUR}}\ 1.2 : (\text{CHF} \rightarrow \text{EUR})\ in \\
locimp\ \lambda x.K_{\text{USD}}\ 3.4 : (\text{EUR} \rightarrow \text{USD})\ in \\
locimp\ \lambda x.case\ x\ of\ K_{\text{Wallet}}\ x_1\ x_2 \rightarrow K_{\text{Wallet}}(j\ x_1)\ (j\ x_2) \\
\quad : (\forall a, b.j : a \rightsquigarrow b \Rightarrow \text{Wallet}\ a \rightsquigarrow \text{Wallet}\ b)\ in \\
K_{\text{Wallet}}\ (K_{\text{CHF}}\ 5.25)\ (locimp\ \lambda x.3.1415 : (Int \rightarrow Float)\ in\ (K_{\text{DKK}}\ 1)) \\
: * : \text{Wallet USD}
\end{array}
$$

Example 4: A more complex TrIC program

Figure 5.7 presents the partial elaboration of the program from Example 4. The j_i's are the place-holding variables introduced during partial elaboration.

$$
\begin{array}{l}
data\ \text{EUR} = K_{\text{EUR}}\ Float; \\
data\ \text{CHF} = K_{\text{EUR}}\ Float;; \\
data\ \text{DKK} = K_{\text{DKK}}\ Float; \\
data\ \text{USD} = K_{\text{USD}}\ Float; \\
data\ \text{Wallet}\ a = K_{\text{Wallet}}\ a\ a; \\
\\
(K_{\text{Wallet}}\ (j_2\ (K_{\text{CHF}}\ (j_1\ 5.25))))\ (j_4\ (K_{\text{DKK}}\ (j_3\ 1)))
\end{array}
$$

Figure 5.7: Partial elaboration of the TrIC program in Example 4.

Chapter 6

Constraint Solving

Chapter 5 discussed the collection of the constraints generated by a TrIC program as well as its partial elaboration to a subset of TrIC without implicit conversions. It remains to be assessed whether the generated constraints are entailable. If so, it must be possible to complete the elaboration by substituting the place-holding variables by the appropriate converting expressions, so that the corresponding System F program type-checks. If not, or if the source program can be elaborated in multiple ways, it shall be rejected.

This chapter discusses the approach taken to check the entailability of the constraints (involving a graph-based solver), the detection of ambiguous programs and the strategy employed to compute the converting expressions to be inserted in the code (Sections 6.1 and 6.2). Section 6.3 discusses the necessary computations for the aforementioned purposes and 6.4 discusses theorems and conjectures regarding TrIC's properties. The chapter ends with an example showcasing constraint solving from top to bottom and the full elaboration of a program (Section 6.5).

6.1 Instantiating Type Variables

The first step towards solving all the collected constraints is to perform unification (exactly as in Figure 3.5) on the equality constraints. Unless this process fails, in which case the program is immediately rejected, it yields the type substitution Θ_E necessary to entail the equality constraints. At this point only the conversion constraints resulting of applying Θ_E to the original constraints need to be entailed.

6.1.1 Motivating Example

Consider again the program presented in Example 2. Neglect the lack of annotation for the sake of exposing corner cases regarding the entailment process of the generated constraints without resorting to overly complex programs.

After solving the equality constraints and applying the resulting type substitution to conversion constraints, these become: $\{Float \rightsquigarrow_{I_1} Float,\ Float \rightsquigarrow_{I_1} Float,\ \text{DKK} \rightsquigarrow_{I_1} a, \text{CHF} \rightsquigarrow_{I_1} a\ \}$.

Assuming $I_1 = \{e_1 : \text{DKK} \rightsquigarrow \text{EUR},\ e_2 : \text{CHF} \rightsquigarrow \text{EUR}\}$, it is clear that it is not possible to convert CHF to DKK nor the other way around under I_1; however, both are convertible to EUR. If the conversion constraints were to be solved sequentially, a would be substituted by either DKK or by CHF, depending on which is considered first, thus leading to the rejection of a sensible program (according to the Chapter 4's definition of sensible).

Consequently, this section develops a graph-based constraint solver that considers multiple constraints in order to instantiate a type variable and accepts programs as long as there is always a best type to substitute a type variable with, *i.e.*, as long as there is a dominator for each set of related constraints.

6.1.2 Building the Graph

After applying the substitution resulting from unification (on the equality constraints) to the conversion constraints, the goal is to substitute the remaining type variables by ground types. The constraints $\overline{\tau_{s_i} \rightsquigarrow_{I_i} \tau_{t_i}}$ with remaining type variables are plotted in a graph, with nodes for each monotype and directed edges representing the need to convert from τ_{s_i} to the other τ_{t_i}. Note that these edges are indexed by the implicit environments I_i specified by each constraint, and it may be the case that multiple edges with the same direction exist between the same types.

To construct the graph, a first conversion constraint $\tau_{s_1} \rightsquigarrow_{I_1} \tau_{t_1}$ is picked and, if it contains type variables, two nodes (τ_{s_1} and τ_{t_1}) are created and joined by a directed edge indexed by the implicit environment I_1. Conversions are then sequentially handled and, if either of its types is a type variable already in the graph, the edges are built from there. Nodes that are not type variables are connected to the rest of the graph by exactly one edge (either incoming or outgoing). This approach avoids cycles and thus there is a clear order in which to entail the conversion constraints: from the sources to the sinks; it also leads to multiple identical nodes.

Upon encountering a constraint $\tau_{s_i} \rightsquigarrow_{I_i} \tau_{t_i}$ such that either τ_{s_i} or τ_{t_i} contains, but is not, a type variable a already in the graph, the components in the graph are sorted according to a partial order to determine which to address first. Whenever τ_s contains a, its component should be handled after the one that contains a.

For each component, its sources are known to be ground types. Type variables in the sources of the first components to be handled could only occur due to top-level lambda abstractions. Solving the equality constraint involving the annotated type ensures their replacement by ground types. Type variables in the sources of other components are substituted by ground types before considering that component due to the partial order enforced.

The approach is to follow the edges of the graph, starting from the sources. Upon encountering a type variable a, all other constraints that have a as the second part of their conversion monotypes are collected. Then the *dominator* (6.1.3) of this set of conversion constraints is computed and substitutes a everywhere. The process continues by following the edges towards the sinks. The type variables encountered are necessarily the second part of the conversion monotype, since the sources are

ground and we only move towards the sinks after the types closer to the sources have been made ground.

6.1.3 The Dominator

As discussed in Chapter 4, the *dominator*, of a set of conversion constraints $S = \overline{\tau_{s_i} \rightsquigarrow_{I_i} a}$, with $\overline{\tau_{s_i}}$ ground types, is the type $\tau_{dominator}$ to which all τ_{s_i}'s can convert to and that, for any other reachable type τ_d such that all τ_{s_i}'s can convert to τ_d, any conversion from any τ_{s_i} to τ_d can be decomposed into a conversion from τ_{s_i} to $\tau_{dominator}$ and another from $\tau_{dominator}$ to τ_d.

To formalize this notion of *being able to implicitly convert*, Figure 6.1 introduces two judgments: Theorem Conversion Judgment specifies the existence of an implicit conversion between two types; Axiom Conversion Judgment is stronger as it states that only one conversion axiom is needed to implicitly convert from its source type to its target type.

Conversion Judgments

The axiom conversion judgment ($I \vDash_{ax} \tau_1 \rightsquigarrow \tau_2$), formalized in Figure 6.1 states that there is a conversion axiom in I that can be used to implicitly convert from τ_1 to τ_2.

For a conversion axiom $e : \forall \overline{a}.\overline{ji : \tau_{\rightsquigarrow}} \Rightarrow \tau_s \rightsquigarrow \tau_t$, to be applicable to implicitly convert τ_1 to τ_2, there must be a substitution Θ that unifies the axioms' final conversion monotype $\tau_s \rightsquigarrow \tau_t$ with $\tau_1 \rightsquigarrow \tau_2$, and grounds all type variables and entails the conditions $(\overline{ji : \tau_{\rightsquigarrow}})$.

In order to prove the satisfiability of the conditions, the auxiliary judgment Theorem is used. Its specification resembles Axiom's but it allows for the conversions to be satisfied by composing multiple conversion axioms.

The highlighted parts are important for the construction of the conversions but can be ignored for now.

Finding the Dominator

The fact that *being able to implicitly convert* is a reflexive relation between types follows immediately from the Unification rule in Figure 6.1. Together with the definition of dominator, this implies that for any environment, the dominator of a single conversion constraint $\tau \rightsquigarrow_I a$ is τ. This reasoning also applies if the source types of all the constraints are the same.

In more complex cases, the dominator of a set of conversion constraints whose target is the same type variable $\overline{\tau_i \rightsquigarrow_{I_i} a}$ is resolved by first computing, for each constraint $\tau_i \rightsquigarrow_{I_i} a$, the set R_i of types reachable from τ_i under the implicit environment I_i ($\tau_i \subseteq R_i$, see Subsection 6.3.1). The intersection of all R_i constitutes the set D of possible dominators: for a type τ_d to be in D, it must be reachable from all the types $\overline{\tau_i}$. Then, the set P of all paths from each τ_i to each type $\tau_d \in D$ is computed. There is a dominator if and only if there is only one type present in all paths in P (the type being the dominator itself), so it suffices to compute their intersection.

$\boxed{I \vDash_{ax} \tau_1 \rightsquigarrow \tau_2}$ Axiom Conversion Judgment

$$\frac{[fv(\tau_1) \mapsto \overline{\tau_{1i}}]\tau_1 = [fv(\tau_2) \mapsto \overline{\tau_{2i}}]\tau_2}{I \vDash_{ax} \tau_1 \rightsquigarrow \tau_2}\ \textsc{Unification}$$

$$\frac{\begin{array}{c}(e : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_s \rightsquigarrow \tau_t) \in I \qquad \exists \Theta : (\tau_1 = \Theta\tau_s \qquad \tau_2 = \Theta\tau_t \\ \forall\ i\ :\ (I \vDash_{th} \Theta\tau_{s_i} \rightsquigarrow \Theta\tau_{t_i}) \qquad \forall i.ftyvsOf(\Theta\tau_{s_i}) = ftyvsOf(\Theta\tau_{t_i}) = \emptyset)\end{array}}{I \vDash_{ax} \tau_1 \rightsquigarrow \tau_2}\ \textsc{Axiom}$$

$\boxed{I \vDash_{th} \tau_1 \rightsquigarrow \tau_2}$ Theorem Conversion Judgment

$$\frac{[fv(\tau_1) \mapsto \overline{\tau_{1i}}]\tau_1 = [fv(\tau_2) \mapsto \overline{\tau_{2i}}]\tau_2}{I \vDash_{th} \tau_1 \rightsquigarrow \tau_2}\ \textsc{Unification}$$

$$\frac{\begin{array}{c}(e : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_s \rightsquigarrow \tau_t) \in I \qquad \exists \Theta : (\tau_1 = \Theta\tau_s \qquad \tau_3 = \Theta\tau_t \\ I \vDash_{th} \tau_3 \rightsquigarrow \tau_2 \qquad \forall i.ftyvsOf(\Theta(\tau_{s_i})) = ftyvsOf(\Theta(\tau_{t_i})) = \emptyset \\ \forall\ i\ :\ (I \vDash_{th} \Theta(\tau_{s_i}) \rightsquigarrow \Theta(\tau_{t_i})))\end{array}}{I \vDash_{th} \tau_1 \rightsquigarrow \tau_2}\ \textsc{Theorem}$$

Figure 6.1: Declarative version of the Axiom and Theorem Conversion Judgments

If there is a constraint of the form $\tau_s \rightsquigarrow_{I_k} \tau_t$, with a a type variable of τ_t, this constraint could also be taken into account when instantiating a. The types reachable from τ_s are computed and unified with τ_t to get the possible values for a from this constraint (R_k). Afterwards, the paths from τ_s to the types τ_{t_d} (τ_t in which a was substituted by the elements $\tau_d \in D$) would be analyzed together with the paths in P. However, the implementation provided with this thesis does not support this approach: constraints in which τ_t contains a type variable a are not considered when instantiating a, in order to avoid an explosion on the complexity of dominator step. Finding a tractable approach that takes such constraints into account is left as future work.

If, in any of the fore mentioned cases, it is not possible to compute the dominator, since either D is empty and no conversion exists or the source program is ambiguous and there is no *best* choice of the type to instantiate a type variable with.

$\boxed{\overline{\tau}; I \vDash_{th} \tau_1 \rightsquigarrow \tau_2; \Theta;\ e}$

$$\frac{[fv(\tau_1) \mapsto \overline{\tau_{1i}}]\tau_1 = [fv(\tau_2) \mapsto \overline{\tau_{2i}}]\tau_2}{\overline{\tau}; I \vDash_{th} \tau_1 \rightsquigarrow \tau_2; [fv(\tau_1) \mapsto \overline{\tau_{1i}}] \circ [fv(\tau_2) \mapsto \overline{\tau_{2i}}];\ \lambda x.x}\ \textsc{Unification}$$

$$\frac{\begin{array}{c}\exists^1(e : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_s \rightsquigarrow \tau_t) \in I : \qquad (\tau_1 = [\overline{a} \mapsto \overline{\tau}]\tau_s \\ \forall\ i\ :\ (\bullet; I \vDash_{th} \Theta_1 \circ \Theta_{i-1}([\overline{a} \mapsto \overline{\tau}]\tau_{s_i}) \rightsquigarrow \Theta_1 \circ \Theta_{i-1}([\overline{a} \mapsto \overline{\tau}]\tau_{t_i}); \Theta_i;\ e_i) \\ \tau_3 = \overline{\Theta_i} \circ [\overline{a} \mapsto \overline{\tau}]\tau_t \\ \tau_3 \notin \overline{\tau} \qquad \overline{\tau}, \tau_1; I \vDash_{th} \tau_3 \rightsquigarrow \tau_2; \Theta;\ e_{rest} \qquad ftyvsOf(\tau_3) = \emptyset)\end{array}}{\overline{\tau}; I \vDash_{th} \tau_1 \rightsquigarrow \tau_2; \Theta \circ \overline{\Theta_i} \circ [\overline{a} \mapsto \overline{\tau}];\ \lambda x.e_{rest}\ ([\overline{j_i \mapsto e_i}]e\ x)}\ \textsc{Theorem}$$

Figure 6.2: Algorithm to construct the conversion

6.2 Constructing the Conversion

Once all the type variables in the conversion constraints have been substituted, it is time to assess their entailability by constructing the conversions. This process is central for the elaboration of the program since it yields the appropriate terms to substitute the place holding variables with. Furthermore, during this process the design decision of rejecting ambiguous programs is again enforced.

Each constraint $\tau_s \rightsquigarrow_{I_i} \tau_t$ (the conversion monotype $\tau_s \rightsquigarrow \tau_t$ is now ground) is considered. If the types τ_s and τ_t are equal, the constraint is trivially entailed and the next one is analysed. This ensures that programs that do not require implicit conversions will not be affected. Otherwise, all the conversion paths (according to 6.3.2) from τ_s to τ_t are computed. If multiple paths are found, the program is ambiguous and as such rejected. If none is found, the constraint is can not be entailed.

However, there may exist multiple ways in which the axioms' conditions are satisfied. Since these conditions determine the converting expression introduced in the program at compile time, entailing them in different ways would result in different converting expressions to be used in the elaborated program and ultimately in distinct elaborated programs. Figure 6.2 takes this into account and enforces that the conditions can only be entailed in a single way.

Subsection 6.2.1 discusses an algorithm to construct the implicit conversion that yields the appropriate converting expression and enforces that the conditions of each axiom used to construct the conversion can be entailed in exactly one way.

6.2.1 Constructing the Conversion Expression

To compute the converting expressions necessary to elaborate the source program, the user-provided converting expressions (introduced in the conversion axioms) are used.

In order to construct the conversion that satisfies some constraint $\tau_s \rightsquigarrow_{I_1} \tau_t$, a converting path from τ_s to τ_t is computed, almost as described in 6.3.2. Again,

an exception is made in case τ_s and τ_t are the same: in that case, the converting expression is the identity function.

However, a variant (discussed below) of the axiom tester is now used that ensures the conditions can only be entailed in one way and that returns the converting expression associated to the axiom, as highlighted in Figure 6.2.

For each conversion constraint $\tau_s \rightsquigarrow_{I_i} \tau_t$, a converting expression under construction $\lambda x.\ body$ is kept. It is initiated to the identity function ($\lambda x.x$) and, for every necessary conversion axiom $I \vDash_{ax} \tau_1 \rightsquigarrow \tau_2; e$ in the converting path, the lambda abstraction is updated to $\lambda x.(e\ \ body)$.

Upon reaching the target τ_t of the conversion constraint, the converting expression e is no longer under construction and will replace the term variable present in the conversion constraint. The term e will be of the form $\lambda x.(e_n ... (e_1\ \ x)...)$ and have type $\tau_s \to \tau_t$, as required.

Unambiguous Axiom Tester

The Axiom Tester discussed in Section 6.3.3 allows for the conditions in a conversion axiom to be entailed in multiple ways. This is the desired behavior for the algorithms that use it: not using it to compute the paths could cause typing a source program that should be rejected due to its ambiguity: consider $\tau_s \rightsquigarrow_I \tau_t$. Suppose there are two axioms capable of converting τ_s to τ_t: *ax1* and *ax2*. The former has no conditions whereas the latter has a condition that, under I, can be entailed in two different ways. Not allowing ambiguity in the conditions would mean $ax2$ would be disregarded and $ax1$ used, which is undesirable since this program should be considered ambiguous because this constraint can be entailed in not one but rather in three different ways.

However, allowing conditions to be entailed in multiple ways is not desirable for constructing the conversions. If $e : \forall \overline{a}.\overline{j_i : \tau_{\rightsquigarrow}} \Rightarrow \tau_{\rightsquigarrow}$ has conditions that can be entailed in more than one ways, the j_i can be substituted in e by multiple terms (resulting in multiple elaborated programs).

It is straightforward to change this behavior: for each condition, the algorithm to compute the possible paths (discussed Section 6.3.2) is ran and, if there is only one path from the source to the target specified by the constraint, the constraint is entailed by the implicit environment. For the construction of the paths, the unambiguous algorithm shown in Figure 6.2 should be used.

Note that, since the size of the types in the conditions is strictly smaller to the size of the types in the final conversion monotype, the process of entailing each constraint is bound to terminate. For cases in which there are type variables on the left hand side of the conditions, the version described in Section 6.3.4 is used instead.

Furthermore, the unambiguous axiom tester returns the appropriate converting expression to use during elaboration. As such, the place holding variables in the converting expression must be substituted by the expressions specified in the (unique) conversion axiom in the environment that satisfies each condition.

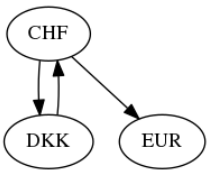


Figure 6.3: Plot of the constraints with type variables. Both are indexed by I_1.

6.3 Computations

This section discusses in detail the computations referenced in Sections 6.1 and 6.2, in the order they appear in the text. While these computations are exponential (in their nature) with respect to the number of implicit axioms applicable to each source type, it is most likely that, in a realistic setting, this number turns out to be very small.

6.3.1 Reachability

To compute the types reachable from a given type τ under a given implicit environment I, a queue (initially consisting of τ) and an accumulator (initially empty) are used. The applicability of each conversion axiom to the head of the queue τ_{head} is checked and, if it is applicable, the types $\overline{\tau_i}$ the axiom can convert τ_{head} to are added to the back of the queue. To check the applicability and compute the $\overline{\tau_i}$'s the axiom tester (6.3.3) which uses the Axiom Conversion Judgment of Figure 6.2.

Once all the axioms have been tried, the head of the queue is moved to the accumulator and the process restarts with the current queue and the entire environment I. When there are no more elements in the queue, all the reachable types are in the accumulator.

6.3.2 Computing the Paths

To compute all the paths from a source type τ_s to a target type τ_t, an algorithm with an accumulator and a queue Q of paths under construction is used. Initially, the accumulator is empty and Q consists of one path with a single type τ_s. The iterative step is to pick the last type $\tau_{j_{n_j}}$ on the first path ($p_j = \tau_{j_1} \to ... \to \tau_{j_{n_j}}$) of the queue and check if it is equal to τ_t. If so, p_j is added to the accumulator. Otherwise, the axiom tester (6.3.3) is applied to $\tau_{j_{n_j}}$ for every axiom in the implicit environment. For each returned type τ_i not present in p_j, a path $p_{j_i} = \tau_{j_1} \to ... \to \tau_{j_{n_j}} \to \tau_i$ is added to the end of Q. After trying every conversion axiom in the implicit environment, p_j is deleted from Q and the process resumes with the new list of paths under construction. When Q becomes empty, the algorithm stops and the accumulator is returned. Note that, since the types added to each path must not already be present in the path, this approach avoids entering loops. As such, under the implicit environment shown in Figure 6.3, the path from DKK to EUR will be correctly computed as DKK $\to$ CHF $\to$ EUR.

6.3.3 Axiom Tester

To be able to assess if a conversion axiom is applicable to some given ground type τ, and thus compute the types it can implicitly convert τ into, multiple cases must be considered.

If the conversion axiom $e : \forall \overline{a}.\overline{(j_i : \tau_{\rightsquigarrow i})} \Rightarrow \tau_s \rightsquigarrow \tau_t$ does not contain type variables, it suffices to check if τ_s equals τ and if the conditions are entailed. If so, it is able to convert τ (only) to τ_t. More generally, if it is possible to unify τ_s with τ resulting in a type substitution Θ and applying Θ to the axiom causes all type to be ground, the same approach is taken.

However, there are more complex cases. Take this conversion axiom presented in Chapter 5: $\lambda x.map\ j\ x : j\ a \rightsquigarrow b \Rightarrow [a] \rightsquigarrow [b]$. For such axioms '$e : \forall \overline{a}.\overline{(j_i : \tau_{\rightsquigarrow i})} \Rightarrow \tau_s \rightsquigarrow \tau_t$', if it is possible to unify τ_s with τ resulting in a type substitution Θ_1, Θ_1 is then applied to the conditions and to τ_t. Afterwards, reachability graphs are constructed for each condition (the types τ_{s_i} are ground by this point, due to the conditions specified in Section 5.1.2), to determine the types that are possible choices for the remaining type variables, *i.e.*, such that the conditions are entailed. This results in (possibly) multiple substitutions, Θ_{i_2}. As such, the axiom can implicitly convert τ to $(\Theta_{i_2} \circ \Theta_1)\tau_t$, for each substitution Θ_{i_2}.

6.3.4 Optimizations

It is clear that some parts of the process described can be optimized in an implementation: instead of computing the reachable types and the possible paths, it is possible to compute the reachability graphs and get the necessary information for both these steps.

Notice how easily the computation of the paths in Section 6.3.2 can be modified to compute all the conversion paths (the reachability graph) from a given type: it suffices to eliminate the equality check of the final type in a path against the (now non-existing) target type and take every partial path $p_j = \tau_{j_1} \rightarrow ... \rightarrow \tau_{j_i}$ from each path $p_j = \tau_{j_1} \rightarrow ... \rightarrow \tau_{j_i} \rightarrow ... \rightarrow \tau_{j_{final}}$ in the accumulator.

Further, the ambiguity check discussed in the second paragraph of Section 6.2 could be moved (for all the constraints involved in "finding the dominator step") to this computation: instead of only keeping the types on the paths, the computation of the paths could be modified to store the axioms used at each step of the way. After taking the partial paths, it could be easily verified whether that two paths exist to the same type τ_{amb}. In that case, not only would the conversion to τ_{amb} be ambiguous as it would also be for any type for which there is a path containing τ_{amb}.

6.4 Properties of TrIC

This section presents the main theorems and conjectures regarding desirable properties of TrIC. It also briefly considers principal types.

6.4.1 Termination of Type Inference

Theorem: TrIC's type inference terminates.
Proof: Consider Figure 5.5. We prove that the computation regarding each premise terminates and thus type inference terminates:

- collection of the constraints terminates as it is syntax directed with respect to the term e and either the size of the terms in the premises is strictly smaller than the e or the premise is trivially entailed;

- the unification of the equality constraints E terminates as the size of E is finite and strictly decreases in each step, considering the size of E to be the sum of the sizes of the constraints, the size of a constraint the sum of the sizes of its two types and the size of a function type $\tau_1 \to \tau_2$ to be equal to the size of τ_1 plus the size of τ_2 plus 1.

- the "dominator" step terminates as each process of computing the paths is finite: both the size of Y and the number of available axioms are finite; it is not possible to enter loops, since repetition of types is avoided in each path; it is not possible to convert from smaller types into bigger types; and the entailment of the conditions terminates as the types in the conditions are strictly smaller than the ones in the final conversion monotype;

- the "conversions" step terminates for the exact sames reasons as for the "dominator" step, as it too is concerned with computing the possible types;

- the last two premises trivially terminate.

6.4.2 Determinism of Type Inference

Theorem: TrIC's type inference is deterministic.
Proof: Again, consider Figure 5.5. We prove that the computation regarding each premise is deterministic and thus so is type inference:

- the collection of constraints is deterministic as it is syntax-directed on the term e;

- the unification step is deterministic as it is syntax-directed on E;

- the determinism of the dominator step stems from its definition;

- the determinism of the "conversion" step is due to the determinism of the dominator and to the way the conversions are unambiguously constructed (Figure 6.2).

6.4.3 Typing Preservation

Conjecture: If $I; \Gamma \vdash_{tm} e : \sigma; e'$ then $\Gamma_F \vdash_F e' : \sigma_F$, where Γ_F and σ_F are, respectively, the translations of the typing environment Γ and of the the type σ to System F. The latter translation of TrIC types to System F types is simply an inclusion mapping. Note that this translation preserves types and that System F has type safety, *i.e.*, progress and preservation (as defined in Section 2.2.4).

6.4.4 Backwards Compatibility

Conjecture: TrIC is backwards compatible with HM, *i.e.*, if $\Gamma \vdash^{HM}_{tm} e : \tau; E$ and $\Theta = unify(E)$ then $\Gamma \vdash^{TrIC}_{tm} e : \tau'; E'; Y'$, $\Theta'_1 = unify(E')$, $\Theta'_2 = dominator(\Theta'_1 Y')$ and $\Theta(\tau) = \Theta'_2 \circ \Theta'_1(\tau')$.

We believe the proof should proceed by structural induction on e.

6.4.5 Principal Types

A type-system has the principal type property if, for any pair of a term e and a typing environment Γ, there is a type σ such that, for any other type σ' that e can be assigned under Γ, σ' is an instance of σ. Under these circumstances, σ is said to be the principal (or most general) type of e.

It is not obvious how to define the principal type of a TrIC term. If, for instance, a conversion from type EUR to type USD is defined, then it would be reasonable to say that EUR is at least as general than USD (note that the converse conversion may also be possible). We have circumvented this issue by using an user-provided annotation and leave the study of principal types for languages with implicit type conversions as future work.

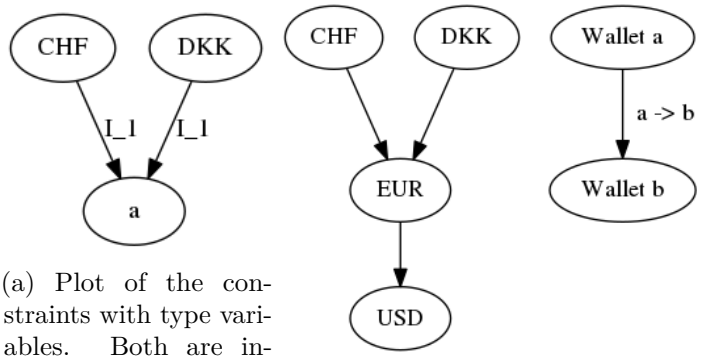
6.5 Example

This section focuses on solving the constraints generated by the program in Example 4. Since this program consists of nested applications, the equality constraint that arises from the type annotation will be disregarded, until further notice, in order to better showcase the graph-based constraint solver.

6.5.1 Substituting Type Variables

Recall that the equality constraints E generated by Example 4 were $\{Float \to \text{CHF} \sim b_1 \to b_2, a \to a \to (\text{Wallet } a) \sim b_3 \to b_4,\ Float \to \text{DKK} \sim b_5 \to b_6,\ b_4 \sim b_7 \to b_8\}$ and the generated conversion constraints Y were $\{Float \rightsquigarrow_{I_1} b_1,\ b_2 \rightsquigarrow_{I_1} b_3,\ Int \rightsquigarrow_{I_2} b_5,\ b_6 \rightsquigarrow_{I_1} b_7\ \}$.

Performing unification on E results in the substitution $\Theta = [b_1 \mapsto Float,\ b_2 \mapsto \text{CHF},\ b_3 \mapsto a,\ b_4 \mapsto a \to \text{Wallet } a,\ b_5 \mapsto Float,\ b_6 \mapsto \text{DKK},\ b_7 \mapsto a,\ b_8 \mapsto \text{Wallet } a]$. The application of Θ to Y yields the updated constraints $Y' = \{(j_1, Float \rightsquigarrow_{I_1} Float),\ (j_2, \text{CHF} \rightsquigarrow_{I_1} a),\ (j_3, Int \rightsquigarrow_{I_2} Float),\ (j_4, \text{DKK} \rightsquigarrow_{I_1} a)\}$.



(a) Plot of the constraints with type variables. Both are indexed by I_1.

(b) The implicit environment I_1.

Figure 6.4

This example results in the very simple graph presented in Figure 6.4a. Both constraints are indexed by the environment $I_1 = \{\lambda x.K_{\text{EUR}}\ 28.0 : \text{DKK} \rightsquigarrow \text{EUR},\ \lambda x.K_{\text{EUR}}\ 1.2 : \text{CHF} \rightsquigarrow \text{EUR},\ \lambda x.K_{\text{USD}}\ 3.4 : \text{EUR} \rightsquigarrow \text{USD},\ (\lambda x.case\ x\ of\ K_{\text{Wallet}}\ y_1\ y_2 \rightarrow K_{\text{Wallet}}(j\ y_1)\ (j\ y_2)) : (\forall a, b.j : a \rightsquigarrow b \Rightarrow \text{Wallet}\ a \rightsquigarrow \text{Wallet}\ b)\}$ (in Figure 6.4b).

Following 6.1.3, the computation of the types reachable from CHF and from DKK results in, respectively, { CHF, EUR, USD } and {DKK, EUR, USD}. The intersection of these sets is { EUR, USD} and, as such all paths from CHF and from DKK to both EUR and USD must be computed. EUR is the dominator since it is present in these paths and, as such, should replace a in all the conversion constraints.

6.5.2 Constructing the Conversions

Entailing a conversion constraint and constructing the corresponding converting expression can (and should) be handled simultaneously. The first constraint $(j_1, Float \rightsquigarrow_{I_1} Float)$ is trivially entailed. As such j_1 should be replaced by the identity function.

Regarding the conversion $(j_2, \text{CHF} \rightsquigarrow_{I_1} \text{EUR})$, note that only one conversion axiom is applicable to the type CHF: $\lambda x.K_{\text{EUR}}\ 1.2$. This axiom has no conditions, so it is applicable and can not introduce ambiguity by itself. The set of paths under construction becomes 'CHF $\rightarrow$ EUR' and is bound to the converting expression $\lambda x.K_{\text{EUR}}\ 1.2$. Because EUR is the target type of the constraint, this path is not extended; and since it is the only path under construction, the implicit conversion is not ambiguous. Hence, j_2 will be replaced by $\lambda x.K_{\text{EUR}}\ 1.2$.

The constraints $(j_3, Int \rightsquigarrow_{I_2} Float)$ and $(j_4, \text{DKK} \rightsquigarrow_{I_1} EUR)$ are treated in a completely identical manner. The place-holders j_3 and j_4 are replaced by, respectively, $\lambda x.3.1415$ and $\lambda x.K_{\text{EUR}}\ 28.0$.

Applying these substitutions to the partially elaborated program (shown in Figure 5.7) results in the fully elaborated program presented in Figure 6.5.

$$
\begin{array}{l}
\mathit{data}\ \mathrm{EUR} = K_{\mathrm{EUR}}\ \mathit{Float};\\
\mathit{data}\ \mathrm{CHF} = K_{\mathrm{EUR}}\ \mathit{Float};;\\
\mathit{data}\ \mathrm{DKK} = K_{\mathrm{DKK}}\ \mathit{Float};\\
\mathit{data}\ \mathrm{USD} = K_{\mathrm{USD}}\ \mathit{Float};\\
\mathit{data}\ \mathrm{Wallet}\ a = K_{\mathrm{Wallet}}\ a\ a;\\
\\
(K_{\mathrm{Wallet}}\\
\quad ((\lambda x.K_{\mathrm{EUR}}\ 1.2)\ (K_{\mathrm{CHF}}\ ((\lambda x.x)\ 5.25)))\\
\quad ((\lambda x.K_{\mathrm{EUR}}\ 28.0)\ (K_{\mathrm{DKK}}\ ((\lambda x.3.1415)\ 1))))
\end{array}
$$

Figure 6.5: Partial elaboration of the TrIC program in Example 4.

Slightly More Complex Conversions

The conversion constraints generated were quite simple. More interesting constraints to consider are $(j_l, \mathrm{CHF} \rightsquigarrow_{I_1} \mathrm{USD})$, $(j_m, \mathrm{Wallet\ CHF} \rightsquigarrow_{I_1} \mathrm{Wallet\ EUR})$ and $(j_n, \mathrm{Wallet\ CHF} \rightsquigarrow_{I_1} \mathrm{Wallet\ USD})$. The idea is always the same: starting from the source type, conversion axioms are applied in order to reach the target type.

Regarding the first, only one axiom matches its source type: $(\lambda x.K_{\mathrm{EUR}}\ 1.2) : \mathrm{CHF} \rightsquigarrow \mathrm{EUR}$. As before, the set of paths under construction becomes $\mathrm{CHF} \rightarrow \mathrm{EUR}$ and is bound to the converting expression $\lambda x.((\lambda x_1.K_{\mathrm{EUR}}\ 1.2)\ x)$. Again, only one axiom applies to the last type of the path under construction: $(\lambda x.K_{\mathrm{USD}}\ 3.4) : \mathrm{EUR} \rightsquigarrow \mathrm{USD}$, resulting in an updated path $(\mathrm{CHF} \rightarrow \mathrm{EUR} \rightarrow \mathrm{USD})$ and corresponding expression: $\lambda x.((\lambda x_1.K_{\mathrm{USD}}\ 3.4)\ ((\lambda x_2.K_{\mathrm{EUR}}\ 1.2)\ x))$. We have arrived at the target type of the constraint, so there is a path; as there is solely one path, this constraint is unambiguously entailed, and the corresponding converting expression has been computed.

As for the constraint bound to j_m, Wallet CHF can be made to match the source type of only one axiom:

- $(\lambda x.\mathit{case}\ x\ \mathit{of}\ K_{\mathrm{Wallet}}\ y_1\ y_2 \rightarrow K_{\mathrm{Wallet}}(j\ y_1)\ (j\ y_2)) : (\forall a, b.j : a \rightsquigarrow b \Rightarrow \mathrm{Wallet}\ a \rightsquigarrow \mathrm{Wallet}\ b)$

Unifying type Wallet CHF with Wallet a and applying the resulting type substitution to the type of the axiom yields $\forall b.j : \mathit{CHF} \rightsquigarrow b \Rightarrow \mathrm{Wallet\ CHF} \rightsquigarrow \mathrm{Wallet}\ b$.

As stated in Section 6.2.1, each type for which there is a conversion $\mathrm{CHF} \rightsquigarrow_{I_1} \tau$ will now be considered. Figure 6.4b shows that b can (apart from the trivial conversion) be substituted by EUR and USD, and the converting expressions are, respectively, "$\lambda x.K_{\mathrm{EUR}}\ 1.2$" and "$(\lambda x.K_{\mathrm{USD}}\ 3.4)\ (\lambda x.K_{\mathrm{EUR}}\ 1.2)$". As such, two instances of the parametric axiom to be applicable:

- $(\lambda x.\mathit{case}\ x\ \mathit{of}\ K_{\mathrm{Wallet}}\ y_1\ y_2 \rightarrow$
  $K_{\mathrm{Wallet}}((\lambda x.K_{\mathrm{EUR}}\ 1.2)\ y_1)\ ((\lambda x.K_{\mathrm{EUR}}\ 1.2)\ y_2))$
  $: (\mathrm{Wallet\ CHF} \rightsquigarrow \mathrm{Wallet\ EUR});$

- $(\lambda x.case\ x\ of\ K_{\text{Wallet}}\ y_1\ y_2 \rightarrow$
  $K_{\text{Wallet}}(((\lambda x.K_{\text{USD}}\ 3.4)\ (\lambda x.K_{\text{EUR}}\ 1.2))\ y_1)\ (((\lambda x.K_{\text{USD}}\ 3.4)\ (\lambda x.K_{\text{EUR}}\ 1.2))\ y_2))$
  $:$ (Wallet CHF $\rightsquigarrow$ Wallet USD)

This results in two paths under construction: 'Wallet CHF $\rightarrow$ Wallet EUR' and 'Wallet CHF $\rightarrow$ Wallet USD'. The last type of the first path is the target type of the constraint, thus the constraint is entailable and "$\lambda x.(\lambda x_1.case\ x_1\ of\ K_{\text{Wallet}}\ y_1\ y_2 \rightarrow$ $K_{\text{Wallet}}((\lambda x_2.K_{\text{EUR}}\ 1.2)\ y_1)\ ((\lambda x_3.K_{\text{EUR}}\ 1.2)\ y_2))\ x$" is a possible converting expression.

However, the other path under construction must also be considered. If it too leads to Wallet EUR, the constraint can be entailed in multiple ways. The parametric axiom is once again the only that needs to be considered. However, since it is not possible to implicitly convert from USD to any other type, the axiom can not be applied and the path is disregarded.

Finally, consider $(j_n, \text{Wallet CHF} \rightsquigarrow_{I_1} \text{Wallet USD})$. The process to check the entailability of this conversion starts exactly as the previous and results in the same two paths under construction and corresponding converting expressions. As one of the paths ends in the target type of the conversion, we know it to be entailable. Nonetheless, we must continue to explore the other. The parametric axiom can be applied to Wallet EUR to convert it to Wallet USD, *i.e.*, the path becomes 'Wallet CHF $\rightarrow$ Wallet EUR $\rightarrow$ Wallet USD'. There are two possible paths: 'Wallet CHF $\rightarrow$ Wallet USD' and 'Wallet CHF $\rightarrow$ Wallet EUR $\rightarrow$ Wallet USD'. As such, the constraint is not unambiguously entailed under the current implicit environment, causing the program to be rejected.

Chapter 7

TrICx

This chapter discusses the interaction of ITC with type classes, one of Haskell's core features: it presents TrICx, an extension of TrIC with type classes. In fact, the implementation provided with this thesis enables the use of type classes (since it is an extension of KHC, which supports them).

The chapter starts by introducing type classes (Section 7.1). Then, in a similar order to Chapter 5's, discusses TrICx syntax, typing rules, type inference and constraint generation (Sections 7.2, 7.3 and 7.4). Afterwards, constraint entailment is addressed (Section 7.5) and the translation of a TrICx program is shown (Section 7.6). The chapter ends with a discussion of an alternative approach to constraint entailment (Section 7.7).

7.1 Brief Overview of Type Classes

As opposed to parametric polymorphism, type classes ([25]) are Haskell's approach to support ad-hoc polymorphism [22], which is characterized by having a function defined over multiple types and acting in distinct ways for different types.

Type classes specify a set of signatures (each a pair of a name and a type abstracted over a type variable) of functions that must be defined for its members.A possible class declaration for a new type class *Currency* could be translated to English as "to belong to class *Currency*, type a must have defined functions *earn* and *spend*, both of type $a \to a$'.

In this case, the assertion *Currency* EUR would state the existence of functions *earn* and *spend* of type EUR $\to$ EUR.

An instance declaration corresponds to an axiom. A *simple axiom* declares a type to be a member of a type class and specifies implementations for the functions required by the type class. An instance declaration for *Currency* EUR implements functions *earn* and *spend* and declares *Currency* EUR to be an axiom.

It is also possible to specify instance declarations for abstract types by requiring (abstract) conditions to hold. One example with the *Equality* type class is an instance declaration that states "if there is an instance of *Equality* for some type a, then there is an instance for *Equality* for type $[a]$". This declarations introduces the *logical*

implication axiom "*Equality a* implies *Equality* [*a*]". Logical implication axioms specify class constraints and a final simple axiom.

7.2 Syntax

Figure 7.1 presents the syntax for a language that extends upon TrIC by supporting type classes (TrICx, for TrIC eXtended). New syntax is introduced for class and instance declarations, for types and for the novel (with respect to this text) program theory.

Classes consist of a name, a type variable and method. Unlike type classes in the GHC, TrICx does not support (for the sake of brevity of exposure) multiple type variables, multiple methods nor superclasses: none of these extensions poses a problem to the ideas specified in this chapter. Class declarations mirror these design choices and consist of a name (TC) for the class, a type variable a, a name for the method f and its type signature σ.

An instance declaration consists of an axiom, consisting of a (possibly empty) set of class constraints, the class name TC and the type τ of the axiom it introduces; and of an implementation of the TC's method f (which must comply with the type signature in the corresponding class declaration). Class constraints' syntax is identical to the syntax of a simple axiom: a class name and a type. The class constraints in a logical implication axiom are known as logical assumptions.

In addition to TrIC's monotypes and polytypes, TrICx has qualified types ρ [8, 9]. These consist of a number of class constraints and a monotype. The definition of polytypes has changed to include universal quantification over qualified types.

Finally, in addition to the typing environment Γ and the implicit environment I there is now a (global) program theory P, which is the set of the axioms introduced by the instance declarations of the program.

7.3 Typing

The typing rules for a TrICx program are presented in Figure 7.2, where the differences with respect to TrIC have been highlighted. These rules are similar to the ones for TrIC (Figure 5.2) with the addition that class and instance declarations affect, respectively, the typing environment and the program theory under which the program's term is typed.

The typing of class and instance declarations is shown in Figure 7.3. Class declarations extend the typing environment by binding the function's name to its specified type. Instance declarations extend the program theory P with the triplet of logical assumptions, class name TC and type τ it specifies, as long as there has been a class declaration for TC and the method implementation complies with the method's type specified in the class declaration.

The declarative rules for term typing are visible in Figure 7.4. When compared with the typing rules for TrIC (Figure 5.6), the presence of two new rules is observable: Constraint Introduction and Constraint Elimination. The first states that,

$$
\begin{array}{lr}
pgm ::= \overline{decl}; (e : \sigma) & \textit{Program} \\
decl ::= data \mid class \mid inst & \\
data ::= data\ T\ a = \overline{K_i\ \ \overline{\tau_{i_j}}} & \textit{data declaration} \\
class ::= class\ TC\ a\ \textbf{where}\ \{f :: \sigma\} & \textit{class declaration} \\
inst ::= instance\ \overline{TC_i\ \tau_i} \Rightarrow TC\ \tau\ \textbf{where}\ \{f = e\} & \textit{instance declaration} \\
\\
\tau ::= a \mid \tau_1 \to \tau_2 \mid T\ \overline{\tau} & \textit{monotypes} \\
\rho ::= \overline{TC_i\ \tau_i} \Rightarrow \tau & \textit{qualified types} \\
\sigma ::= \rho \mid \forall a.\sigma & \textit{type schemes} \\
\\
\tau_{\rightsquigarrow} ::= \tau \rightsquigarrow \tau & \textit{Conversion Monotypes} \\
\sigma_{\rightsquigarrow} ::= \forall \overline{a}.\overline{(j : \tau_{\rightsquigarrow})} \Rightarrow \tau_{\rightsquigarrow} & \textit{Conversion Polytypes} \\
\\
e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid K \mid case\ e_1\ of\ \overline{(K_i \overline{x_{i_j}}) \to e_{i_2}} & \textit{Terms} \\
\quad \mid let\ x : \sigma\ =\ e_1\ in\ e_2 \mid locimp\ e_1 : \sigma_{\rightsquigarrow}\ in\ e_2 & \\
\\
I ::= \overline{e : \sigma_{\rightsquigarrow}} & \textit{Implicit Environment} \\
\\
\Gamma ::= \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, K : \sigma \mid \Gamma, T & \textit{Typing Environment} \\
\\
Cls ::= \overline{TC_i\ \tau_i} \Rightarrow TC\ \tau & \textit{Constraint Scheme} \\
P ::= \overline{Cls} & \textit{Theory Environment}
\end{array}
$$

Figure 7.1: TrICx's syntax

if under the program theory P extended with the axiom $TC\ \tau_{ax}$, term e has type τ, then under P, it has the qualified type $TC\ \tau_{ax} \Rightarrow \tau$; the second states that if the program theory P entails a class constraint present in the (qualified) type of some term e, then that constraint can be removed from the type of e, under P.

7.4 Type Inference and Collecting Constraints

Type inference and the collection of the constraints is similar to those presented in Chapter 5. Figure 7.5 presents the top-level rule, which is identical to Figure 5.5 except for the class constraints and their entailment (the fourth premise).

The generation of equality and conversion constraints is the same as TrIC's. As for the collection of class constraints C, it follows an intuitive and standard approach of collecting them from the typing environment Γ.

According to the VAR rule in Figure 7.6, whenever a term variable x has a type

$\boxed{P; I; \Gamma \vdash_p pgm : \tau}$ Program Typing

$$\frac{\Gamma \vdash_{data} data : \Gamma' \qquad P; I; \Gamma' \vdash_p pgm : \tau}{P; I; \Gamma \vdash_p data; pgm : \tau}\ \textsc{Data}$$

$$\frac{\Gamma \vdash_{class} class : \Gamma' \qquad P; I; \Gamma' \vdash_p pgm : \tau}{P; I; \Gamma \vdash_p class; pgm : \tau}\ \textsc{Class}$$

$$\frac{P; \Gamma \vdash_{inst} inst : P' \qquad P'; I; \Gamma \vdash_p pgm : \tau}{P; I; \Gamma \vdash_p inst; pgm : \tau}\ \textsc{Instance}$$

$$\frac{I; \Gamma \vdash_{tm} e : \tau}{P; I; \Gamma \vdash_p e : \tau}\ \textsc{Expression}$$

Figure 7.2: Program Typing

$\boxed{\Gamma \vdash_{data} data : \Gamma'}$ Data Declaration Typing

$$\frac{}{\Gamma \vdash_{data} (data\ T\ a = K\ \overline{\tau}) : \Gamma, T, (K : \forall a.\overline{\tau} \to T\ a)}$$

$\boxed{\Gamma \vdash_{class} class : \Gamma'}$ Class Declaration Typing

$$\frac{\bullet \vdash_{ty} \sigma}{\Gamma \vdash_{class} class\, TC\, a\ \textbf{where}\ \{f :: \sigma\} : \Gamma, f : \forall a.TC\, a \Rightarrow \sigma}$$

$\boxed{P; \Gamma \vdash_{inst} inst : P'}$ Instance Declaration Typing

$$\frac{\bullet \vdash_{ty} \tau \qquad class\, TC\, a\ \textbf{where}\ \{f :: \sigma\} \qquad P; I; \Gamma \vdash_p e : [a \mapsto \tau]\sigma}{P; \Gamma \vdash_{inst} instance\ \overline{TC_i\ \tau_i} \Rightarrow TC\, \tau\ \textbf{where}\ \{f = e\} : P, \overline{TC_i\ \tau_i} \Rightarrow TC\, \tau}$$

Figure 7.3: Declaration Typing

$\boxed{P; I; \Gamma \vdash_{tm} e : \sigma}$ Term Typing

$$\frac{(x : \sigma) \in \Gamma}{P; I; \Gamma \vdash_{tm} x : \sigma}\ \textsc{Var} \qquad \frac{(K : \sigma) \in \Gamma}{P; I; \Gamma \vdash_{tm} K : \sigma}\ \textsc{Constr}$$

$$\frac{P; I; \Gamma \vdash_{tm} e_1 : \tau_{arg} \to \tau_2 \qquad P; I; \Gamma \vdash_{tm} e_2 : \tau_1 \qquad I \vDash_{th} \tau_{arg} \rightsquigarrow \tau_2}{P; I; \Gamma \vdash_{tm} e_1\ e_2 : \tau_2}\ \textsc{TmApp}$$

$$\frac{P; I; \Gamma, x : \tau_1 \vdash_{tm} e : \tau_2}{P; I; \Gamma \vdash_{tm} \lambda x.e : \tau_1 \to \tau_2}\ \textsc{TmAbs}$$

$$\frac{P; I; \Gamma, x : \sigma_1 \vdash_{tm} e_1 : \sigma_1 \qquad P; I; \Gamma, x : \sigma_1 \vdash_{tm} e_2 : \tau_2}{P; I; \Gamma \vdash_{tm} let\ x : \sigma_1 = e_1\ in\ e_2 : \tau_2}\ \textsc{TmLet}$$

$$\frac{\begin{array}{c}\overline{(K_i : \forall a.\overline{\tau_{i_k}} \to T\ a) \in \Gamma} \\ P; I; \Gamma \vdash_{tm} e_1 : \tau_1 \qquad \overline{P; I; \Gamma, \overline{x_{i_j}} : [a \mapsto b]\overline{\tau_{i_k}} \vdash_{tm} e_i : \tau_i} \\ I \vDash_{th} \tau_1 \rightsquigarrow T\ b \qquad \overline{I \vDash_{th} \tau_i \rightsquigarrow \tau_2} \qquad fresh\ b\end{array}}{P; I; \Gamma \vdash_{tm} case\ e_1\ of\ \overline{(K_i\ \overline{x_{i_j}}) \to e_i} : \tau_2}\ \textsc{TmCase}$$

$$\frac{P; \bullet; \Gamma, \overline{a}, \overline{j_i : \tau_{\sim_i}} \vdash_{tm} e : \tau_{\rightsquigarrow} \qquad P; I, i; \Gamma \vdash_{tm} e_2 : \tau_2}{P; I; \Gamma \vdash_{tm} (locimp\ \forall \overline{a}.\overline{j_i : \tau_{\sim_i}} \Rightarrow \tau_{\rightsquigarrow} : i\ in\ e_1 e_2) : \tau_2}\ \textsc{TmLocimp}$$

$$\frac{P, Cls\,; I; \Gamma \vdash_{tm} e : \tau}{P\,; I; \Gamma \vdash_{tm} e : Cls \Rightarrow \tau}\ \textsc{Constraint Introduction}$$

$$\frac{P\,; I; \Gamma \vdash_{tm} e : Cls \Rightarrow \tau \qquad P \vDash Cls}{P\,; I; \Gamma \vdash_{tm} e : \tau}\ \textsc{Constraint Elimination}$$

$$\frac{P; I; \Gamma, a \vdash_{tm} e : \sigma}{P; I; \Gamma \vdash_{tm} e : \forall a.\sigma}\ \textsc{Forall Introduction}$$

$$\frac{P; I; \Gamma \vdash_{tm} e : \forall a.\sigma \qquad \Gamma \vdash_{ty} \tau}{P; I; \Gamma \vdash_{tm} e : [a \mapsto \tau]\sigma}\ \textsc{Forall Elimination}$$

Figure 7.4: Term Typing

$\boxed{P; I; \Gamma \vdash_{tm} (e : \forall \overline{a}.C' \Rightarrow \tau) : \sigma; e'}$

$$\frac{\begin{array}{c} P; I; \Gamma, \overline{a} \vdash_{tm} e : \tau'; E; Y; C; e' \qquad \Theta_E = unify(\overline{a}; E, \tau' \sim \tau) \\ \Theta_Y = dominator(\Theta_E\, Y) \qquad \varphi = conversions(\Theta_Y(\Theta_E Y)) \\ P \vDash_{cts} (\Theta_Y(\Theta_E C)) \hookrightarrow C' \qquad fv(\Theta_Y(\Theta_E(\tau'))) \subseteq \overline{a} \qquad (\Theta_Y(\Theta_E(\tau'))) = \tau \end{array}}{P; I; \Gamma \vdash_{tm} e : \forall \overline{a}.C' \Rightarrow \tau; \varphi(e')}$$

Figure 7.5: TrICx's top-level inference judgment

$\boxed{P; I; \Gamma \vdash_{tm} e : \tau; E; Y; C; e'}$ Type Inference and Partial Translation

$$\frac{(x : \forall \overline{a}.\overline{C_j \Rightarrow} \tau) \in \Gamma \qquad \overline{b}\ fresh}{P; I; \Gamma \vdash_{tm} x : [\overline{a} \mapsto \overline{b}]\tau; \bullet; \bullet; \overline{[\overline{a} \mapsto \overline{b}]C_j}; x}\ \textsc{Var} \qquad \frac{(K : \forall a.\tau) \in \Gamma}{P; I; \Gamma \vdash_{tm} K : [a \mapsto \tau']\tau; \bullet; \bullet; \bullet; K}\ \textsc{Constr}$$

$$\frac{P; I; \Gamma, x : a \vdash_{tm} e : \tau; E; Y; C; e'}{P; I; \Gamma \vdash_{tm} \lambda x.e : a \to \tau; E; Y; C; e'}\ \textsc{Abstraction}$$

$$\frac{\begin{array}{c} P; I; \Gamma \vdash_{tm} e_1 : \tau_1; E_1; Y_1; C_1; e_1' \\ P; I; \Gamma \vdash_{tm} e_2 : \tau_2; E_2; Y_2; C_2; e_2' \qquad fresh\ a,\ b,\ j \end{array}}{P; I; \Gamma \vdash_{tm} e_1 e_2 : a; E_1, E_2, (\tau_1 \sim b \to a); Y_1, Y_2, (j, \tau_2 \rightsquigarrow_I b); C_1, C_2; e_1'(j\ e_2')}\ \textsc{Application}$$

$$\frac{\begin{array}{c} \overline{(K_i : \forall a.\overline{\tau_{i_k}} \to T\ a) \in \Gamma} \qquad \overline{P; I; \Gamma, \overline{x_{i_j}} : [a \mapsto b]\overline{\tau_{i_k}} \vdash_{tm} e_{2i} : \tau_i; E_i; Y_i; C_i; e_{2i}'} \\ P; I; \Gamma \vdash_{tm} e_1 : \tau_1; E_1; Y_1; C_1; e_1' \qquad fresh\ b,\ d,\ j_1,\ \overline{j_i} \\ E = E_1, \overline{E_i} \qquad Y = Y_1, (j_1, \tau_1 \rightsquigarrow_I T\ b), \overline{Y_i}, \overline{(j_i, \tau_i \rightsquigarrow_I d)} \qquad C = C_1, \overline{C_i} \end{array}}{P; I; \Gamma \vdash_{tm} case\ e_1\ of\ \overline{(K_i\ \overline{x_{i_j}}) \to e_{2i}} : d; E; Y; C; case\ (j\ e_1')\ of\ \overline{(K_i'\ \overline{x_{i_j}'}) \to j_i\ e_{2i}'}}\ \textsc{Case}$$

$$\frac{P; I; \Gamma \vdash_{tm} e_1 : \tau_1; e_1' \qquad P; I; \Gamma, x : \sigma_1 \vdash_{tm} e_2 : \tau_2; E_2; Y_2; C_2; e_2'}{P; I; \Gamma \vdash_{tm} let\ x : \sigma_1 = e_1\ in\ e_2 : a; E_1, E_2; Y_1, Y_2; C_1, C_2; let : \sigma_1\ x = e_1'\ in\ e_2'}\ \textsc{Let}$$

$$\frac{\begin{array}{c} P; I; \Gamma, \overline{j_i : \tau_{s_i} \to \tau_{t_i}} \vdash_{tm} e_1 : \forall \overline{a}.\tau; e_1' \\ unify(fv(\tau_{source} \to \tau_{target}); \tau \sim \tau_{source} \to \tau_{target}) = \theta \\ P; I, e_1 : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_{source} \rightsquigarrow \tau_{target}; \Gamma \vdash_{tm} e_2 : \tau; E; Y; C; e_2' \end{array}}{P; I; \Gamma \vdash_{tm} (locimp\ e_1 : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_{source} \rightsquigarrow \tau_{target}\ in\ e_2) : \tau; E; Y; C; e_2'}\ \textsc{LocImp}$$

Figure 7.6: TrICx Type Inference and Partial Translation

$\boxed{P \vDash_{cts} \overline{TC_i\ \tau_i} \hookrightarrow \overline{TC_j\ \tau_j}}$ Constraint Set Solving

$$\frac{\nexists\, (TC_i\ \tau_i \in C) : P \vDash_{ct} TC_i\ \tau_i \hookrightarrow \overline{TC_j\ \tau_j}}{P \vDash_{cts} C \hookrightarrow C}\ \textsc{CtsStop}$$

$$\frac{P \vDash_{ct} TC_i\ \tau_i \hookrightarrow C_3 \qquad P \vDash_{cts} C_1, C_2, C_3 \hookrightarrow C_4}{P \vDash_{cts} C_1, TC_i\ \tau_i, C_2 \hookrightarrow C_4}\ \textsc{CtsStep}$$

$\boxed{P \vDash_{ct} TC\ \tau\ \hookrightarrow \overline{TC_i\ \tau_i}}$ Single Constraint Solving

$$\frac{(\forall \overline{a}.\overline{TC_i\ \tau_i} \Rightarrow TC\ \tau_2) \in P \qquad \tau_1 = [fv(\tau_2) \mapsto \overline{\tau_j}]\tau_2}{P \vDash_{ct} TC\ \tau_1 \hookrightarrow [fv(\tau_2) \mapsto \overline{\tau_j}](\overline{TC_i\ \tau_i})}\ \textsc{Constraint}$$

Figure 7.7: TrICx Type Class Constraints Solving

scheme that specifies class constraints (the binding of such variables occurs during the typing of class declarations), these are collected according to the ground type x has been assigned.

The other rules are identical to the ones shown in Figure 5.6 and, regarding the class constraints, they simply accumulate the constraints from their subterms.

7.5 Solving the Constraints

TrICx solves the class constraints after the equality and conversion constraints. This simplifies the solving process and ensures type classes do not influence type inference by isolating the different features of the language.

The first step is, as for TrIC, to solve the equality constraints by performing unification on them. The resulting type substitution is then applied to both the conversion and class constraints.

Then, the conversion constraints are entailed exactly as specified in Chapter 6. The type substitution resulting from the dominator step are then applied to the class constraints, which, in turn are (recursively) entailed in the standard way for a HM-based language with type classes, as shown in Figure 7.7. The letter C is used simply for readability and stands for a set of class constraints.

In order to entail a class constraint $TC\ \tau$, the program theory P is traversed in search of an axiom $\overline{TC_{i_j}\ \tau_{i_j}} \Rightarrow TC_i\ \tau_i$ such that its head $TC_i\ \tau_i$ can be unified with $TC\ \tau$. Since at most one applies (overlapping instances are not allowed, which is a common restriction, enforced, for example, in the GHC), no backtracking is necessary, and thus a class constraint is entailable if and only if the logical assumptions $\overline{TC_{i_j}\ \tau_{i_j}}$ are entailable. Keep in mind that each of these is simply a class constraint.

Rules CtsStop and CtsStep simplify as far as possible the set of class constraints generated during traversal of the term.

7.6 Example

$$
\begin{array}{l}
\mathit{data}\ \mathrm{DKK} = K_{\mathrm{DKK}}\ \mathit{Float};\\
\mathit{data}\ \mathrm{CHF} = K_{\mathrm{CHF}}\ \mathit{Float};\\
\mathit{data}\ \mathrm{EUR} = K_{\mathrm{EUR}}\ \mathit{Float};\\
\mathit{data}\ \mathrm{Wallet}\ a = K_{\mathrm{Wallet}}\ a\ a;\\
\\
\mathit{class}\ \mathit{Total}\ a\ \mathit{where}\\
\quad \mathit{total} :: a \to \mathit{EUR}\\
\\
\mathit{instance}\ \mathit{Total}\ (\mathrm{Wallet\ EUR})\ \mathit{where}\\
\quad \mathit{total} = \lambda x.\mathit{case}\ x\ \mathit{of}\ K_{\mathrm{Wallet}}\ x_1\ x_2 \to K_{\mathrm{EUR}}(x_1 + x_2)\\
\\
\mathit{locimp}\ \lambda x.K_{\mathrm{EUR}}\ 3.2 : (\mathrm{DKK} \to \mathrm{EUR})\ \mathit{in}\\
\mathit{locimp}\ \lambda x.\mathit{case}\ x\ \mathit{of}\ K_{\mathrm{Wallet}}\ x_1\ x_2 \to K_{\mathrm{Wallet}}(j\ x_1)\ (j\ x_2)\\
: \forall a, b.j : a \rightsquigarrow b \Rightarrow \mathrm{Wallet}\ a \rightsquigarrow \mathrm{Wallet}\ \mathit{in}\\
\mathit{total}\ (\mathit{locimp}\ \lambda x.K_{\mathrm{EUR}}\ 5.0 : (\mathrm{CHF} \to \mathrm{EUR})\ \mathit{in}\\
\quad (K_{\mathrm{Wallet}}\ (K_{\mathrm{DKK}}\ 6.3)\ (K_{\mathrm{CHF}}\ 3.1415)))\\
: * : \mathrm{EUR}
\end{array}
$$

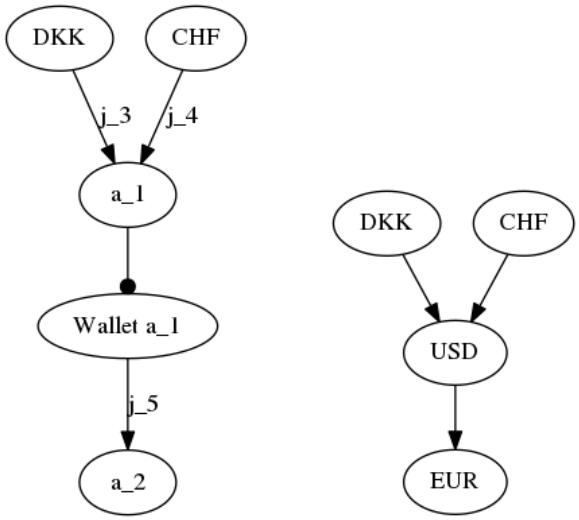
Example 5: A TrICx program

Consider the program in Example 5. The collected equality constraints are $\{\mathit{Float} \to \mathrm{DKK} \sim b_1 \to b_2, \mathit{Float} \to \mathrm{CHF} \sim b_3 \to b_4, a_1 \to a_1 \to \mathrm{Wallet}\ a_1 \sim b_5 \to b_6, b_6 \sim b_7 \to b_8, a_2 \to \mathrm{EUR} \sim b_9 \to b_{10}\}$; and the set of conversion constraints is: $\{(j_1, \mathit{Float} \rightsquigarrow_{I_2} b_1),\ (j_2, \mathit{Float} \rightsquigarrow_{I_2} b_3),\ (j_3, b_2 \rightsquigarrow_{I_2} b_5),\ (j_4, I_2 \vDash_{ct} b_2 : b_7 \rightsquigarrow\ ),\ (j_5, b_8 \rightsquigarrow_{I_1} b_9)\ \}$, with $I_1 = \{(\lambda x.K_{\mathrm{EUR}}\ 3.2 : \mathrm{DKK} \to \mathrm{EUR}, \lambda x.\mathit{case}\ x\ \mathit{of}\ K_{\mathrm{Wallet}}\ x_1\ x_2 \to K_{\mathrm{Wallet}}(j\ x_1)\ (j\ x_2)) : (\forall a, b.j : a \rightsquigarrow b \Rightarrow \mathrm{Wallet}\ a \rightsquigarrow \mathrm{Wallet})\}$ and $I_2 = I_1, \{\lambda x.K_{\mathrm{EUR}}\ 5.0 : \mathrm{CHF} \to \mathrm{EUR}\}$. The only class constraint is $\{\mathit{Total}\ a_2\}$. Finally, the equality constraint resulting from the user-provided annotation is $\{\mathrm{EUR} \sim b_{10}\}$.

Performing unification on the equality constraints yields the substitution $\Theta = [b_1 \mapsto \mathit{Float},\ b_2 \mapsto \mathrm{DKK},\ b_3 \mapsto \mathit{Float},\ b_4 \mapsto \mathrm{CHF},\ b_5 \mapsto a_1,\ b_6 \mapsto a_1 \to \mathrm{Wallet}\ a_1,\ b_7 \mapsto a_1,\ b_8 \mapsto \mathrm{Wallet}\ a_1,\ b_9 \mapsto a_2,\ b_{10} \mapsto \mathrm{EUR}]$.

Applying Θ to the conversion and class constraints results in $\{(j_1, \mathit{Float} \rightsquigarrow_{I_2} \mathit{Float}), (j_2, \mathit{Float} \rightsquigarrow_{I_2} \mathit{Float}), (j_3, \mathrm{DKK} \rightsquigarrow_{I_2} a_1), (j_4, \mathrm{CHF} \rightsquigarrow_{I_2} a_1), (j_5, \mathrm{Wallet}\ a_1 \rightsquigarrow_{I_1} a_2)\ \}$ and $\{\mathit{Total}\ a_2\}$, respectively.

Afterwards, constraints with type variables are plotted (see Chapter 6) resulting in the graph shown in Figure 7.8a, where the edge with the full circle symbolizes the partial order on the components: the component with the circle has lower priority. The conversions bound to j_3 and j_4 take precedence in relation to one bound to j_5, since source type in the latter contains a type variable present as the target type of the former. Computing the dominator of constraints bound to j_3 and j_4 results in a_1 being substituted by EUR. Then, Wallet EUR is found to be the appropriate substitute for a_2. The subsequent process of constructing the conversion



(a) Constraints with type variables from Example 5

(b) Alternative Implicit Environment

Figure 7.8

$$
\begin{aligned}
&\textit{data}\ \text{DKK} = K_{\text{DKK}}\ \textit{Float};\\
&\textit{data}\ \text{CHF} = K_{\text{CHF}}\ \textit{Float};\\
&\textit{data}\ \text{EUR} = K_{\text{EUR}}\ \textit{Float};\\
&\textit{data}\ \text{Wallet}\ a = K_{\text{Wallet}}\ a\ a;\\
&\\
&\textit{class}\ \textit{Total}\ a\ \textit{where}\\
&\quad \textit{total} :: a \to \textit{EUR}\\
&\\
&\textit{instance}\ \textit{Total}\ (\text{Wallet EUR})\ \textit{where}\\
&\quad \textit{total} = \lambda x.\textit{case}\ x\ \textit{of}\ K_{\text{Wallet}}\ x_1\ x_2 \to K_{\text{EUR}}(x_1 + x_2)\\
&\\
&\textit{total}\ (K_{\text{Wallet}}\ ((\lambda x.K_{\text{EUR}}\ 3.2)(K_{\text{DKK}}\ 6.3))\ ((\lambda x.K_{\text{EUR}}\ 5.0)(K_{\text{CHF}}\ 3.1415)))
\end{aligned}
$$

Figure 7.9: A more complex TrIC program

and substituting the place holding variables in the program is completely identical to the one described in Chapter 6.

Applying these substitutions to the class constraint(s), results in $\{\textit{Total}$ Wallet EUR$\}$. This constraint is trivially entailed by the program theory $P = \{\textit{Total}$ Wallet EUR$\}$.

The translation of the program in Example 5 into the subset of TrICx without implicit conversions (which is also a subset of Haskell) is shown in Figure 7.9.

7.7 Alternative Design

This approach to solve each sort of constraint separately will cause rejection of some programs that could, due to their class constraints, only be corrected in one way. Because these constraints are invisible to our conversion constraint solver, it may (correctly) find the conversion constraints ambiguous and as such reject the program.

Alternatively, more programs will be elaborated unambiguously and thus accepted if the class and conversion constraints are entailed simultaneously. When trying to resolve what type to instantiate a type variable a with in the grounding of the implicit conversions, the first step is, as before, to compute the common types, *i.e.* reachable from every source type. Afterwards, only the types that satisfy all the class constraints placed upon a are eligible to substitute a. The algorithm to instantiate a then proceeds as before.

As a practical example, consider the program in Example 5, and the alternative implicit environment specified in Figure 7.8b. The approach presented in Section 7.5 would entail the equality and conversion constraints and assign type Wallet USD to the argument of *total*, resulting in unsatisfiable class constraints. On the other hand, the approach of this Section would succeed because it would convert both subterms of the Wallet constructor to EUR, since Wallet USD would be immediately disqualified from being the dominator for not being an instance of *Total*.

However, we have opted by Section 7.5 because it allows for type classes to remain a feature orthogonal to type inference, as in both the GHC and the KHC.

Chapter 8

Related work

This chapter positions TrIC and TrICx against existing related work.

A theory of typed coercions and its applications

In their paper ([24]), the Swamy *et al.* present a framework for coercion generation and type-directed coercion insertion. This framework supports polymorphic coercions, and the construction of coercions using the elements of a *coercion set* as building blocks and allowing component-wise generation of coercions, according to the type structure and transitivity. This coercion set roughly corresponds to TrIC's implicit environment.

Even though there is some overlap between this paper and the work developed in this thesis, there are crucial differences: TrIC is a fully functional language that supports implicit conversions whereas the paper presents framework to support coercions: "A Theory of Typed Coercions and its Applications" does not elaborate on how to construct the converting expressions so that the target expression can then be evaluated. Furthermore, TrIC is based on HM and performs local type inference whereas the aforementioned paper is build upon the simply-typed lambda calculus.

TrIC's conversions are more general than the coercions on the framework, as it too supports transitivity and polymorphism and, contrary to the framework (which only considers base types, functions types and product types), TrIC imposes no restrictions on the structures of types. In addition, neither local scoping, constrained conversions, nor type classes are supported by the framework discussed in the fore mentioned paper.

Cochis

On Calculus of Coherent Implicits (or Cochis, [20]), Schrijvers *et al.* present a type-safe and coherent calculus with support for local scoping, overlapping instances, first-class instances and higher order rules.

Cochis introduces *queries*, which are values to be fetched by type. Calling a function with a query as argument will cause the system to look up a value of the corresponding type in the implicit environment. In Cochis, there is syntax to abstract

over implicit values of some given type and to extend the environment where the implicits are looked up.

Note that, in spite of the fact that both our work on implicit type conversions and Cochis are implicit programming mechanisms, they are completely orthogonal: theirs refers to implicit parameters whereas ours is about implicit conversions.

Dimension Types

Kennedy ([11]) introduces numeric types parameterized on dimensions. These are used to check that expressions that relate multiple numerical values are sensible from a dimensional point of view. As an example, both speed and length may have the same type (say, *Float*) but it would not make sense to sum these values. The system the paper presents allows the programmer to write code that enforces that the dimensions involved are appropriate with respect to one another. The system presented is polymorphic in that functions can be written to work over any dimension.

It would be interesting to consider this idea of dimensions in order to decide if an implicit conversion between some types is appropriate. This would allow enforcing that conversions are only possible between types of the same dimension: for example, types EUR and USD could both have dimension Currency, and, as such, a conversion between them makes sense.

Another avenue worth exploring would be to allow the programmer to write axioms constrained in the dimensions. Instead of writing the axiom "provided there is a conversion from a to b, here is a conversion from list of a to list of b" the programmer could now express the same but with the added constraint that both a and b need to be currencies.

Chapter 9

Conclusion

This thesis proposed extending Haskell with user-defined implicit type conversions, so that the programmer can avoid writting uninteresting code and thus maximizing her efficiency. To showcase this feature we have formalized a minimal language (TrIC) that supports implicit type conversions. Afterwards, the text introduces TrICx, a superset of TrIC with support for type classes, making it closer to Haskell.

In spite of the presence of user-defined implicit type conversions in mainstream programming languages like Scala and C#, TrIC is a trailblazer as is the only language (to the best of our knowledge) to support transitivity in user-defined ITC. Furthermore, TrIC supports parametric polymorphism in the conversions, multiple conditionals (as opposed to Scala's single implicit parameter) and a novel, user-friendly way of locally scoping the conversion axioms. We conjecture TrICx to be complete with respect to the KHC Haskell dialect and transparent to all the programs well-typed without ITC. As such, we are confident (alas, from the definition of conjecture, we can not yet be certain) that this text has met the goals set in Section 4.3. Because we do not expect, in a realistic program, many types to have multiple implicit conversions defined on them, performance should not outweight the benefits of ITC.

The main contributions of this work were the specification and an inference algorithm of the two calculi (TrIC and TrICx) presented (respectively) in Chapters 5 and 7, which includes a graph-based solver for the conversion constraints. In addition to this text, a fully operational extension of the KHC compliant with the specifications of TrICx resulted from this thesis (https://github.com/mesqg/thesis).

Future Work

Future work we believe to be worth investigating includes:

- as discussed in Chapter 4, instead of rejecting programs if there are more than one ways to convert between two ground types, it would be interesting to assign costs to axioms and use the conversion that minimizes the total cost; the total cost could be the sum, over the conversion axioms used to compose the conversion, of a function f that assigns a cost to each conversion axiom.

Some candidate f functions are the function constantly equal to 1 (this would mean choosing the conversion that uses less conversion axioms) or equal to a value assigned by the user when introducing an conversion axiom in the program, enabling the user to penalize some less desirable conversions (for example, conversions with information loss).

- ensuring the axioms with conditions and parametric polymorphism (such as the one in Chapter 6, of type $\forall a, b.j : a \rightsquigarrow b \Rightarrow \text{Wallet } a \rightsquigarrow \text{Wallet } b$) are homomorphic with respect to the composition, *i.e.*, the converting expression $e_{\text{CHF}\rightsquigarrow\text{USD}}$ resulting from applying the axiom with the composed condition CHF $\rightsquigarrow$ USD (EUR$\rightsquigarrow$USD $\circ$ CHF $\rightsquigarrow$ EUR) must be equal to the composition of the converting expressions resulting from applying the axiom twice on the simple conditions: $e_{\text{EUR}\rightsquigarrow\text{USD}} \circ e_{\text{CHF}\rightsquigarrow\text{EUR}}$. With this property, TrIC should be transformed so that the last example presented in Section 6.5 is accepted, since it can no longer introduce any ambiguity;

- introducing the notion of dimension ([11]) and restrict conversions to sensible conversions and/or allow the programer to write axioms that apply only to certain dimensions;

- proving type safety and backwards compatibility for TrIC;

- not requiring type annotations. Within this approach two alternatives stand out: either using complex types (as discussed in Section 4.5) internally but shield the user from them; or enable inference of complex types and further state and prove the existence of a principal solution;

- adding user provided limits on the number of axioms that can be composed into a conversion (for efficiency purposes).

- taking all the constraints that contain a given type variable into account when deciding with what type should substitute that variable, as mentioned in Chapter 6;

- studying optimizations, as for example for the computation of the dominator [12] and for the computation of the conversion paths between two ground types. With TrIC's aversion to multiple converting paths, the search must guarantee to find all the possible conversions. However, a modified version of TrIC with user-provided costs bound to each conversion axiom enables search techniques as the A* [5].

- implementing ITC with the features discussed in Chapter 4 on the GHC.

Appendix A

Poster

KU LEUVEN

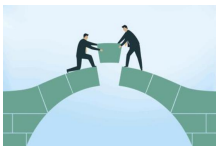
DEPARTMENT OF COMPUTER SCIENCE

Implicit Type Conversions

Francisco Gabriel
francisco.gabriel@student.kuleuven.be
Daily Advisor: Gert-Jan Bottu
Promotor: Prof. Tom Schrijvers

Using Implicit Type Conversions

```
data Euro          = mkEuro Int
data Kwanza        = mkKwanza Int
class Eq a where
  (==) :: a -> a -> Bool
instance Eq Kwanza where
  (==) (mkKwanza val1) (mkKwanza val2) = val1 == val2
implicit Euro ~> Kwanza = \(mkEuro x). mkKwanza (361*x))
(\x . x == (mkKwanza 5)) (mkEuro 4)
```



Scala's (Evil) Spirit

```
implicit def conv(x:Char):Int=3
def weird(x:Char):String=
  x match{
    case 3    => "Int 3"
    case 97   => "Int 97"
    case 'a'  => "Char 'a'"
    case _    => "Other"}
weird('a')
  > "Int 97"
```

Advanced Features

Transitivity (NEW!)

```
implicit Euro    ~> Kwanza = \(mkEuro x). mkKwanza (361*x))
implicit Kwanza ~> Dollar = \(mkKwanza x). mkDollar (0.003*x))
(\x . x == (mkKwanza 5)) (mkEuro 4)
```

Constrained Conversions (NEW!) & Parametric Polymorphism

```
implicit  j: a~>b => [a]~>[b] = map j
```

Local Scoping (NEW!)

```
implicit Euro ~> Kwanza = \(mkEuro x). mkKwanza (361*x))
isRich :: Kwanza -> Bool
isRich (mkKwanza x) = x > 361 000 000
isRich (mkEuro 999 000)
```

```
implicit Euro ~> Kwanza = \(mkEuro x). mkKwanza (370*x))
isRich( mkEuro 999 000)
```

Ambiguity

- How to proceed when multiple conversions are possible?
  - Pick the first found
  - Pick the shortest path
  - ➢ Reject ambiguous code

The Idea

Collect constraints
- Introduce placeholders for conversions
- Store Equality and Conversion Constraints

Solve the constraints
- Yielding the necessary conversions

Substitute placeholders by the converting expressions

Final type and expression!

Encountered Issues

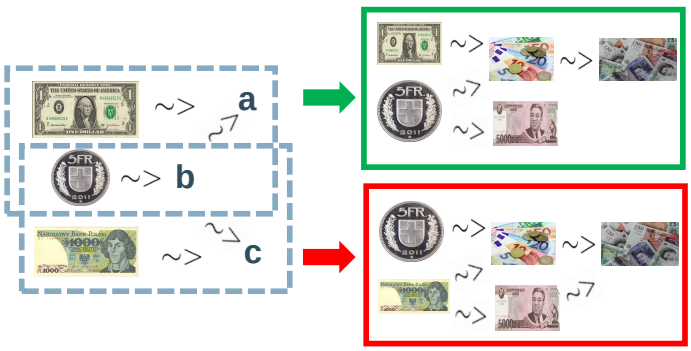
Loops

- When converting A to B
- When satisfying the conditions on implicits

Type Classes

```
instance Eq Euro where …
instance Eq Pound where …
(mkDollar 5)) == (mkCHF 4)
```

Solving the Constraints



Main Results

- Design of a calculus with implicit conversions
- Design of an inference algorithm
- A graph-based constraint solver
- A full implementation of these features

References

- Hindley, J. Roger. The principal type-scheme of an object in combinatory logic. Transactions of the American Mathematical Society, 146:29–60, 1969.
- Pierce, Benjamin C. (2002). Types and programming languages. Cambridge, MA, USA: MIT Press.

KU Leuven
Department of Computer Science
Celestijnenlaan 200A box 2402
3001 Leuven, Belgium

DTAI
KNOWLEDGE REPRESENTATION AND REASONING

Appendix B

Translation to System F

This appendix presents the judgments first introduced in Figure 5.6 and the algorithm to construct a conversion returning a System F expression. Both figures are almost identical to their counterparts presented in the main text but deal with specifics of System F, including type annotations and type applications. The highlighted terms denote System F terms.

$\boxed{I; \Gamma \vdash_{tm} e : \tau; E; Y; e'}$

$$\frac{\begin{array}{c} I; \Gamma, \overline{j_i : \tau_{s_i} \to \tau_{t_i}} \vdash_{tm} e_1 : \forall \overline{a}.\tau; e_1' \\ unify(fv(\tau_{source} \to \tau_{target}); \tau \sim \tau_{source} \to \tau_{target}) = \theta \\ I, e_1' : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_{source} \rightsquigarrow \tau_{target}; \Gamma \vdash_{tm} e_2 : \tau; E; Y; e_2' \end{array}}{I; \Gamma \vdash_{tm} (locimp\ e_1 : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_{source} \rightsquigarrow \tau_{target}\ in\ e_2) : \tau; E; Y; e_2'} \textsc{LocImp}$$

Figure B.1: Modified last rule of Figure 5.6 so that the implicit environment stores System F terms. Other rules remain the same.

$\boxed{\overline{\tau}; I \vDash_{th} \tau_1 \rightsquigarrow \tau_2; \Theta; e}$

$$\frac{[fv(\tau_1) \mapsto \overline{\tau_{1i}}]\tau_1 = [fv(\tau_2) \mapsto \overline{\tau_{2i}}]\tau_2}{\overline{\tau}; I \vDash_{th} \tau_1 \rightsquigarrow \tau_2; [fv(\tau_1) \mapsto \overline{\tau_{1i}}] \circ [fv(\tau_2) \mapsto \overline{\tau_{2i}}]; \lambda x : [fv(\tau_1) \mapsto \overline{\tau_{1i}}]\tau_1.x} \textsc{Unification}$$

$$\frac{\begin{array}{c} \exists^1(e : \forall \overline{a}.\overline{j_i : \tau_{s_i} \rightsquigarrow \tau_{t_i}} \Rightarrow \tau_s \rightsquigarrow \tau_t) \in I \qquad \tau_1 = [\overline{a} \mapsto \overline{\tau}]\tau_s \\ \forall\ i\ :\ (\bullet; I \vDash_{th} \Theta_1 \circ \Theta_{i-1}([\overline{a} \mapsto \overline{\tau}]\tau_{s_i}) \rightsquigarrow \Theta_1 \circ \Theta_{i-1}([\overline{a} \mapsto \overline{\tau}]\tau_{t_i}); \Theta_i; e_i) \\ \tau_3 = \overline{\Theta_i} \circ [\overline{a} \mapsto \overline{\tau}]\tau_t \\ \tau_3 \notin \overline{\tau} \qquad \overline{\tau}, \tau_1; I \vDash_{th} \tau_3 \rightsquigarrow \tau_2; \Theta; e_{rest} \qquad ftyvsOf(\tau_3) = \emptyset \end{array}}{\overline{\tau}; I \vDash_{th} \tau_1 \rightsquigarrow \tau_2; \Theta \circ \overline{\Theta_i} \circ [\overline{a} \mapsto \overline{\tau}]; \lambda x : \tau_1.e_{rest}\ (((\overline{[j_i \mapsto e_i]}e)\ \overline{\tau})\ x)} \textsc{Theorem}$$

Figure B.2: Algorithm to construct the conversion, returning System F expression

Appendix C

TrIC extended Abstract

The following extended abstract (of the work developed in this thesis) was submitted to the International Conference on Functional Programming Student Research Competition (2019) and accepted to appear in the conference.

TrIC: TRansitive Implicit Conversions

FRANCISCO GABRIEL, Katholieke Universiteit Leuven

1 INTRODUCTION

This text presents TrIC (Transitive Implicit Conversion), a calculus with (user-defined) implicit type conversions (ITC). This calculus was developed within the scope of my master thesis. As such, many of its ideas originated from my mentor, Gert-Jan Bottu. The promotor of my thesis, Prof. Tom Schrijvers, has also played an active role in the development of the calculus. Furthermore, George Karachalias contributed to the graph-based constraint solver.

An understanding of the Hindley-Milner type system [1] is sufficient background for this text.

TrIC was designed as a Haskell language feature, so type inference and compatibility with type classes [5] were taken into account. In general, any Haskell program that type-checks without ITC should not be affected in any way. A further goal is that the language is as clear and simple as possible so that the user is able to easily predict its behavior.

To the best of our knowledge, only two mainstream languages already support user-defined implicit type conversions: Scala [3] and C#. TrIC allows transitivity to be used in the implicit type conversions, a feature that is not supported by either of the fore mentioned languages. Other novelties are the local scoping of the conversion axioms and the fact that the user is able to write polymorphic axioms with multiple conditions on the implicit environment.

2 APPROACH

TrIC relies on two basic aspects: a set of user-defined expressions intended to be used to convert terms of type τ_1 to terms of type τ_2; and a way to figure out where, which and how these expressions should be inserted in the source program.

A user-defined *conversion axiom* consists of one such converting expression e, the type $(\tau_s)\ e$ converts from and the type $(\tau_t)\ e$ converts to. Even though τ_s and τ_t could be inferred, the inferred types could be more general then intended by the programmer and thus applicable in undesired circumstances. As such, they should be explicitly written by the user. The set of conversion axioms is referred to as the *implicit environment*. Since text presents locally scoped conversion axioms, the implicit environment can be locally extended.

As for the "where, which and how" to use these converting expressions, the main idea is to perform type inference and, upon encountering an inconsistency, flag it and associate it with a *conversion constraint*, stating that under the current implicit environment I, type τ_1 was inferred where type τ_2 is needed.

For a TrIC source program, type inference and translation to System F [4] occur in parallel, in a two-phased approach, as is common for HM based systems: first the program is partially elaborated and both equality and conversion constraints are generated. Afterwards, we try to entail all the constraints. If that is possible, the program is accepted, a substitution mapping type variables to ground types constructed and applied to it in order to obtain the final type and converting expressions are inserted where needed to obtain the elaborated expression.

3 TRIC

The syntax for terms e in TrIC is the following:

$$e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid K \mid case\ e_1\ of\ \overline{(K_i\ x_{i_j}) \rightarrow e_i} \mid let\ x\ =\ e_1\ in\ e_2 \mid locimp\ e_1 : \sigma_{\rightsquigarrow}\ in\ e_2.$$

Author's address: Francisco Gabriel, francisco.gabriel@student.kuleuven.com, Katholieke Universiteit Leuven.

The novel term is the *locimp*. It consists of two parts: the first, $e_1 : \sigma_{\rightsquigarrow}$, is a conversion axiom where $\sigma_{\rightsquigarrow}$ contains the argument and the result types of the conversion (it also informs about the conditions necessary in order for the axiom to be applicable) and e_1 is the corresponding converting expression; e_2 is any term. The whole *locimp* term behaves as e_2. While typing and elaborating e_2, the implicit environment is extended with the conversion axiom $e_1 : \sigma_{\rightsquigarrow}$.

Suppose there are datatypes EUR, DKK, CHF and USD defined. Each has only one type constructor that takes as an argument a Float. TrIC's syntax enables writing programs as the following:

$$
\begin{aligned}
&\mathit{data}\ \text{Wallet}\ a = K_{\text{Wallet}}\ a\ a \\
&\mathit{locimp}\ \lambda x.K_{\text{USD}}\ 28.0 : \text{DKK} \longrightarrow \text{USD}\ \mathit{in} \\
&\mathit{locimp}\ \lambda x.K_{\text{EUR}}\ 1.41 : \text{USD} \longrightarrow \text{EUR}\ \mathit{in} \\
&\mathit{locimp}\ \lambda x.K_{\text{EUR}}\ 14.1 : \text{CHF} \longrightarrow \text{EUR}\ \mathit{in} \\
&\mathit{locimp}\ \lambda x.\mathit{case}\ x\ \mathit{of}\ K_{\text{Wallet}}\ x_1\ x_2 \longrightarrow K_{\text{Wallet}}(j\ x_1)\ (j\ x_2) \\
&\quad : \forall a, b.j : a \rightsquigarrow b \Rightarrow \text{Wallet}\ a \rightsquigarrow \text{Wallet}\ b\ \mathit{in} \\
&K_{\text{Wallet}}\ (K_{\text{CHF}}\ 5.25)\ (K_{\text{DKK}}\ 1.4142))
\end{aligned}
$$

This program defines a parametric datatype Wallet whose constructor takes two arguments of some type a (suppose they stand for the monetary value in coins and in notes). It also defines four conversion axioms. The first three are "simple" conversions between ground types; the forth showcases polymorphic and constrained user defined conversions. This example program is accepted by TrIC: it is assigned type Wallet EUR.

4 CONSTRAINT GENERATION

The first step in the type inference and translation to System F process is a traversal of the source program in which three things happen: subterms are translated into system F, equality and conversion constraints are generated and place holding variables are introduced, each associated with a conversion constraint in a bijective mapping.

Constraints are generated while typing applications and CASE expressions. Upon encountering an application expression $(e_1\ e_2)$ constraints are generated as follows: if the operator e_1 has type τ_1 and the operand e_2 type τ_2, $e_1\ e_2$ is typed as having type a and an equality constraint specifying that τ_1 is of the form $b \longrightarrow a$, for fresh type variables a and b is stored. A conversion constraint specifying the need to be able to implicitly convert from τ_2 to b under the current implicit environment I is also generated.

The idea for CASE expressions ($\mathit{case}\ e_1\ \mathit{of}\ \overline{(K_i\ \overline{x_{i_j}}) \longrightarrow e_i}$) is that it must be able to implicitly convert from the type of the scrutinee to (an instantiation of) the type specified by the data constructor on the patterns. Concretely, for a scrutinee e_1 with type τ_1, a conversion constraint from τ_1 to a fresh type variable b is generated. Then, for each alternative, b must be equal to the type of the pattern and all the resulting expressions, under the typing environment extended with the respective variables in the pattern, have the same type. Other terms in TrIC merely collect the constraints of their sub-expressions.

In order for a TrIC program to be run, it is first translated to System F. This translation involves using the converting expressions introduced with each conversion axiom to effectively transform an ill-typed program into a System F program.

Note that conversion constraints specify pairs of types such that the first needs to be convertible into the second; these constraints are relations between the types of subterms of applications or subterms of case expressions. In addition, the implicit environment is a set of pairs of an expression and the types it converts from and to.

The idea is to introduce place-holding terms in the program to act as a bridge between the fore mentioned terms whenever a constraint is generated. A place-holder is bound to a conversion

constraint and, after a converting expression e has been computed (during the the constraint entailment process) it will be substituted by e.

In an application, a place holding variable j is put between the operator and the operand. This variable j will later be substituted by a term capable of converting the operand into a term of the type expected by the operator. If the types already match, it will simply be substituted by the identity function.

Similarly, for case expressions $case\ e_1\ of\ \overline{(K_i\ \overline{x_{i_j}}) \rightarrow e_i}$, place-holders will be put to the left of the scrutinee and of the terms returned from the alternatives so that they can later be replaced by the needed conversions.

This approach allows the collection of all the constraints generated by the program before the solving process starts. In addition to the clear separation of the two phases, this approach provides "hooks" (the place-holders) onto which the appropriate converting expressions can be inserted.

5 SOLVING THE CONSTRAINTS

The two types of generated constraints are solved separately: the first step is to solve the equality constraints by performing standard unification [2] on them. Unless this process fails, in which case the program is immediately rejected, it yields the type substitution θ necessary to entail the equality constraints. This type substitution is then applied to conversion constraints.

At this point, the goal is to substitute the remaining type variables in the conversion constraints by ground types. To this end, the constraints with remaining type variables are plotted in a graph. This approach ensures all the appropriate constraints are taken into account when instantiating each type variable and optimizes the constraint solving process. Starting from the sources, we follow the edges of the graph. Upon encountering a type variable a (necessarily as the target type), all the constraints C_i that contain a as the target type are collected. We then proceed to find the dominator: a type τ that is reachable from the source types $\overline{\tau_s}$ of every C_i and such that, for any other type τ' reachable from all $\overline{\tau_s}$, any converting path from any τ_s to τ' can be decomposed into a path from τ_s to τ and another from τ to τ'. Then, a is replaced everywhere by the dominator (a ground type) and we carry on following the edges.

Once all the type variables have been substituted by ground types, we proceed to the entailment of the conversion constraints. An appropriate converting function is generated in response to each conversion constraint. Starting from the source type of the conversion constraint, the system will attempt to apply the conversion axioms present in the implicit environment until the target type is reached. If there are multiple "equally good" ways this can be achieved, the program is rejected. If n axioms are necessary, with converting expressions e_i, the final conversion expression has the form $\lambda x.e_n\ (...(e_1\ x)...)$.

Finally, the place holding variables are substituted by the expression generated during the entailment of their corresponding conversion constraint.

An extension to TrIC with support for type classes, TrICx, has also been studied. TrICx generates three types of constraints: equality and conversion constraints are generated exactly as in TrIC. Class constraints are generated in a standard way and are entailed separately, after the other constraints have been entailed and its resulting type substitutions applied.

6 CONCLUSION

The results of this work are the design of a calculus with implicit type conversions that supports transitive, polymorphic, constrained and locally scoped conversions. Furthermore, an inference algorithm and a graph-based constraint solver have been developed. The (extended) calculus presented in this text (TrICx) has been implemented in the KU Leuven Haskell Compiler.

REFERENCES

[1] Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs.. In *POPL*, Vol. 82. 207–212.
[2] Alberto Martelli and Ugo Montanari. 1982. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 2 (1982), 258–282.
[3] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. The Scala language specification.
[4] John C Reynolds. 1990. Introduction to part II: Polymorphic lambda calculus. *Logical Foundations of Functional Programming, Addison-Wesley* (1990).
[5] Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 60–76.

Bibliography

[1] H. P. Barendregt et al. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[2] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, volume 82, pages 207–212, 1982.

[4] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[6] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[7] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[8] M. P. Jones. A theory of qualified types. In *European symposium on programming*, pages 287–306. Springer, 1992.

[9] M. P. Jones. *Qualified types: theory and practice*, volume 9. Cambridge University Press, 2003.

[10] S. P. Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of functional programming*, 17(1):1–82, 2007.

[11] A. Kennedy. Dimension types. In *European Symposium on Programming*, pages 348–362. Springer, 1994.

[12] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[13] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[14] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[15] T. Mozilla Research. The rust programming language. https://www.rust-lang.org/en-US/.

[16] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The scala language specification, 2004.

[17] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[18] R. T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138. ACM, 1959.

[19] J. C. Reynolds. Introduction to part ii: Polymorphic lambda calculus. *Logical Foundations of Functional Programming, Addison-Wesley*, 1990.

[20] T. SCHRIJVERS, B. C. OLIVEIRA, P. WADLER, and K. MARNTIROSIAN. Cochis: Stable and coherent implicits. *Journal of Functional Programming*, 29:e3, 2019.

[21] M. Sozeau and N. Oury. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2008.

[22] C. Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1-2):11–49, 2000.

[23] M. Sulzmann, M. M. Chakravarty, S. P. Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.

[24] N. Swamy, M. Hicks, and G. M. Bierman. A theory of typed coercions and its applications. In *ACM Sigplan Notices*, volume 44, pages 329–340. ACM, 2009.

[25] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

Master's thesis filing card

Student: Francisco Gabriel

Title: Implicit Type Conversions

UDC: 621.3

Abstract:

This text presents TrIC (Transitive Implicit Conversions), a calculus with (user-defined) implicit type conversions (ITC). TrIC was designed as a Haskell language feature, so type inference and compatibility with type classes [25] were taken into account. In general, any Haskell program that type-checks without ITC should not be affected in any way. A further goal is that the language is as clear and simple as possible so that the user is able to easily predict its behavior. To the best of our knowledge, only two mainstream languages already support user-defined implicit type conversions: Scala [16] and C# [6]. TrIC allows transitivity to be used in the implicit type conversions, a feature that is not supported by either of the fore mentioned languages. Other novelties are the local scoping of the conversion axioms and the fact that the user is able to write polymorphic axioms with multiple conditions on other implicit conversions. In a TrIC program, the programmer is free to introduce conversion axioms (a triple of the type the axiom converts from, the type it converts to and a term capable of performing said conversion) that will, when necessary, be composed and inserted in the source program, in order to translate it into a System F [19] program. This translation occurs simultaneously to type inference, as is common for Hindley-Milner (HM) based systems: the program is traversed and partially elaborated (a process similar to the elaboration of a HM term into System F) and both equality and the novel conversion constraints are generated (place-holding variables are introduced in the program, each associated with a conversion constraint in a bijective mapping). Conversion constraints are generated while typing applications and CASE expressions. They relax HM's equality constraints: it is no longer required that the types present in an application match literally, but rather that it is possible to implicitly convert between the appropriate types, and similarly for CASE expressions. If it is possible to entail all the generated constraints, the program is accepted, a substitution mapping type variables to ground types constructed and applied to it in order to obtain the final type; and converting expressions substitute the place-holders in order to obtain the elaborated expression. Solving the conversion constraints involves a novel graph-based approach. An extension of TrIC with support for type classes (TrICx) is also presented.

Thesis submitted for the degree of Master of Science in Engineering: Computer Science

Thesis supervisor: Prof. dr. ir. Tom Schrijvers

Assessors: Dr. Clement Gautrais
Dr. Amin Timany

Mentor: Ir. Gert-Jan Bottu