

Complete and Decidable Type Inference for GADTs

Tom Schrijvers*

Katholieke Universiteit Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Simon Peyton Jones

Microsoft Research Cambridge, UK
simonpj@microsoft.com

Martin Sulzmann

Intaris Software GmbH, Germany
martin.sulzmann@gmail.com

Dimitrios Vytiniotis

Microsoft Research Cambridge, UK
dimitris@microsoft.com

Abstract

GADTs have proven to be an invaluable language extension, a.o. for ensuring data invariants and program correctness. Unfortunately, they pose a tough problem for type inference: we lose the principal-type property, which is necessary for modular type inference.

We present a novel and simplified type inference approach for local type assumptions from GADT pattern matches. Our approach is complete and decidable, while more liberal than previous such approaches.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Functional Languages; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type Structure

General Terms Algorithms, Languages

Keywords Haskell, type inference, GADTs

1. Introduction

Generalized Algebraic Data Types (GADTs) pose a particularly tough problem for type inference: we lose the principal-type property, which is necessary for modular type inference (CH03), and it is even undecidable whether or not a term *has* a principal type (Section 4.3).

A variety of papers have tackled this problem, by a combination of user-supplied type annotations and/or constraint-based inference (PVWW06; PRG06; SP07; SSS08). Unfortunately, none of these approaches is satisfying, even to their originators, for a variety of reasons (Section 9). Simonet and Pottier give an excellent executive summary in the closing sentences of their recent paper (SP07):

We believe that one should, instead, strive to produce simpler constraints, whose satisfiability can be efficiently determined by a (correct and complete) solver. Inspired by Peyton Jones et al.'s wobbly types (PVWW06), recent work by

* Post-doctoral researcher of the Fund for Scientific Research - Flanders.

Pottier and Régis-Gianas (PRG06) proposes one way of doing so, by relying on explicit, user-provided type annotations and on an ad hoc local shape inference phase. It would be interesting to know whether it is possible to do better, that is, not to rely on an ad hoc preprocessing phase.

This is the challenge we meet in this paper. In particular, our contributions are:

- We present **OutsideIn**, a new inference algorithm for GADT programs (Sections 4 and 5). It deals with the tricky problem of solving so-called implication constraints (Section 4.2) by allowing information to propagate from outside a GADT pattern match to the inside, but not vice versa.
- The declarative specification of the type system is given in Section 6. While still not entirely satisfactory, is arguably significantly simpler than earlier attempts. It is also rather expressive: it types strictly more programs than GHC's existing algorithm (PVWW06), and very nearly all those typed by (PRG06) — Section 9 elaborates.
- Our type system has the crucial principal-types property; and any program accepted by our type system is also typed by the simpler “natural” type system for GADTs, which lacks principal types. Furthermore, all programs typeable in our approach *do have principal types* in the natural type system for GADTs and hence our approach is not “ad hoc” (Section 7.1). That is not the case for either (PVWW06) or (PRG06).
- The type inference algorithm is sound and complete with respect to the specification; and it is decidable (Section 7.2).
- The type inference algorithm is easy to implement (Section 8). The inference engine gathers and solves constraints, but that is something Haskell compilers already do for type classes; all we do here is add some new forms of constraints. Nevertheless inference remains efficient, because almost all equalities can be solved with on-the-fly unification in the conventional Hindley-Milner way, with constraints gathered only when necessary. Our prototype implementation is available for download. *Note: we have not yet implemented it in GHC, but expect to have done so before ICFP.*

Our approach builds directly on the work of others. We urge the reader to consult Section 9 for a summary of these foundations.

2. The challenge we address

Generalized Algebraic Data Types (GADTs) have proved extremely popular with programmers, but they present the type inference engine with tricky choices. Consider the following GADT program:

```
data T :: *->* where
  T1 :: Int -> T Bool
  T2 :: [a] -> T a

f1 (T1 n) = n>0
```

What type should be inferred for function `f1`? Alas there are two possible most-general types, neither of which is an instance of the other:

```
f1 :: ∀a. T a → Bool
f1 :: ∀a. T a → a
```

The loss of principal types is both well-known and unavoidable (CH03). Since `f1` has no principal type (one that is more general than all others) the right thing must be to reject the program, and ask the programmer to say which type is required by means of an explicit type signature, like this, for example:

```
f1 :: T a -> a
f1 (T1 n) = n
```

But exactly *which* programs should be rejected in this way? For example, consider `f2`:

```
f2 (T1 n) = n
f2 (T2 xs) = null xs
```

Since `null :: [a] -> Bool` returns a `Bool`, and `T2` is an ordinary (non-GADT) data constructor, the *only* way to type `f2` is with result `Bool`, so the programmer might be confused at being required to say so. After all, there is only one solution: why can't the compiler find it?

An exactly similar issue arises in relation to variables in the environment. Consider

```
h1 x (T1 n) = x && n>0
h1 x (T2 xs) = null xs
```

Which of these two incomparable types should we infer?

```
h1 :: ∀a. a → T a → Bool
h1 :: ∀a. Bool → T a → Bool
```

Again, since neither is more general than the other, we should reject the program. But if we somehow know from elsewhere that `x` is a `Bool`, then there is no ambiguity, and we might prefer to accept the definition. Here is an example

```
h2 x (T1 n) = x && n>0
h2 x (T2 xs) = not x
```

The key difficulty is that *a GADT pattern match brings local type constraints into scope*. For example in the `T1` branch of the definition of `f1`, we know that the constraint $a \sim \text{Bool}$ holds, where the second argument of `f1` has type `T a`.¹ Indeed, while the declaration for the GADT `T` above is very convenient for the programmer, it is quite helpful to re-express it with an explicit equality constraint, like this²:

```
data T :: *->* where
  T1 :: (a ~ Bool) => Int -> T a
  -- Was: T1 :: Int -> T Bool
  T2 :: [a] -> T a
```

¹ We consistently use “ \sim ” to denote type equalities, because “ $=$ ” is used for too many other things.

² GHC allows both forms, and treats them as equivalent.

Term variables	x, y, z
Type variables	a, b, c
Unification variables	α, β, γ
Type constructors	S, T
Data constructors	K
Programs and data type declarations	
$prog$	$::= dd_1 \dots dd_n; e$
dd	$::= \text{data } T \ a_1 \dots a_m \ \text{where}$
	$\frac{K :: \forall a_1 \dots a_m, b_1 \dots b_n. C \Rightarrow \tau_1 \rightarrow \dots \rightarrow \tau_p \rightarrow T \ a_1 \dots a_m}{}$
Terms	$e ::= K \mid x \mid \lambda x. e \mid e \ e$ $\quad \mid \text{let } \{g = e\} \text{ in } e$ $\quad \mid \text{let } \{g :: \sigma = e_1\} \text{ in } e_2$ $\quad \mid \text{case } e \text{ of } [p_i \rightarrow e_i]_{i \in I}$
Patterns	$p ::= K \ x_1 \dots x_n$
Type envt	$\Gamma ::= \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$
Type variables	$\nu ::= \alpha \mid a$
Monotypes	$\tau, v ::= \nu \mid \tau \rightarrow v \mid T \ \bar{\tau}$
Type Schemes	$\sigma ::= \tau \mid \forall \bar{a}. C \Rightarrow \tau$
Constraints	$C, D ::= \tau \sim \tau \mid C \wedge C \mid \epsilon$
Impl. Constraints	$F ::= C \mid [\bar{\alpha}] (\forall \bar{b}. C \supset F) \mid F \wedge F$
Unifiers	$\theta ::= \emptyset \mid \theta, \{\alpha := \tau\}$
Substitutions	$\phi ::= \emptyset \mid \phi, \{\nu := \tau\}$
	$fiv(\tau) = \text{The free unification variables of } \tau$ (and similarly $fiv(\Gamma)$)
Substitution	$\phi(F_1 \wedge F_2) = \phi(F_1) \wedge \phi(F_2)$ $\phi([\bar{\alpha}] \forall \bar{b}. C \supset F) = [fiv(\phi(\bar{\alpha}))] \forall \bar{b}. \phi(C) \supset \phi(F)$ Substitution on C and τ is conventional
Abbreviations	$\forall \bar{a}. \tau \triangleq \forall \bar{a}. \epsilon \Rightarrow \tau$ $[\bar{\alpha}] (\forall \bar{b}. F) \triangleq [\bar{\alpha}] (\forall \bar{b}. \epsilon \supset F)$

Figure 1. Syntax of Programs

You may imagine a value of type `T τ`, built with `T1`, as a heap-allocated object with two fields: a value of type `Int`, and some *evidence* that $\tau \sim \text{Bool}$. When the value is *constructed* the evidence must be supplied; when the value is *de-constructed* (i.e. matched in a pattern) the evidence becomes available locally. While in many systems, including GHC, this “evidence” has no run-time existence, the vocabulary can still be helpful and GHC does use explicit evidence-passing in its intermediate language (SCPD07).

3. Formal setup

Before we can present our approach, we briefly introduce our language, and the general form of its type system.

3.1 Syntax

Figure 1 gives the syntax of terms and types, which should look familiar to Haskell programmers. A program consists of a set of data type declarations together with a term e . Terms consist of the lambda calculus, together with `let` bindings (perhaps with a user-supplied type signature), and simple `case` expressions to perform pattern matching. A data type declaration introduces a type

$C, \Gamma \vdash e : \tau$	$C, \Gamma \vdash_p p \rightarrow e : \tau \rightarrow v$
$\text{(VAR)} \frac{(x : \forall \bar{a}. v) \in \Gamma \quad \phi = \{\bar{a} := \bar{\tau}\}}{C, \Gamma \vdash x : \phi(v)} \quad \text{(CON)} \frac{K :: \forall \bar{a}. D \Rightarrow v}{C, \Gamma \vdash K : \phi(v)}$	$\text{(EQ)} \frac{C, \Gamma \vdash e : \tau_1 \quad C \models \tau_1 \sim \tau_2}{C, \Gamma \vdash e : \tau_2}$
$\text{(ABS)} \frac{C, \Gamma \cup \{x : \tau_1\} \vdash e : \tau_2}{C, \Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}$	$\text{(APP)} \frac{C, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad C, \Gamma \vdash e_2 : \tau_1}{C, \Gamma \vdash e_1 e_2 : \tau_2}$
$\text{(LET)} \frac{C, \Gamma \cup \{g : \tau_1\} \vdash e_1 : \tau_1 \quad \bar{a} = f\bar{v}(\tau_1) - f\bar{v}(C, \Gamma) \quad C, \Gamma \cup \{g : \forall \bar{a}. \tau_1\} \vdash e_2 : \tau_2}{C, \Gamma \vdash \text{let } \{g = e_1\} \text{ in } e_2 : \tau_2}$	$\text{(LETA)} \frac{C, \Gamma \cup \{g : \forall \bar{a}. \tau_1\} \vdash e_1 : \tau_1 \quad C, \Gamma \cup \{g : \forall \bar{a}. \tau_1\} \vdash e_2 : \tau_2}{C, \Gamma \vdash \text{let } \{g :: \forall \bar{a}. \tau_1 = e_1\} \text{ in } e_2 : \tau_2}$
$\text{(CASE)} \frac{C, \Gamma \vdash e : \tau_1 \quad C, \Gamma \vdash_p p_i \rightarrow e_i : \tau_1 \rightarrow \tau_2 \quad \text{for } i \in I}{C, \Gamma \vdash \text{case } e \text{ of } [p_i \rightarrow e_i]_{i \in I} : \tau_2}$	$\text{(PAT)} \frac{K :: \forall \bar{a}, \bar{b}. D \Rightarrow v_1 \rightarrow \dots \rightarrow v_p \rightarrow T \bar{a} \quad f\bar{v}(C, \Gamma, \bar{\tau}, \tau_r) \cap \bar{b} = \emptyset \quad \phi = \{\bar{a} := \bar{\tau}\} \quad \text{consistent}(C \wedge \phi(D))}{C \wedge \phi(D), \Gamma \cup \phi\{x_1 : v_1, \dots, x_p : v_p\} \vdash e : \tau_r}$
	$C, \Gamma \vdash_p K x_1 \dots x_p \rightarrow e : T \bar{\tau} \rightarrow \tau_r$

Figure 2. Simple but over-permissive typing rules

$\text{(TRUE)} \frac{}{C \models \epsilon}$	$\text{(REFL)} \frac{}{C \models \tau \sim \tau}$	$\text{(SYM)} \frac{C \models \tau \sim v}{C \models v \sim \tau}$
$\text{(TRANS)} \frac{C \models \tau_1 \sim \tau_2 \quad C \models \tau_2 \sim \tau_3}{C \models \tau_1 \sim \tau_3}$		
$\text{(GIVEN)} \frac{}{C_1 \wedge C_2 \models C_2}$	$\text{(CONJ)} \frac{C \models F_1 \quad C \models F_2}{C \models F_1 \wedge F_2}$	
$\text{(STRUCT)} \frac{C \models \tau_i \sim v_i}{C \models T \bar{\tau}_i \sim T \bar{v}_i}$	$\text{(TCON)} \frac{C \models T \bar{\tau}_i \sim T \bar{v}_i}{C \models \tau_i \sim v_i}$	
$\text{(IMPL)} \frac{C \wedge C_1 \models F \quad \bar{b} \cap f\bar{v}(C) = \emptyset}{C \models [\bar{a}] (\forall \bar{b}. C_1 \supset F)}$		

Figure 3. Equality theory

constructor T and one or more data constructors K_i , each of which is given a type signature. As described in Section 2, in the case of GADTs the data constructor’s type contains a set of constraints C , that are brought into scope when K is used in a pattern match, and required when K is used as a constructor.

The syntax of types, and of constraints, is also given in Figure 1. Note that unification variables α denote unknown types and only appear during type inference, never in the resulting typings. To avoid clutter we use only equality constraints $\tau_1 \sim \tau_2$ in our formalism, although in GHC there are several other sorts of constraint, including implicit parameters and type classes. We treat conjunction (\wedge) as a commutative and associative operator as is conventional. Implication constraints F will be introduced in Section 4.4.

3.2 Type system

The declarative specification of a type system usually takes the form of a typing judgement

$$\Gamma \vdash e : \tau$$

with the meaning “in type environment Γ the term e has type τ ”. In a system with GADTs, however, a pattern match may bring into scope some *local* equality constraints. The standard way to express this is with the judgement

$$C, \Gamma \vdash e : \tau$$

meaning “in a context where constraints C are in scope, and type environment Γ the term e has type τ ”. For example, here is a valid judgement:

$$(a \sim \text{Bool}), \{x : a, \text{not} : \text{Bool} \rightarrow \text{Bool}\} \vdash \text{not } x : \text{Bool}$$

The judgement only holds because of the availability of the local equality $a \sim \text{Bool}$.

The type system of Figure 2 takes exactly this form. For example, rule (CON) instantiates the type scheme of a data constructor in the usual way, except that it has the additional premise

$$C \models \phi(D)$$

This requires that the “wanted” constraints $\phi(D)$ must be deducible from the “given” constraints C . To be concrete we give the (routine) definition of \models in Figure 3. Compare rule (CON) to rule (VAR), where the type scheme does not mention constraints.

Rule (EQ) allows us to use the available constraints C to adjust the result type τ_1 to any equal type τ_2 . Finally, a `case` expression uses an auxiliary judgement \vdash_p to typecheck the case alternatives. Notice the way that the local constraints C are extended when going inside a pattern match (in rule (PAT)), just as the type environment is augmented when going inside a lambda-term (in rule (ABS)).

Whenever we go inside a pattern match, we require the given constraint $C \wedge \phi(D)$ to be *consistent*. Consistency is defined by the rule:

$$\text{(CONSISTENT)} \frac{\exists \phi. \epsilon \models \phi(C)}{\text{consistent}(C)}$$

i.e. C is consistent if it has a unifier. Consistency implies that we will reject programs with inaccessible case branches.

A second point to notice about rule (PAT) is that a data constructor may have existential type variables \bar{b} as well as universal type

variables \bar{a} ³. Rule (PAT) must check that the existential variables are not mentioned in the environment C, Γ , or the scrutinee type $T \bar{\tau}$, or the result type τ_r . In the following example, `fx1` is well-typed, but `fx2` is not because the existential variable `b` escapes:

```
data X where
  X1 :: forall b. b -> (b->Int) -> X

fx1 (X1 x f) = f x
fx2 (X1 x f) = x
```

3.3 Properties

The type checking problem for GADTs is decidable (CH03; SP07). However, type inference turns out to be extremely difficult. The example from Section 2 shows that GADTs lack principal types. In Appendix B, we show that there are programs which can be given an infinite number of most-general types. The difficulty is that the type system can type *too many terms*. Hence, our goal is to restrict the type system to reject just enough programs to obtain a tractable type inference system which enjoys principal types. Nevertheless, we regard Figure 2 as the “natural” type system for GADTs, against which any such restricted system should be compared.

4. A new approach

In this section we describe our new approach to type inference for GADTs. Type system designers often develop a type inference algorithm hand-in-hand with the specification of the type system: there is no point in a specification that we cannot implement, or an implementation whose specification is incomprehensible. We begin with the inference algorithm.

4.1 Type inference by constraint solving

It is well known that type inference can be carried out in two stages: first *generate constraints* from the program text, and then *solve the constraints* ignoring the program text (PR05). The generated constraints involve *unification variables*, which stand for as-yet-unknown types, and solving the constraints produces a *substitution* that assigns a type to each unification variable. The most basic form of constraint is a *type equality constraint* of form $\tau_1 \sim \tau_2$, where τ_1 and τ_2 are types.

For example, consider the definition

```
data Pair :: *->*->* where
  MkP :: a -> b -> Pair a b

f = \x. MkP x True
```

The data type declaration specifies type of the constructor `MkP`, thus:

$$\text{MkP} : \forall a b. a \rightarrow b \rightarrow \text{Pair } a \ b$$

Now consider the right-hand-side of `f`. The constraint generator makes up unification variables as follows:

α type of the entire right-hand side
 β_x type of `x`
 γ_1, γ_2 instantiate `a, b` respectively, when instantiating the call of `MkP`

From the text we can generate the following equalities:

$\beta_x \sim \gamma_1$ First argument of `MkP`
 $\text{Bool} \sim \gamma_2$ Second argument of `MkP`
 $\alpha \sim P \ \gamma_1 \ \gamma_2$ Result of `MkP`

³The former are called existential, despite their apparent quantification with \forall , because the constructor’s type is isomorphic to $K : \forall \bar{a}. (\exists \bar{b}. D \times v_1 \times \dots \times v_p) \rightarrow T \bar{a}$.

These constraints can be solved by unification, yielding the substitution $\{\alpha := P \ \beta_x \ \text{Bool}, \gamma_2 := \text{Bool}, \gamma_1 := \beta_x\}$. This substitution constitutes a “solution”, because under that substitution the constraints are all of form $\tau \sim \tau$. Not only that, but the unification algorithm finds the *most general* substitution that solves the constraints. Temporarily leaving aside the question of generalization that’s all there is to type inference for ML.

4.2 Constraint solving with GADTs

What happens when GADTs enter the picture? Consider our standard example term:

$$\backslash x. \text{ case } x \text{ of } \{ T1 \ n \rightarrow n > 0 \}$$

recalling the type of `T1`:

$$T1 : \forall a. (\text{Bool} \sim a) \Rightarrow \text{Int} \rightarrow T \ a$$

Again we make up fresh unification variables for any unknown types:

α type of the entire right-hand side
 β_x type of `x`

Matching `x` against a constructor from type `T` imposes the constraint $\beta_x \sim T \ \gamma$, for some new unification variable γ . From the term `n > 0` we get the constraint $\alpha \sim \text{Bool}$, but that arises inside the branch of a `case` that brings into scope the constraint $\gamma \sim \text{Bool}$. We combine these two into a new sort of constraint, called an *implication constraint*:

$$\gamma \sim \text{Bool} \supset \alpha \sim \text{Bool}$$

Now our difficulty becomes clear: there is no most-general unifier for implication constraints. The substitutions

$$\{\alpha := \text{Bool}\} \quad \text{and} \quad \{\alpha := \gamma\}$$

are both solutions, but neither is more general than the other.

On the other hand, sometimes there obviously *is* a unique solution. Consider `f2` from Section 2:

$$\backslash x. \text{ case } x \text{ of } \{ T1 \ n \rightarrow n > 0; T2 \ xs \rightarrow \text{null } xs \}$$

From the two alternatives of the `case` we get two constraints, respectively:

$$\gamma \sim \text{Bool} \supset \alpha \sim \text{Bool} \quad \text{and} \quad \alpha \sim \text{Bool}$$

Since the second constraint can be solved only by $\{\alpha := \text{Bool}\}$, there is a unique most-general unifier to this system of constraints.

4.3 GADT type inference is undecidable

Multiple pattern clauses give rise to a conjunction of implication constraints

$$(C_1 \supset C'_1) \wedge \dots \wedge (C_n \supset C'_n)$$

The task of GADT type inference is to find a substitution θ such that each $\theta(C'_i)$ follows from $\theta(C_i)$. This problem is identical to the simultaneous rigid E-unification problem which is known to be undecidable (DV95). Hence, we can immediately conclude that GADT type inference is undecidable. To restore decidability and most general solutions, we consider a restricted implication solver algorithm.

4.4 The OutsideIn solving algorithm

Our idea is a simple one: *we must refrain from unifying a global unification variable under a local equality constraint*. By “global” we mean “free in the type environment⁴”, and we must record that information in the implication constraint itself, thus

$$[\alpha] (\gamma \sim \text{Bool} \supset \alpha \sim \text{Bool})$$

⁴We must treat the result type as part of the type environment.

because α is free in the type environment. When solving this constraint we must refrain from unifying $\{\alpha := \text{Bool}\}$; hence, the constraint by itself is insoluble. It can be solved only if there is some *other* constraint that binds α .

The syntax of implication constraints F is given in Figure 1. An implication constraint an ordinary constraint C , or a conjunction of implications, or has the form $[\bar{\alpha}]\bar{v}\bar{b}.C \supset F$. We call the set of unification variables $\bar{\alpha}$ the *untouchables* of the constraint, and the set of type variables \bar{b} the *skolems* of the constraint. Applying a substitution to an implication constraint requires a moment's thought, because an untouchable might be mapped to a type by the substitution, so we must take the free unification variables of the result; see Figure 1. We often omit the untouchables, skolems, or C when they are empty.

More precisely, to solve a set of implication constraints F , proceed as follows:

1. Split F into $F_g \wedge F_s$, where all the constraints in F_g are proper implications, and F_s are all simple. An implication is *simple* if it does not involve any local equalities, and *proper* otherwise:

$$\begin{aligned} F_g &::= F_g \wedge F_g \mid [\bar{\alpha}]\bar{v}\bar{b}.C \supset F & C \neq \epsilon \\ F_s &::= F_s \wedge F_s \mid C \mid [\bar{\alpha}]\bar{v}\bar{b}.\epsilon \supset F_s \end{aligned}$$

2. Solve the simple constraints F_s by ordinary unification, yielding a substitution θ .
3. Now apply θ to F_g , and solve each implication in $\theta(F_g)$.

In the last step, how do we solve a proper implication $[\bar{\alpha}]\bar{v}\bar{b}.C \Rightarrow F$? Simply find ϕ , the most general unifier of C , and solve $\phi(F)$, under the restriction that the solution must not bind $\bar{\alpha}$.

This algorithm is conservative: if it finds a unifier, that solution will be most general, but the converse is not true. For example, the algorithm fails to solve the constraint

$$[\alpha] (\gamma \sim \text{Bool} \supset \alpha \sim \text{Int})$$

but the constraint actually has a unique solution, namely $\{\alpha := \text{Int}\}$.

5. The OutsideIn approach in detail

It is time to nail down the details. Our approach relies on constraint generation and constraint solving. We specify top-level constraint generation with $\Gamma \vdash_W e : \tau, F$ to be read as: in the environment Γ , we may infer type τ for the expression e and generate constraint F . Solving a constraint F to produce a substitution θ is specified with $\vdash_s F : \theta$. The top-level inference algorithm is then given by the judgement \vdash_{inf} in Figure 4.⁵

We start by discussing constraint generation (Section 5.1) and constraint solving (Section 5.2). Subsequently we present the high-level declarative type system (Section 6).

5.1 Generating implication constraints

The constraint generation algorithm is given in Figure 4 with the judgement

$$\Gamma \vdash_W e : \tau, F$$

In this judgement, thought of as an algorithm, Γ and e are inputs, while τ and F are outputs.

Rules (VAR), (CON), (ABS), and (APP) are straightforward. Rule (PAT) generates an implication constraint, as described informally

⁵We start with an initially-empty environment, informally relying on a fixed, implicit global environment to specify the types of each data constructor.

$$\begin{array}{c} \boxed{\Gamma_{\text{inf}} e : \tau} \\ \text{(INFER)} \frac{\vdash_W e : \tau, F \quad \vdash_s F : \theta}{\vdash_{\text{inf}} e : \theta(\tau)} \\ \boxed{\Gamma \vdash_W e : \tau, F} \\ \text{(VAR)} \frac{(x : \forall \bar{a}.\tau) \in \Gamma \quad \bar{\alpha} \text{ fresh} \quad \phi = \{\bar{a} := \bar{\alpha}\}}{\Gamma \vdash_W x : \phi(\tau), \epsilon} \\ \text{(CON)} \frac{K :: \forall \bar{a}.C \Rightarrow \tau \quad \bar{\alpha} \text{ fresh} \quad \phi = \{\bar{a} := \bar{\alpha}\}}{\Gamma \vdash_W K : \phi(\tau), \phi(C)} \\ \text{(APP)} \frac{\Gamma \vdash_W e_1 : \tau_1, F_1 \quad \Gamma \vdash_W e_2 : \tau_2, F_2 \quad \alpha \text{ fresh} \quad F = F_1 \wedge F_2 \wedge (\tau_1 \sim \tau_2 \rightarrow \alpha)}{\Gamma \vdash_W e_1 e_2 : \alpha, F} \\ \text{(ABS)} \frac{\alpha \text{ fresh} \quad \Gamma \cup \{x : \alpha\} \vdash_W e : \tau, F}{\Gamma \vdash_W \lambda x.e : \alpha \rightarrow \tau, F} \\ \text{(LETA)} \frac{\Gamma \cup \{g : \forall \bar{a}.\tau\} \vdash_W e_1 : \tau', F_1 \quad \Gamma \cup \{g : \forall \bar{a}.\tau\} \vdash_W e_2 : v, F_2 \quad F = F_2 \wedge [\text{fuw}(\Gamma)](\forall \bar{a}.F_1 \wedge \tau \sim \tau')}{\Gamma \vdash_W \text{let } \{g :: \forall \bar{a}.\tau = e_1\} \text{ in } e_2 : v, F} \\ \text{(LET)} \frac{\alpha \text{ fresh} \quad \Gamma \cup \{g : \alpha\} \vdash_W e_1 : \tau, F_1 \quad F'_1 = F_1 \wedge \alpha \sim \tau \quad F_s = \text{simple}(F'_1) \quad \vdash_s F_s : \phi_s \quad \bar{\beta} = \text{fuw}(\phi_s(\tau)) - \text{fuw}(\phi_s(\Gamma)) \quad \bar{b} \text{ fresh} \quad \theta_k = \{\bar{\beta} := \bar{b}\}}{\Gamma \cup \{g : \forall \bar{b}.\theta_k(\phi_s(\tau))\} \vdash_W e_2 : v, F_2 \quad F = F_2 \wedge [\text{fuw}(\Gamma)]\forall \bar{b}.\theta_k(F'_1)} \\ \Gamma \vdash_W \text{let } \{g = e_1\} \text{ in } e_2 : v, F \\ \text{(CASE)} \frac{\Gamma \vdash_W e : \tau_e, F_e \quad \Gamma \vdash_P p_i \rightarrow e_i : \tau_i \rightarrow v_i, F_i \quad \text{for } i \in I \quad \alpha \text{ fresh} \quad F = F_e \wedge \bigwedge_{i \in I} (F_i \wedge \tau_e \sim \tau_i \wedge \alpha \sim v_i)}{\Gamma \vdash_W \text{case } e \text{ of } [p_i \rightarrow e_i]_{i \in I} : \alpha, F} \\ \boxed{\Gamma \vdash_P p \rightarrow e : \tau \rightarrow v, F} \\ \text{(PAT)} \frac{K :: \forall \bar{a}.\bar{b}.D \Rightarrow \tau_1 \rightarrow \dots \rightarrow \tau_p \rightarrow T \bar{a} \quad \bar{b} \notin \text{ftv}(\Gamma, \tau_e) \quad \bar{\alpha} \text{ fresh} \quad \phi = \{\bar{a} := \bar{\alpha}\}}{\Gamma \cup \phi\{x_1 : \tau_1, \dots, x_p : \tau_p\} \vdash_W e : \tau_e, F_e \quad F = [\bar{\alpha} \cup \text{fuw}(\Gamma, \tau_e)](\forall \bar{b}.\phi(D) \supset F_e)} \\ \Gamma \vdash_P K x_1 \dots x_p \rightarrow e : T \bar{\alpha} \rightarrow \tau_e, F \end{array}$$

Figure 4. Translation to Constraints

in Section 4.2, while (CASE) simply combines the constraints from the alternatives.

Rule (LETA) generates implication constraints for an *annotated* let-binding. Pay attention to two details: (a) the inferred type τ' must equate to the declared type τ , and (b) the universally quantified variables \bar{a} must not escape their scope. The first is captured in an

additional equality constraint $\tau \sim \tau'$, and the latter in the degenerate implication constraint.

Rule (LET) for *unannotated* let-expressions is much trickier. First, it derives the constraint F_1 for the bound expression e_1 . The conventional thing to do at this point is to create fresh type variables \bar{b} for the variables $\bar{\beta}$ that are not free in the environment with a substitution $\theta_k = \{\bar{\beta} := \bar{b}\}$, and abstract over the constraints, inferring the following type for g (SP07):

$$g : \forall \bar{b}. \theta_k(F_1 \Rightarrow \tau)$$

This is correct, but by postponing all solving until the second phase we get unexpectedly complicated types for simple definitions. For example, from the definition

```
g = \x.x && True
```

we would infer the type

$$g :: \forall b. b \sim \text{Bool} \Rightarrow b \rightarrow \text{Bool}$$

when the programmer would expect the equivalent but simpler type $\text{Bool} \rightarrow \text{Bool}$. Furthermore, this approach obviously requires that types can take the form $F \Rightarrow \tau$ — including the possibility that F is itself an implication! It all works fine (see (SP07) for example), but it makes the types significantly more complicated and, in an evidence-passing internal language such as that used by GHC, creates much larger elaborated terms.

Instead, we interleave constraint *generation* and constraint *solving* in rule (LET), thus⁶:

- Generate constraints for e_1 under the assumption that $g : \alpha$ (we allow recursion in let).
- Add the constraint $\alpha \sim \tau$ to tie the recursive knot in the usual way, forming F'_1 .
- We cannot, at this stage, guarantee to solve *all* the constraints in F'_1 , because the latter might include implications that can only be solved in the presence of information from elsewhere. So we extract from F'_1 the “simple” constraints, F_s :

$$\begin{aligned} \text{simple}(C) &= C \\ \text{simple}(F_1 \wedge F_2) &= \text{simple}(F_1) \wedge \text{simple}(F_2) \\ \text{simple}([\bar{\alpha}](\forall \bar{b}. F)) &= [\bar{\alpha}]\forall \bar{b}. \text{simple}(F) \\ \text{simple}([\bar{\alpha}](\forall \bar{b}. C \supset F)) &= \epsilon \quad C \neq \epsilon \end{aligned}$$

- Solve F_s appealing to our solver $\vdash F_s : \phi_s$. Notice that this ϕ_s may bind skolems, a point that we will return to. Moreover, notice that if unification fails for a simple constraint (such as $\text{Bool} \sim \text{Int}$) then the program is definitely untypeable.
- Apply the solving substitution ϕ_s to τ and Γ , and compute the set of variables $\bar{\beta}$ over which to quantify in the usual way.
- Skolemise the variables we can quantify over, using a substitution θ_k .
- Typecheck the body of the let, with a suitable type for g .
- Lastly we figure out the constraint F to return. It include F_2 of course, and F'_1 suitably wrapped in a \forall to account for the skolemized variables just as in (LETA).

There are two tricky points in this process.

- First, notice that the substitution returned by solving F_s is a ϕ -substitution and not merely a θ -substitution, and hence can bind skolem variables. This is perhaps unintuitive—after all in ordinary Hindley-Milner we would require that the substitution

⁶This interleaving is not so unusual: every Haskell compiler does the same for type-class constraints.

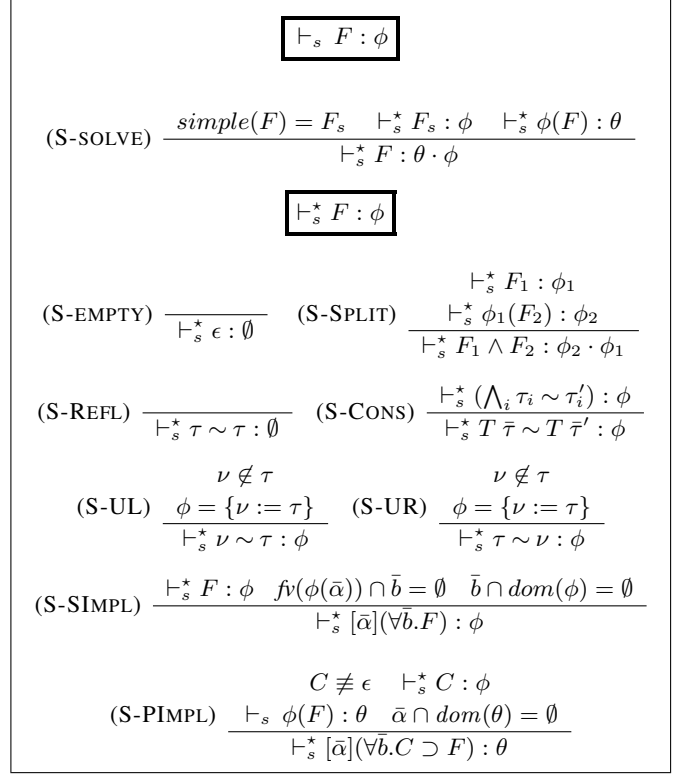


Figure 5. Solver algorithm

binds only unification variables. Nevertheless, in the presence of given equations this is not enough. Consider:

```
data T where
  MkT :: forall a b. (a ~ b) => a -> b -> T
```

```
foo = case e of
  MkT y z -> let h = [y,z] in ()
```

Constraint generation for the inner let definition produces the constraint $a \sim b$, where a and b are the existential variables introduced by the pattern match. But we must not fail at this point and hence the solver of the F_s constraint must be prepared to encounter equalities between skolem variables.

- Second, notice that we do *not* apply ϕ_s to F'_1 . Why not? Because ϕ_s may bind variables free in Γ , and we must not lose that information. But the very same information is present in the original F'_1 , so if we return F'_1 unchanged (apart from applying the skolemizing substitution θ_k , then the rest of the derivation will be able to “see” it too.

Finally, notice that there is quite a bit of “junk” in F'_1 . Consider the definition of g given earlier in this subsection. We will get

$$\tau = \beta \rightarrow \text{Bool}, F'_1 = \beta \sim \text{Bool}, \theta_s = \{\beta := \text{Bool}\}$$

Now β plays no part in the rest of the program, but still lurks in F'_1 . Because of our freshness assumptions, however, it does no harm either.

5.2 The OutsideIn Implication Solver

Figure 5 presents the rules of our implication solver. The solver judgement is of the form $\vdash_s F : \phi$. This judgement should be thought of as taking F as input and producing a ϕ , such that $\models \phi(F)$ according to the equational theory of Figure 3. The

judgement appeals to $\text{simple}(F)$ first, to extract the simple part of the constraint F_s . It solves the simple part using the auxiliary judgement $\vdash_s^* F : \phi$. It applies the substitution to the original constraint and tries to solve the returned constraint.⁷

Notice that the solver returns a ϕ substitution, which can bind both skolem variables and unification variables. As discussed in the previous section, being able to handle equalities between skolem variables is important for the interleaving of solving and constraint generation in rule (LET). Nevertheless, only a θ is returned the second time we attempt to solve the constraints. This is because the second time the solver will attempt to solve the proper implications that remain – and solutions to those may only bind unification variables as we shall shortly see (rule (S-PIMPL)).

The judgement $\vdash_s^* F : \phi$ is the core of our constraint solver. Rules (S-EMPTY) and (S-SPLIT) are straightforward. The remaining rules deal with a single equality constraint. Rule (S-CONS) deconstructs a type constructor application, and Rule (S-REFL) discharges trivial equality constraints. Rules (S-UL) and (S-UR) actually instantiate a type variable ν with a type τ . They must be careful not to violate the occurs-check ($\nu \notin \tau$).

Simple implication constraints, i.e. with empty given constraints, are treated by the (S-SIMPL) rule. A simple implication constraint is treated almost as if it were just a basic constraint with two differences. First, we make sure that the returned ϕ does not unify any of the skolemized variables of the constraint – it would be unsound to do otherwise. Second, we must never instantiate any of the variables capture in $[\bar{\alpha}]$ with a type that contains some of the skolemized variables \bar{b} . (In this case “untouchables” for the $\bar{\alpha}$ variables is a bad name. For example, it is fine – indeed essential – to unify α in $[\alpha]\forall b. \alpha \sim \text{Bool}$.)

Proper implication constraints are tackled by the (S-PIMPL) rule. First it computes ϕ that solves the assumptions C . Next, it applies it to F and solves F recursively yielding θ . Finally, it checks that the solution θ does not touch any of the untouchables.

There are several tricky points:

- There is some non-determinism in rule (S-SPLIT), but it is harmless. When solving simple constraints, the order of solving them does not matter; and when solving conjunctions of proper constraints solutions from one can never affect the other.
- Some non-determinism appears in (S-PIMPL). For example, consider the constraint $[\alpha]\forall. C \supset \alpha \sim \beta$. The recursive invocation of \vdash_s could return either $\phi = \{\alpha := \beta\}$ or $\{\beta := \alpha\}$, but only one will satisfy the untouchables check. In contrast, any most-general unifier of C will do for ϕ . Similarly, in a simple constraint $[\forall c. \beta \sim c]$ there is a choice to bind either β or c when we solve the constraint $\beta \sim c$. However, because of the conditions in rule (S-SIMPL), only the solution $\{\beta := c\}$ is acceptable.
- Rule (S-PIMPL) does not need the skolem-escape check that appears in (S-SIMPL). Because θ does not affect α , such a check cannot fail.
- In (S-PIMPL), solving C requires us to bind skolem variables as well as unification variables, and hence we return a ϕ . This is important to type constraints whose assumptions involve skolem variables, such as $(a \sim \text{Int} \supset \text{Int} \sim a)$. Furthermore, in rule (S-PIMPL) the solution of the right-hand side of the constraint is required to be a θ . The reason is because this

rule is triggered whenever we are trying to solve a proper implication constraint, and hence the solver is *not* called from rule (LET), but rather *after* constraint generation has finished. In order to solve such a constraint at the end, it is unsound to bind skolem variables: Equalities that involve skolems may only be discharged by given equalities. Hence we must not return a ϕ , but a substitution that binds unification variables only (i.e. a θ).

5.3 Example

Consider again our standard example

```
\x. case x of { T1 n -> n>0 }
```

for which the type $\alpha_x \rightarrow \beta$ is derived, and the constraint

$$F = \alpha_x \sim \text{T } \alpha \wedge [\alpha, \alpha_x, \beta](\alpha \sim \text{Bool} \supset \beta \sim \text{Bool})$$

If we solve first the simple constraint on the left, we get the substitution $[\alpha_x := \text{T } \alpha]$. We apply this substitution on the implication constraint, yielding $[\alpha_x, \beta](\alpha \sim \text{Bool} \supset \beta \sim \text{Bool})$. Next, we try to solve the implication constraint. Firstly, applying the mgu of $\alpha \sim \text{Bool}$, i.e. $[\alpha := \text{Bool}]$, to $\beta \sim \text{Bool}$ has no impact. Secondly, we try to solve $\beta \sim \text{Bool}$ by substituting β for Bool . Yet this fails, because β is an “untouchable”. Hence, our algorithm rejects the program.

Now let’s add a second branch to the example

```
\x. case x of { T1 n -> n>0 ; T2 xs -> null xs }
```

Again the type $\alpha_x \rightarrow \beta$ is derived, now with the constraint

$$F' = F \wedge \alpha_x \sim \text{T } \alpha' \wedge [\alpha'] \sim [\alpha''] \wedge \beta \sim \text{Bool}$$

The first additional constraint originates from the pattern $\text{T2 } xs$, the second and third from $\text{null } xs$. Solving all simple constraints first, we get the substitution $[\alpha_x := \text{T } \alpha, \alpha' := \alpha, \alpha'' := \alpha, \beta := \text{Bool}]$. These reduce the implication constraint to

$$[\alpha](\alpha \sim \text{Bool} \supset \text{Bool} \sim \text{Bool})$$

, which is now readily solved. Hence, the expression is accepted with type $\text{T } \alpha \rightarrow \text{Bool}$.

6. Specifying the restricted type system

It is all very well having an inference algorithm, but must also explain to the programmer which programs are accepted by the type checker and which are not. Every GADT inference algorithm has difficulty with this point, and ours is no exception.

Figure 6 presents the rules of the restricted type system. The top-level typing judgement is $C, \Gamma \vdash e : \tau$ which asserts that expression e has type τ with respect to environment Γ and type constraints C . This judgement is defined by rule (R-MAIN) which in turn is defined in terms of the auxiliary judgement

$$C, \Gamma \vdash_r e : \tau, P$$

which should be read “under constraints C and type environment Γ , the term e has type τ and suspended typing judgements P ”.

What are these suspended judgements? The idea is that we type-check the original term *simply ignoring* any GADT case alternatives. Instead, these ignored alternatives, along with the current environment and result type, are collected in a set P of tuples $\langle C, \Gamma, e, \tau \rangle$. Suppose the original top-level program term can be typed, so that

$$\Gamma \vdash_r e : \tau, P$$

holds. Then, for *every* such typing (or, more realistically for the principal typing) we require that *all* the suspended typing problems in P are soluble. That is what the (rather complicated) rule

⁷A more realistic implementation would split the constraint, solve the simple part and use that substitution to solve the proper part – no need to re-solve the simple part. We chose our current formalism as it saves us the definition of splitting.

$\boxed{C, \Gamma \vdash_R e : \tau}$		
$\text{(R-MAIN)} \frac{\text{consistent}(C) \quad C, \Gamma \vdash_r e : \tau, P \quad \forall \tau', P'. (C, \Gamma \vdash_r e : \tau', P') \Rightarrow \forall (C_i, \Gamma_i, e_i, \tau_i) \in P'. C_i, \Gamma_i \vdash_R e_i : \tau_i}{C, \Gamma \vdash_R e : \tau}$		
$\boxed{C, \Gamma \vdash_r e : \tau, P}$		
$\text{(R-VAR)} \frac{(x : \forall \bar{a}. v) \in \Gamma \quad \phi = \{\bar{a} := \bar{\tau}\}}{C, \Gamma \vdash_r x : \phi(v), \emptyset}$	$\text{(R-CON)} \frac{K :: \forall \bar{a}. D \Rightarrow v \quad \phi = \{\bar{a} := \bar{\tau}\} \quad C \models \phi(D)}{C, \Gamma \vdash_r K : \phi(v), \emptyset}$	$\text{(R-EQ)} \frac{C, \Gamma \vdash_r e : \tau_1, P \quad C \models \tau_1 \sim \tau_2}{C, \Gamma \vdash_r e : \tau_2, P}$
$\text{(R-APP)} \frac{C, \Gamma \vdash_r e_1 : \tau_1 \rightarrow \tau_2, P_1 \quad C, \Gamma \vdash_r e_2 : \tau_1, P_2}{C, \Gamma \vdash_r e_1 e_2 : \tau_2, P_1 \cup P_2}$	$\text{(R-ABS)} \frac{C, \Gamma \cup \{x : \tau_1\} \vdash_r e : \tau_2, P}{C, \Gamma \vdash_r \lambda x. e : \tau_1 \rightarrow \tau_2, P}$	
$\text{(R-LET)} \frac{C, \Gamma \cup \{g : \tau_1\} \vdash_r e_1 : \tau_1, P_1 \quad \bar{a} = f\nu(\tau_1) - f\nu(C, \Gamma) \quad C, \Gamma \cup \{g : \forall \bar{a}. \tau_1\} \vdash_r e_2 : \tau_2, P_2}{C, \Gamma \vdash_r \text{let } \{g = e_1\} \text{ in } e_2 : \tau_2, P_1 \cup P_2}$	$\text{(R-LETA)} \frac{C, \Gamma \cup \{g : \forall \bar{a}. \tau_1\} \vdash_r e_1 : \tau_1, P_1 \quad C, \Gamma \cup \{g : \forall \bar{a}. \tau_1\} \vdash_r e_2 : \tau_2, P_2}{C, \Gamma \vdash_r \text{let } \{g :: \forall \bar{a}. \tau_1 = e_1\} \text{ in } e_2 : \tau_2, P_1 \cup P_2}$	
$\text{(R-CASE)} \frac{C, \Gamma \vdash_r e : \tau_1, P \quad C, \Gamma \vdash_{rp} p_i \rightarrow e_i : \tau_1 \rightarrow \tau_2, P_i \text{ for } i \in I}{C, \Gamma \vdash_r \text{case } e \text{ of } [p_i \rightarrow e_i]_{i \in I} : \tau_2, P \cup \bigcup_{i \in I} P_i}$		
$\boxed{C, \Gamma \vdash_{rp} p \rightarrow e : \tau \rightarrow v, P}$		
$\text{(R-VPAT)} \frac{K : \forall \bar{a}, \bar{b}. \epsilon \Rightarrow v_1 \rightarrow \dots \rightarrow v_p \rightarrow T \bar{a} \quad f\nu(C, \Gamma, \bar{\tau}, \tau_r) \cap \bar{b} = \emptyset \quad \phi = \{\bar{a} := \bar{\tau}\} \quad C, \Gamma \cup \phi\{x_1 : v_1, \dots, x_p : v_p\} \vdash_r e : \tau_r, P}{C, \Gamma \vdash_{rp} K x_1 \dots x_p \rightarrow e : T \bar{\tau} \rightarrow \tau_r, P}$	$\text{(R-GPAT)} \frac{K : \forall \bar{a}, \bar{b}. D \Rightarrow v_1 \rightarrow \dots \rightarrow v_p \rightarrow T \bar{a} \quad D \neq \epsilon \quad f\nu(C, \Gamma, \bar{\tau}, \tau_r) \cap \bar{b} = \emptyset \quad \phi = \{\bar{a} := \bar{\tau}\} \quad P = \{(C \wedge \phi(D), \Gamma \cup \phi\{x_1 : v_1, \dots, x_l : v_l\}, e, \tau_r)\}}{C, \Gamma \vdash_{rp} K x_1 \dots x_l \rightarrow e : T \bar{\tau} \rightarrow \tau_r, P}$	

Figure 6. Typing Rules for the Restricted Type System

(R-MAIN) says. It ensures that typing information from inside a GADT match does not influence the typing of code outside that match — just as the algorithm does. Observe the recursive nature of rule (R-MAIN), which defers and processes nested case expressions one layer at a time.

The only rule that adds a deferred typings to P is R-GPAT; it defers the typing of a branch of a case expression that matches a GADT constructor pattern. This rule only applies to GADT constructors that bring a type equality into scope. In all other cases, when no new type equalities are brought into scope, the rule R-PAT applies, which does not defer the typing.

7. Formal properties

In this section we describe the properties of our type system and its inference algorithm.

7.1 Properties of the type system

As we have discussed, implication constraints arising from program text may have a finite or infinite set of incomparable solutions. This ambiguity makes type inference hard. Even in the case when the solutions are finite (but cannot be described by a common most general solution) modular type inference is impossible. Our restricted system however imposes conditions on the typeable programs of the unrestricted system, which ensure that we can perform tractable type inference without having to search the complete space of possibly incomparable solutions for the arising constraints.

First, the restricted type system is sound wrt. the unrestricted type system:

THEOREM 7.1 (Soundness). *If $\epsilon, \Gamma \vdash_R e : \tau$ in the restricted type system (Fig. 6), then $\epsilon, \Gamma \vdash e : \tau$ in the unrestricted type system (Fig. 2).*

It is fairly easy to see that this theorem holds. In addition to all the constraints of the unrestricted type system, the restricted type system imposes one more constraint on well-typing: the universal well-typing of the deferred typings discussed above.

Moreover, the restricted type system has the important property that it only admits expressions that have a principal type.

THEOREM 7.2 (Principal Typing in the Restricted Type System). *If an expression e is typeable in the restricted type system wrt. a type environment Γ , then there is a principal type τ_p such that $\epsilon, \Gamma \vdash_R e : \tau_p$ and such that for any other τ for which $\epsilon, \Gamma \vdash_R e : \tau$, there exists a substitution ϕ such that $\phi(\tau_p) = \tau$.*

Note that the principality is not an artifact of the restricted type system. A principal type in the restricted type system is also a principal type in the unrestricted type system:

THEOREM 7.3 (Principal Typing in the Unrestricted Type System). *Assume that the type τ_p is the principal type of e wrt. a type environment Γ in the restricted type system. Then, for any other τ for which $\epsilon, \Gamma \vdash e : \tau$, there exists a substitution ϕ such that $\phi(\tau_p) = \tau$.*

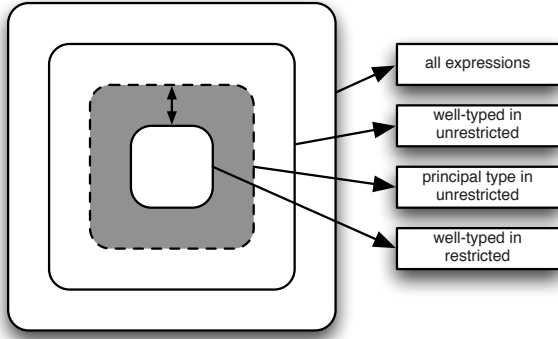


Figure 7. The space of programs

Note that not all programs with a principal type in the unrestricted type system, are accepted in the unrestricted type system.

Consider the following program:

```
data T a where MkT :: (a ~ Bool) => T a

f :: T a -> Char
f x = let h = case x of MkT -> 3
      in 'a'
```

The principal type for h in the unrestricted type system is Int . The restricted, ignoring the case branch, attempts to assign $\forall b.b$ as the principal type for h . However, this does not allow the implication $(a \sim \text{Bool}) \supset (b \sim \text{Int})$ to be solved. Hence, the program is rejected.

Hence, the gray area in Figure 7 is non-empty. We leave it as a challenge for future work to expand the innermost area towards the dashed line.

7.2 Properties of the inference algorithm

The solver algorithm has a number of vital properties. First, the search for solution always terminates, either in failure or success.

THEOREM 7.4 (Termination). *The solver algorithm terminates.*

Second, when a solution is found, it is a proper well-typing in the restricted type system.

THEOREM 7.5 (Soundness). *If $\vdash_{\text{inf}} e : \tau$ then $\epsilon, \emptyset \vdash_R e : \tau$*

Third, when a solution is found, it is not an arbitrary solution, but the principal solution.

THEOREM 7.6 (Principality). *The inferred type is the principal type in the restricted type system: If $\vdash_{\text{inf}} e : \tau$ and $\epsilon, \emptyset \vdash_R e : v$ then $v = \phi(\tau)$ for some ϕ .*

Finally, if an expression is well-typed, then the solver algorithm finds a solution.

THEOREM 7.7 (Completeness). *If $\epsilon, \emptyset \vdash_R e : \tau$ then $\vdash_{\text{inf}} e : v$.*

Of course, in order to prove the solver algorithm properties, we have to generalize appropriately the statements of the theorems in this section, but we refrain from presenting the generalized statements for the sake of clarity of exposition.

8. Implementation Aspects

A key property of **OutsideIn** is that it is easy to implement, and the implementation is efficient. To substantiate this claim we briefly

describe our implementation of **OutsideIn** in Haskell. Our implementation is available for download from

research.microsoft.com/people/dimitris/

and additionally supports bidirectional type checking, open type annotations, and type annotations with constraints.

We introduce a datatype `MetaTv` for unification variables α, β, \dots and a datatype `TyVar` for skolem variables a, b, \dots . As in traditional implementations (PVWS07), the `MetaTv` contains a reference cell that may contain a type to which the variable is bound:

```
data MetaTv = Meta Name (IORef (Maybe Type))
newtype TyVar = TyV Name
```

The main type checker is written in a monad `Tc a`, which is a function from environments `TcEnv` and encapsulates IO and threading of error messages.

```
newtype Tc a = Tc (TcEnv -> IO (Either ErrMsg a))
data TcEnv
  = TcEnv { var_env      :: Map Name Type
          , lie_env      :: IORef [Constraint]
          , untouched    :: [MetaTv]
          , ... }


```

Among other fields, the `TcEnv` environment contains a typing environment `var_env`, which is a map from term variable names to types. The field `lie_env` collects the set of `Constraint`s that arise during type inference.⁸ The `Constraint` datatype holds equality and implication constraints.

8.1 Constraint generation

In traditional implementations, unification variables are typically *eagerly* unified to types as type inference proceeds. In contrast, the algorithm of Figure 4 first generates (lots of) constraints, and then solves them, which is much less efficient. In our implementation we choose an intermediate path, which results in much more compact generated constraints. The environment `TcEnv` is equipped with the `untouchables` field, which records the untouchable variables. As type inference proceeds we perform eager unification by side effect in the usual way, *except* that we refrain from unifying a variable α from the untouchable set to a type τ . In that case, we defer the constraint $\alpha \sim \tau$, to be dealt with after constraint generation is finished. Hence, the unifier has signature:

```
unify :: Type -> Type -> Tc [Constraint]
```

It accepts, two types to unify, unifies them (perhaps using side effects on `MetaTvs` that are not untouchable), and returns a list of deferred equalities for variables that belong in the `untouchables` field of the environment.

How does the `untouchables` environment field get updated? Whenever we perform type inference for a pattern match clause with non-empty given equations, the main type checker:

1. extends the `untouchables` field with the unification variables of the scrutinee and the environment and the return type, as required by Figure 4,
2. performs type inference for the right-hand-side of the clause and returns the deferred constraints, and
3. defers an implication constraint whose right-hand side consists of the aforementioned deferred constraints.

⁸The name `lie_env` is folklore from type class implementations, where it stands for Local Instance Environment.

8.2 Constraint solving

During type inference we need to solve the generated constraints at two points: when the constraints for the complete program have been generated (rule (INFER), Figure 4), but also, more subtly, when we encounter a let-bound definition with no annotation (rule (LET), Figure 4) – in the latter case we must only solve the simple constraints.

Post-constraint-generation-solving After constraint generation is finished, the `lie_env` field holds the set of deferred constraints. At this point we may appeal to our constraint simplifier, which is written in a lightweight error-threading monad, implemented with Haskell’s `Either` datatype. By design, this monad is pure in the sense that it does not support in-place updating of `MetaTvs`.

```

solveConstraints :: [MetaTv] -> [CConstraint] ->
  Either SimplifierError ()
solveConstraints untch cs
  = do { let (simples, propers) = splitCConstrs cs
        ; subst <- solveSimples (Unif untch) simples
        ; spropers <- substToCConstrs subst propers
        ; mapM_ solveProper spropers }

```

The function `solveConstraints` accepts a list of untouchable variables and a list of constraints (`cs`)⁹ to simplify. It splits the constraints to `simples` and `propers`, solves the `simples`, applies the substitution to the `propers` (yielding `spropers`) and solves `spropers`. The function `solveConstraints` is the final step of type inference, and we need not return any value back – hence the `()` return type.

We will return to `solveSimples` and the meaning of the argument (`Unif untch`), but for now let us focus on the `solveProper` function:

```

solveProper :: CConstraint -> Either SimplifierError ()

```

The function `solveProper` accepts a single proper implication and solves it. Notice that, since the argument is a proper implication, its solution cannot affect anything in the environment and hence we may simply return `()` – the substitution in Figure 5, rule (S-PIMPL) is only there to enable clean statements and proofs of some formal properties.

We now turn to `solveSimples`, below:

```

solveSimples :: SimplifierMode -> [CConstraint] ->
  Either SimplifierError Substitution
solveSimples mode = foldM (solveSimple mode) emptySubst
solveSimple :: SimplifierMode ->
  Substitution -> CConstraint ->
  Either SimplifierError Substitution

```

The `Substitution` datatype denotes substitutions from *either* `MetaTv` or `TyVar` variables to types. Notice that `solveSimples` is defined as a *fold*, that starts-off with the empty substitution and updates its as it solves each simple constraint. In contrast, we use `mapM_` to solve each proper constraint independently in `solveConstraints`, because they cannot affect each other. The `SimplifierMode` argument to `solveSimple` stands for the mode of operation:

- If the flag is (`Unif untch`) then the returned `Substitution` binds *only* unification variables that do not appear in the list of untouchables, `untch`.¹⁰

⁹The `CConstraint` datatype is a “canonicalized” variant of `Constraint`, and we can ignore their differences below.

¹⁰We could in principle apply the returned substitution as side-effect but we chose to not do so in order to treat this case uniformly with the case when the flag is `All`, to be described next.

- If the flag is `All` then the returned `Substitution` binds in-variably skolem and unification variables. Notice that it would be *wrong* to apply this substitution to unification variables as a side-effect – for example it is definitely wrong in the context of solving the local assumptions of an implication constraint.

One reason we need the flag `All` is because `solveSimple` has to unify both skolem and unification variables when called on the *given* equalities of implication constraints. Concretely, here is the definition of `solveProper` (simplified):

```

solveProper (CImplicConstraint envs sks gs ws)
  = do { subst <- solveSimples All gs
        ; ws_cs <- substToCConstrs subst ws
        -- find untouchables from envs and subst
        ; let all_envs = ...
        ; solveConstraints all_envs ws_cs }

```

The datatype `CImplicConstraint envs sks gs ws` stands for the proper constraint $[\text{envs}] \forall \text{sks}. \text{gs} \supset \text{ws}$. Notice characteristically that the `givens` `gs` are unified using the `All` flag. Subsequently, the substitution is applied to `ws`, and the new untouchable variables are calculated (`all_envs`). Finally, `solveConstraints` is recursively called with the new list of untouchables. The function `solveConstraints` returns `()` but that’s all that we need when solving the right-hand side of proper implication constraints: Any solution for a proper implication constraint could only bind “internal” variables to that constraint (i.e. not in the untouchables) and consequently does not affect any other constraint.

It is because of this recursive call to `solveConstraints` from inside of solving a proper implication that we need to pass it the list of untouchable variables — our top-level call to `solveConstraints` is with empty untouchable variables.

Solving for let-bound definitions When type checking a let-bound definition, rule (LET) in Figure 4 requires that we solve the generated simple constraints. We already have the mechanism for doing so, via `solveSimples`. We we give below a code excerpt from type checking let bindings.

```

-- type check binding
...
-- call the simplifier
; (propers, phi) <- simplifyTo $
  do { ...
      -- cs: the constraints from type checking
      ; let (simples, propers) = splitCConstrs cs
          ; phi <- solveSimples All simples
          ; return (propers, subst) }
...
-- compute quantified vars
; let forall_tvts = ...
-- create new skolems
; sks <- ...
-- skolemizing substitution forall_tvts |-> sks
; let thetak = ...
; let phi_thetak = thetak 'composeSubst' phi
;   spropers = map (substToConstr phi_thetak) propers
-- write back constraints
; deferred <- eqCheckSubst phi_thetak
; deferImpl env_tvts sks [] (spropers ++ deferred)
-- quantify and return type
...

```

We first type check the binding and get the resulting constraints and its type. Subsequently, we call the simplifier: we split the constraints to `simples` and `propers`, we solve the `simples` (using mode `All`) and return the `propers` and the resulting substitution, `subst`. Next, we compute the variables to quantify and the skolemizing substitution θ_k of rule (LET) in Figure 4 (`thetak`).

Next, we need to extend the `lie_env` with a constraint. At this point we could in principle return the original constraint to which we have applied θ_k , wrapped as a simple implication constraint, as in rule (LET). As an optimization however, we call `unify` on each binding $\nu := \tau$ in `phi_thetak`; in the common case where ν is a (touchable) unification variable α `unify` will update α in-place, otherwise it will defer the constraints. Those deferred constraints are bound to `deferred`, and finally return a simple implication constraint that contains the skolemized proper part of the original (`sprobers`) and those `deferred`.

9. Related Work

Since GADTs have become popular there has been a flurry of papers on inference algorithms to support them in a practical programming language.

9.1 Fully-annotated programs

One approach is to assume that the program is fully type-annotated, i.e. each sub-expression carries explicit type information. Under this (strong) assumption, we speak of type *checking* rather than *inference*. Type checking boils down to unification which is decidable. Hence, we can conclude that type checking for GADTs is decidable. For example, consider (CH03) and (SP07).

9.2 Entirely unannotated programs

Type inference for unannotated programs turns out to be extremely hard. The difficulty lies in the fact that GADT pattern matches bring into scope local type assumptions (Section 2). Following the standard route of reducing type inference to constraint solving, GADTs require implication constraints to capture the inference problem precisely (SSS08).

Unification is no longer sufficient to solve such constraints. We require more complicated solving methods such as constraint abduction (Mah05) and E-unification (GNRS92). It is fairly straightforward to construct examples which show that no principal solutions (and therefore no principal types) exist. We can even conclude that GADT inference is undecidable by reduction to simultaneous rigid E-unification problem which is known to be undecidable (DV95).

How do previous inference approaches tackle these problems?

Simonet and Pottier (SP07) solve the inference problem by admitting (much) richer constraints. They sidestep the problems of undecidability and lack of principal types altogether by reducing type inference to type checking. Their inference approach only accumulates (implication) constraints and refrains from solving them. As a result, implications may appear in type schemes, which is a serious complication for the poor programmer (we elaborate in Section 5.1). Furthermore, no tractable solving algorithm is known for the constraints they generate, largely because of the (absolutely necessary) use of implications.

Sulzmann *et al* (SSS08) go the other direction, by keeping constraints (in types) simple, and instead apply a very powerful (abductive) solving mechanism. To avoid undecidability, they only consider a selected set of “intuitive” solutions. However they give only an inference algorithm, and it is not clear how to give a declarative description that specifies which programs are well-typed and which are not. Furthermore their system lacks principal types.

9.3 Practical compromises

We conclude that tractable type inference for completely unannotated programs is impossible. It is therefore acceptable to demand a certain amount of user-provided type information. We know of two well-documented approaches:

Régis-Gianas and Pottier stratify type inference into two passes. The first figures out the “shape” of types involving GADTs, while the second performs more-or-less conventional type inference (PRG06). Régis-Gianas and Pottier present two different shape analysis procedures, the *Wob* and *Inst* systems. The *Wob* system has similar expressiveness and need for annotation as in (PVWW06). The *Ibis* system on the other hand has similar expressiveness as our system, with a very aggressive iterated shape analysis process. This is reminiscent of our unification of simple constraints arising potentially from far-away in the program text, prior to solving a particular proper constraint. In terms of expressiveness, the vast majority of programs typeable by our system are typeable in *Ibis* but we conjecture that there exist programs typeable in our system not typeable in *Ibis*, because unification of simple (global) constraints may be able to figure more out about the types of expressions than the preprocessing shape analysis of *Ibis*. On the other hand, *Ibis* lacks a single declarative specification that does not force the programmer to understand the intricacies of shape propagation.

Peyton Jones *et al* require that the scrutinee of a GADT match has a “rigid” type, known to the type checker *ab initio*. A number of *ad hoc* rules describe how a type signature is propagated to control rigidity (PVWW06). Because rigidity analysis is more aggressive in our system we type many more programs than in (PVWW06), including the carefully-chosen Example 7.2 from (PRG06). On the other hand a program fails to type check in our approach if the type of a case branch is not determined by some “outer” constraint:

```
data Eq a b where { Refl :: forall a. Eq a a }

test :: forall a b. Eq a b -> Int
test x = let funny_id = \z -> case x of Refl -> z
        in funny_id 3
```

By contrast this program is typeable in (PVWW06). Arguably, though, this program *should* be rejected, because there are several incomparable types for `funny_id` (in the unrestricted system of Figure 2, including $\forall c.c \rightarrow c$ and $a \rightarrow b$).

The implementation of GHC is a slight variation that requires that the right-hand-side of a pattern match clause be typed in a rigid environment¹¹. Hence, it would reject the previous example. Our system is strictly more expressive than this variation:

```
test :: forall a b. Eq a b -> Int
test x = (\z -> case x of Eq -> z) 34
```

The above program would fail to type check in GHC, as the “wobbly” variable `z` cannot be used in the right-hand-side of a pattern match clause, but in our system it would be typeable because the “outer” constraint forces `z` to get type `Int`.

In both approaches, inferred types are maximal, but *not necessarily principal* in the unrestricted natural GADT type system. The choice for a particular maximal type over others relies on the *ad hoc* rigidity analysis or shape pre-processing. By contrast, in our system only programs that enjoy principal types in the unrestricted type system are accepted.

Moreover, in both approaches the programmer is required to understand an entirely new concept (shape or rigidity respectively), with somewhat complex and *ad hoc* rules (e.g. Figure 6 of (PRG06)). Nor is the implementation straightforward; for example, GHC’s implementation of (PVWW06) is known to be flawed in a way that is not trivial to fix (Appendix A).

¹¹ GHC’s algorithm is described in an Appendix to the online version of the paper, available from: <http://research.microsoft.com/people/simonpj/papers/gadt>

10. Further work

Although we have focused exclusively on GADTs, we intend to apply our ideas in the context of Haskell, and more specifically of the Glasgow Haskell Compiler. The latter embodies numerous extensions to Haskell 98, some of which are highly relevant. Notably, s data constructor can bring into scope a local type-class constraint:

```
class Eq a where { (==) :: a -> a -> Bool }
data D a where { D1 :: Eq a => a -> D a }

h : a -> D a -> Bool
h x (D1 y) = x==y
```

The pattern match on D1 brings the (Eq a) constraint into scope, which can be used to discharge the (Eq a) constraint that arises from the occurrence of (==). Note that D1 is not a GADT; it brings into scope no new type equalities. The same thing may happen with Haskell’s implicit parameters (LLMS00).

Since type inference for Haskell already involves gathering and solving type-class constraints, the constraint-gathering approach to inference is quite natural. The above extension to Haskell generalises the idea of *local* type constraints to constraints other than equalities, and these naturally map to the same implication constraints we need for GADTs.

More ambitiously, GHC also supports *indexed type families* and type-equality constraints between them (SJCS08). So we may write

```
type family F :: * -> *
type instance F Int = Int
type instance F [a] = F a
```

```
data E a where
  E1 :: (F a ~ Int) => a -> E a
```

Here, when we match on E1 we get the local constraint that $F a \sim \text{Int}$, which in turn gives rise to new questions for the solver (SJCS08).

Unsurprisingly, these extensions raise similar issues that we found with simple equality constraints. For example, it turns out that type classes suffer from the same lack of principal types as equality constraints (SSS06). Consider this function:

```
data T a where { MkT :: Eq a => T a }

f x y = case x of { MkT -> y==y } :: Bool
```

What type should be inferred for f ? Here are two, neither of which are more general than the other:

```
f ::  $\forall a. T a \rightarrow a \rightarrow \text{Bool}$ 
f ::  $\forall ab. \text{Eq } b \Rightarrow T a \rightarrow b \rightarrow \text{Bool}$ 
```

However, type classes cannot be treated *identically* to equality constraints. For example, consider this variation of f :

```
data T a where
  MkT :: Eq a => T a

f x y = case x of { MkT -> y } :: Bool
```

This will generate the constraint $[\beta_y](\text{Eq } a \supset \beta_y \sim \text{Bool})$, where $y : \beta_y$. In our treatment (thus far) of implication constraints we treat β_y as untouchable, so the constraint appears insoluble; but in fact the C part of the implication (Eq a in this case) cannot bring any equalities into scope, so it is perfectly fine to unify $[\beta_y := \text{Bool}]$; indeed, it would be unreasonable not to. We must modify our notion of “proper” constraints to ones whose C constraints include equalities. For the inference algorithm, this is straightforward; it is not so clear what the type system might look like.

References

- [CH03] J. Cheney and R. Hinze. First-class phantom types. TR 1901, Cornell University, 2003. <http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1901>.
- [DV95] A. Degtyarev and A. Voronkov. Simultaneous rigid E-unification is undecidable. In *Proc. of CSL’95*, volume 1092 of *LNCS*, pages 178–190. Springer-Verlag, 1995.
- [GNRS92] J. H. Gallier, P. Narendran, S. Raatz, and W. Snyder. Theorem proving using equational matings and rigid e-unification. *J. ACM*, 39(2):377–429, 1992.
- [LLMS00] J. R. Lewis, J. Launchbury, E. Meijer, and M. Shields. Implicit parameters: Dynamic scoping with static types. In *POPL*, pages 108–118, 2000.
- [Mah05] M. Maher. Herbrand constraint abduction. In *Proc. of LICS’05*, pages 397–406. IEEE Computer Society, 2005.
- [PR05] F. Pottier and D. Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
- [PRG06] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proc. of POPL’06*, pages 232–244. ACM Press, 2006.
- [PVWS07] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82, January 2007.
- [PVWW06] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. of ICFP’06*, pages 50–61. ACM Press, 2006.
- [SCPD07] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Proc. of TLDI’07*. ACM, 2007.
- [SJCS08] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. *SIGPLAN Not.*, 43(9):51–62, 2008.
- [SP07] V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Prog. Languages Systems*, 29(1), January 2007.
- [SSS06] M. Sulzmann, T. Schrijvers, and P. J. Stuckey. Principal type inference for GHC-style multi-parameter type classes. In *Proc. of APLAS’06*, volume 4279 of *LNCS*, pages 26–43. Springer-Verlag, 2006.
- [SSS08] M. Sulzmann, T. Schrijvers, and P. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Report CW 507, Department of Computer Science, K.U.Leuven, Leuven, Belgium, January 2008.

Note for the reviewers: The appendices are included for the convenience of the reviewers, but entirely optional for the understanding of the paper. They will be made available in a separate technical report in the case the paper is accepted.

A. Rigidity

GHC's implementation of rigidity is deficient. This section documents that fact.

```
data T a where
  K :: T Int

g :: Int -> Int
h :: T Int -> Int

-- f1 y x = (g x, g (case y of K -> x + 1))
-- f2 y x = (g (case y of K -> x + 1), g x)
f3 y x = (h y, g x, g (case y of K -> x + 1))
f4 y x = (h y, g (case y of K -> x + 1), g x)
```

GHC 6.9.20080910, accepts f3 and f4, but not f1 and f2 because y has a non-rigid type. It does not complain about x. Our algorithm accepts all 4 functions, with y getting type T a in f1 and f2.

B. Loss of principal types

Here is a GADT program which can be given an infinite set of maximal types. A type is *maximal* if there is no other more general type.

```
data Erk x y z where
  K :: x -> y -> Erk x y Char

f (K x y) = x:y
```

When pattern matching over $K\ x\ y$ we may make use of the fact that z has type `Char`. Among others, function f can be given the following set of types

$$f :: \forall t_z. \text{Erk } [t_z]^n [\text{Char}]^{n+1} t_z \rightarrow [\text{Char}]^{n+1}$$

for any $n \geq 0$. We write $[t]^n$ to denote a list of lists of ... lists (depth n) of t 's. It also has the type

$$f :: \forall t_x, t_z. \text{Erk } t_x [t_x] t_z \rightarrow [t_x]$$

All these types are incomparable, and there is no other type for f which is more general than any of the above types.

The GADT type system is less well-behaved than the Hindley-Milner (HM) system in a second, more subtle way. From HM we are used that a typing derivation is principal iff all its sub-derivations are principal. This is the key to obtain a complete HM type inference algorithm. In the case of GADTs, this property is lost.

Consider the following variant of the above example:

```
data Erk x y z where
  K :: x -> y -> Erk x y Char
  L :: x -> y -> Erk x y Bool

g = let f (K x y) = x:y
      f (L x y) = y
      in f (L [True] [['a']])
```

The subexpression $(L\ [True]\ [['a']])$ has type

$$\text{Erk } [\text{Bool}] [[\text{Char}]] \text{Bool}$$

Function f can be given the type

$$\forall t_z. \text{Erk } [t_z] [[\text{Char}]] t_z \rightarrow [[\text{Char}]]$$

which is maximal but not principal (see above). Hence, the main expression $f\ (L\ [True]\ [['a']])$ has type $[[\text{Char}]]$. Hence, the outermost expression g has the principal type $[[\text{Char}]]$.

The other maximal type of f ,

$$\forall t_x, t_z. \text{Erk } t_x [t_x] t_z \rightarrow [t_x]$$

, yields the same type for g .

C. Proof of Principality: Theorem 7.2

Proof The principality of the \vdash_R relation follows from the principality of the \vdash_r relation.

LEMMA C.1 (Principality of Deferring Typing). *If there is a typing $\epsilon, \Gamma \vdash_r e : \tau_1, P_1$, then there exists a principal typing $\epsilon, \Gamma \vdash_r e : \tau, P$ such that for any other typing $\epsilon, \Gamma \vdash_r e : \tau_2, P_2$ there exists a substitution ϕ such that $\phi(\tau) = \tau_2$ and $\phi(P) = P_2$.*

Proof Assume that we have two distinct types τ_1, τ_2 for a given expression e and environment Γ , such that $\epsilon, \Gamma \vdash_r e : \tau_1, P_1$ and $\epsilon, \Gamma \vdash_r e : \tau_2, P_2$.

Let τ be the least common generalization of τ_1 and τ_2 , such that $\tau_1 = \theta_1(\tau)$ and $\tau_2 = \theta_2(\tau)$ with θ_1 and θ_2 type variable substitutions. Similarly for P versus P_1 and P_2 .

We will show that $\epsilon, \Gamma \vdash_r e : \tau, P$ also holds. Hence, in general the least common generalization of two well-typings is also a well-typing. Since the least common generalization relation is well-founded, there must be a principal type as a consequence.

We discern the following cases:

- If $\tau \equiv \tau_1$ or $\tau \equiv \tau_2$, then τ is obviously a well-typing.
- If $\epsilon, \Gamma \vdash_r e : \tau, P$, then the lemma holds.
- Otherwise, we have that $\epsilon, \not\vdash_r e : \tau, P$. Consider that the given constraint component C is empty (ϵ) as is the program theory P . Hence, the existing derivations for $\epsilon, \vdash_r e : \tau_1, P_1$ and $\epsilon, \vdash_r e : \tau_2, P_2$, for some P_1 and P_2 are fully syntax directed; rule (R-EQ) cannot be applied.

Let us consider the two existing syntax-directed derivations side-by-side inductively.

Base case is:

- (R-VAR): We know that $\epsilon, \theta_1(\Gamma) \vdash_r x : \theta_1(\tau), \emptyset$ and $\epsilon, \theta_2(\Gamma) \vdash_r x : \theta_2(\tau), \emptyset$ hold, with $(x : \forall \bar{a}. D \Rightarrow \tau')$ and $\theta_1(\tau) = [\bar{\tau}_1/\bar{a}]\tau'$ and $\theta_2(\tau) = [\bar{\tau}_2/\bar{a}]\tau'$. Because τ is the least common generalization, we have that $\tau = [\bar{\tau}/\bar{a}]\tau'$ where $\theta_1(\bar{\tau}) = \bar{\tau}_1$ and $\theta_2(\bar{\tau}) = \bar{\tau}_2$.

Assume that $\epsilon, \Gamma \vdash_r x : \tau, \emptyset$ does not hold. This can only be the case if $\not\vdash [\bar{\tau}/\bar{a}]D$. However, since $\models [\bar{\tau}_1/\bar{a}]D$ and $\models [\bar{\tau}_2/\bar{a}]D$ and τ is the least common generalization, this cannot be the case.

Recursive cases are:

- (R-ABS) We know that $\epsilon, \theta_1(\Gamma) \vdash_r \lambda x. e : \tau_{1,1} \rightarrow \tau_{1,2}, P_1$, and $\epsilon, \theta_2(\Gamma) \vdash_r \lambda x. e : \tau_{2,1} \rightarrow \tau_{2,2}, P_2$ with $\theta_1(\tau) \equiv \tau_{1,1} \rightarrow \tau_{1,2}$ and $\theta_2(\tau) \equiv \tau_{2,1} \rightarrow \tau_{2,2}$. Hence, $\tau \equiv \tau_1 \rightarrow \tau_2$ with τ_1 and τ_2 the least common generalizations of the corresponding components, with θ_1 and θ_2 the corresponding substitutions.

Because of induction, $\epsilon, \Gamma \cup \{x : \tau_1\} \vdash_r e : \tau_2, P$ must also hold. Hence, $\epsilon, \Gamma \vdash_r \lambda x. e : \tau, P$ holds.

- (R-APP) We know that $\epsilon, \theta_1(\Gamma) \vdash_r e_1 e_2 : \theta_1(\tau), P_{1,1} \cup P_{1,2}$, and $\epsilon, \theta_2(\Gamma) \vdash_r e_1 e_2 : \theta_2(\tau), P_{2,1} \cup P_{2,2}$, as well as $\epsilon, \theta_1(\Gamma) \vdash_r e_1 : \tau_1 \rightarrow \theta_1(\tau), P_{1,1}$, $\epsilon, \theta_1(\Gamma) \vdash_r e_2 : \tau_1, P_{1,2}$, and $\epsilon, \theta_2(\Gamma) \vdash_r e_1 : \tau_2 \rightarrow \theta_2(\tau), P_{2,1}$, $\epsilon, \theta_2(\Gamma) \vdash_r e_2 : \tau_2, P_{2,2}$.

Let τ' be the least common generalization of τ_1 and τ_2 , and θ'_1 and θ'_2 the corresponding extensions of θ_1 and θ_2 . The rest follows from induction, as in the case for (R-ABS).

- The other cases are similar in style.

D. Proof of Unrestricted Principality:

Theorem 7.3

Proof Let τ_1 be the principal type of an expression e wrt. type environment Γ in the restricted type system. Assume that τ_2 is a type of e in the unrestricted type system, but not the restricted type system, that is not an instance of τ_1 .

We must have that $\epsilon, \Gamma \vdash_r e : \tau_2, P_2$ for some P_2 (1), as the rules \vdash_r are strictly weaker than those for the unrestricted type system (\vdash). Moreover, since τ_1 is a type in the restricted type system, it must satisfy rule (R-MAIN). From this we know that (2):

$$\forall \tau', P'. (\epsilon, \Gamma \vdash_r e : \tau', P') \Rightarrow \forall \langle C_i, \Gamma_i, e_i, \tau_i \rangle \in P'. C_i, \Gamma_i \vdash_R e_i : \tau_i$$

From (1) and (2) It follows that τ_2 satisfies the whole antecedent of rule (R-MAIN) itself. Hence, τ_2 must be a well-typing in the restricted type system. This contradicts our earlier assumption. Hence the assumption is invalid, and the theorem holds.

E. Proof of Solver Termination: Theorem 7.4

Proof The solver algorithm terminates for any given constraint F . At any time we only have a finite number of possible choices for applying the rules. Each rule reduces the number of unification variables, or, if not that, decreases a (type) term size, or, if neither of these two, the number of constraints.

Formally, we assign the following norm $|\cdot|$ to a wanted constraint F :

$$|F| = \langle |F|_v, |F|_i, |F|_s, |F|_c \rangle$$

i.e. it is a tuple of four different norms. The ordering \preceq on this norm is lexicographic:

$$\begin{aligned} \langle v_1, s_1, c_1 \rangle \preceq \langle v_2, s_2, c_2 \rangle &\Leftrightarrow \\ &v_1 \preceq_v v_2 \\ &\vee (v_1 = v_2 \wedge i_1 \preceq_i i_2) \\ &\vee (v_1 = v_2 \wedge i_1 = i_2 \wedge s_1 \preceq_s s_2) \\ &\vee (v_1 = v_2 \wedge i_1 = i_2 \wedge s_1 = s_2 \wedge c_1 \preceq_c c_2) \end{aligned}$$

The ordering is well-founded, and, as we will show next, each rule application decreases the norm. Hence, the solver algorithm must terminate.

The first norm, $|F|_v$, counts the number of unification variables:

$$|F|_v = \#fuw(F)$$

and its ordering is that of the natural numbers:

$$v_1 \preceq_v v_2 \Leftrightarrow v_1 \leq v_2$$

Observe that this norm is reduced by the substitution produced in the rules (S-UNIFL) and (S-UNIFR). If the number of unification variables goes down to 0, these rules are no longer applicable. Hence, an infinite derivation, cannot involve an infinite number of applications of this rule. None of the rules creates fresh unification variables.

The second norm, $|F|_i$, is the implication count:

$$\begin{aligned} |\epsilon|_i &= 0 \\ |\tau_1 \sim \tau_2|_i &= 0 \\ |F_1 \wedge F_2|_i &= |F_1|_i + |F_2|_i \\ |[\bar{\alpha}](\bar{b}.C \supset F)|_i &= 1 + |F|_i \end{aligned}$$

and its ordering is that of the natural numbers:

$$i_1 \preceq_i i_2 \Leftrightarrow i_1 \leq i_2$$

Observe that the rules (S-IMPL) and (S-DIMPL) both decrease the implication count. No rule increases this number.

The third norm, $|F|_s$, is the *multiset* of term sizes:

$$\begin{aligned} |\epsilon|_s &= \emptyset \\ |\tau_1 \sim \tau_2|_s &= |\tau_1|_t \uplus |\tau_2|_t \\ |F_1 \wedge F_2|_s &= |F_1|_s \uplus |F_2|_s \\ |[\bar{\alpha}](\bar{b}.C \supset F)|_s &= |F|_s \end{aligned}$$

where \uplus is multiset union. The ordering of the multiset is defined as:

$$\begin{aligned} s_1 \preceq_s s_2 &\Leftrightarrow s_1 = s_2 = \emptyset \\ &\vee m_1 < m_2 \\ &\vee m_1 = m_2 \wedge s_1 - \{m_1\} \preceq_s s_2 - \{m_2\} \end{aligned}$$

where m_1 and m_2 are the maximal elements of s_1 and s_2 . Observe that rule (S-CONS) strictly decreases the multiset norm. It replaces two bigger terms by many smaller terms. Rule (S-SPLIT) may increase the multiset norm if the substitution θ_1 is non-empty. However, as a consequence, the number of constraints must have been reduced (from $|F_1 \wedge F_2|_c$ to $|F_2|_c$), and, more importantly, the number of unification variables must have gone down (from $\#fuw(F_1 \wedge F_2)$ to $\#fuw(\theta_1(F_2))$). Similarly, the rule (S-IMPL) may increase the multiset norm, but it decreases the implication count.

The fourth norm, $|F|_c$, is the constraint count:

$$\begin{aligned} |\epsilon|_c &= 0 \\ |\tau_1 \sim \tau_2|_c &= 1 \\ |F_1 \wedge F_2|_c &= |F_1|_c + |F_2|_c \\ |[\bar{\alpha}](\bar{b}.C \supset F)|_c &= |F|_c \end{aligned}$$

and its ordering is that of the natural numbers:

$$c_1 \preceq_c c_2 \Leftrightarrow c_1 \leq c_2$$

Observe that the rules (S-SPLIT), (S-CONS), (S-REFL), (S-UNIFL) and (S-UNIFR) all decrease this norm. Only rule (S-CONS) may increase it, but, as we have already observed above, it increases the multiset norm.

F. Proof of Solver Soundness and Principality: Theorems 7.5 and 7.6

F.1 Constraint Solving Lemmas

LEMMA F.1 (Constraint Solving Soundness). *The substitution θ produced by solving the constraint $\text{simple}(F)$ is a solution:*

$$\forall F, \theta : (\vdash_s \text{simple}(F) : \theta) \Rightarrow (\epsilon \models \theta(\text{simple}(F)))$$

Proof We conduct the proof by induction.

The **base cases** are the rules:

- (S-REFL): We know:

$$\vdash_s \tau \sim \tau, \emptyset$$

We have to show that:

$$\epsilon \models \tau \sim \tau$$

This follows from the (REFL) rule in the equality theory.

- (S-UNIFL): We know:

$$\vdash_s \alpha \sim \tau, \{\alpha := \tau\}$$

with $\alpha \notin \tau$. We have to show that:

$$\epsilon \models \{\alpha := \tau\}(\alpha \sim \tau)$$

Applying the substitution, we get:

$$\epsilon \models \tau \sim \tau$$

This follows again from the (REFL) rule in the equality theory.

- (S-UNIFR): This is similar to rule (S-UNIFL).

The **inductive cases** are the other rules:

- (S-SPLIT): We know that:

$$\begin{aligned} \vdash_s C_1 \wedge C_2 : \theta_1 \wedge \theta_2 \\ \vdash_s C_1 : \theta_1 \\ \vdash_s \theta_1(C_2) : \theta_2 \end{aligned}$$

Moreover, because of the inductive assumption, we know that:

$$\begin{aligned} \epsilon \models \theta_1(C_1) \\ \epsilon \models (\theta_1 \cup \theta_2)(C_2) \end{aligned}$$

We have to show that:

$$\epsilon \models (\theta_1 \cup \theta_2)(C_1 \wedge C_2)$$

Applying rule (CONJ) from the equality theory, we get:

$$\begin{aligned} \epsilon \models (\theta_1 \cup \theta_2)(C_1) \\ \epsilon \models (\theta_1 \cup \theta_2)(C_2) \end{aligned}$$

The latter corresponds to the second assumption. The former easily follows from the first assumption.

- (S-CONS): We know that:

$$\begin{aligned} \vdash_s T \bar{\tau} \sim T \bar{\tau}' : \theta \\ \vdash_s \bigwedge_i \tau_i \sim \tau'_i : \theta \end{aligned}$$

Moreover, because of the inductive assumption, we know that:

$$\epsilon \models \theta(\bigwedge_i \tau_i \sim \tau'_i)$$

We have to show that:

$$\epsilon \models \theta(T \bar{\tau} \sim T \bar{\tau}')$$

Applying rule (STRUCT) from the equality theory, we get:

$$\forall i. \epsilon \models \theta(\tau_i \sim \tau'_i)$$

By repeated reverse application of rule (CONJ) from the equality theory, we get the inductive assumption.

LEMMA F.2 (Constraint Solving Principality). *The solution θ of constraint C obtained through constraint solving is more general than any other solution θ' .*

$$\forall F, \theta, \theta' : (\epsilon \models \theta'(simple(F))) \wedge (\vdash_s simple(F) : \theta) \implies \exists \theta''. \theta \cdot \theta'' \equiv \theta'$$

Proof The proof is easy to establish. It deviates little from that of principality for simple Herbrand equality solving.

LEMMA F.3 (Constraint Solving Completeness). *If constraint $simple(F)$ has a solution θ , then constraint solving produces one:*

$$\forall F, \theta : (\epsilon \models \theta(simple(F))) \implies \exists \theta'. (\vdash_s simple(F) : \theta')$$

Proof The proof is easy to establish. It deviates little from that of completeness for simple Herbrand equality solving.

F.2 Deferring Typing Lemmas

LEMMA F.4 (Deferring Soundness). *The simple generated constraints yield a solution for the deferring typing:*

$$\begin{aligned} \forall e, \Gamma, \tau, F, \theta. (\Gamma \vdash_W e : \tau, F) \wedge (\vdash_s simple(F) : \theta) \\ \implies \exists P. \epsilon, \theta(\Gamma) \vdash_r e : \theta(\tau), P \end{aligned}$$

Proof We conduct our proof by induction on e .

The **base case** is:

- x : We have:

$$\begin{aligned} (x : \forall \bar{a}. C \Rightarrow \tau) \in \Gamma \\ \bar{\alpha} \text{ fresh} \\ \theta' \equiv [\bar{a} := \bar{\alpha}] \\ \Gamma \vdash_W x : \theta'(\tau), \theta'(C) \\ \vdash_s \theta'(C) : \theta \end{aligned}$$

We have to show that:

$$\exists P. \epsilon, \theta(\Gamma) \vdash_r x : \theta(\theta'(\tau)), P$$

Let $P \equiv \emptyset$, then we can show:

$$\epsilon, \theta(\Gamma) \vdash_r x : \theta(\theta'(\tau)), \emptyset$$

by means of rule (R-VAR). The first antecedent of this rule, $\theta(x : \forall \bar{a}. C \Rightarrow \tau) \in \Gamma$, is already assumed. The second antecedent is:

$$\epsilon \models \theta(\theta'(C))$$

It follows from applying Lemma F.1 to the assumption $\vdash_s \theta'(C) : \theta$.

The **inductive cases** are:

- $\lambda x. e$: We have that:

$$\begin{aligned} \alpha \text{ fresh} \\ \Gamma \cup \{x : \alpha\} \vdash_W e : \tau, F \\ \Gamma \vdash_W \lambda x. e : \alpha \rightarrow \tau, F \\ \vdash_s simple(F) : \theta \\ \exists P'. \epsilon, \theta(\Gamma) \cup \{x : \theta(\alpha)\} \vdash_r e : \theta(\tau), P' \end{aligned}$$

We have to show that:

$$\exists P. \epsilon, \theta(\Gamma) \vdash_r \lambda x. e : \theta(\alpha \rightarrow \tau), P$$

Let $P \equiv P'$, then we can show the above by means of rule (R-ABS). The antecedent we must show is:

$$\epsilon, \theta(\Gamma) \cup \{x : \theta(\alpha)\} \vdash_r e : \theta(\tau), P'$$

which is already assumed.

- $e_1 e_2$: We have that:

$$\begin{aligned} \Gamma \vdash_W e_1 : \tau_1, F_1 \\ \Gamma \vdash_W e_2 : \tau_2, F_2 \\ \alpha \text{ fresh} \\ \Gamma \vdash_W e_1 e_2 : \alpha, F_1 \wedge F_2 \wedge (\tau_1 \sim \tau_2 \rightarrow \alpha) \\ \vdash_s simple(F_1) \wedge simple(F_2) \wedge (\tau_1 \sim \tau_2 \rightarrow \alpha) : \theta \end{aligned}$$

We must show that:

$$\exists P. \epsilon, \theta(\Gamma) \vdash_r e_1 e_2 : \theta(\alpha), P$$

Because of the principality and completeness of constraint solving, we have from the fourth assumption:

$$\exists \theta_i, \theta'_i. (\vdash_s simple(F_i) : \theta_i) \wedge \theta \equiv \theta_i \cdot \theta'_i$$

where $i \in \{1, 2\}$. Moreover, because of the inductive assumption, we also have:

$$\exists P_i. \epsilon, \theta(\Gamma) \vdash_r e_i : \theta_i(\tau_i), P_i$$

Let $P \equiv \theta'_1(P_1) \cup \theta'_2(P_2)$, then we must actually show:

$$\epsilon, \theta(\Gamma) \vdash_r \lambda e_1 e_2 : \theta(\alpha), P$$

by means of rule (R-APP). The second antecedent for this is:

$$\epsilon, \theta(\Gamma) \vdash_r e_2 : \theta(\tau_2), \theta'_2(P_2)$$

which follows from the prior assumption. Similar for the first antecedent, but now we must also take into account that $\theta(\tau_1 \sim \tau_2 \rightarrow \alpha)$ holds.

- let $\{g = e_1\}$ in e_2 : We have:

$$\begin{aligned}
& \alpha \text{ fresh} \\
& \Gamma \cup \{g : \alpha\} \vdash_W e_1 : \tau_1, F_1 \\
& F'_1 = F_1 \wedge \alpha \sim \tau_1 \\
& F_s = \text{simple}(F'_1) \\
& \vdash_s F_s : \theta_1 \\
& \bar{\beta} = \text{fiv}(\theta_1(\tau_1)) - \text{fiv}(\theta_1(\Gamma)) \\
& \bar{b} \text{ fresh} \\
& \theta' = [\bar{\beta} := \bar{b}] \\
& F \cup \{g : \forall \bar{b}. \theta'(\theta_1(\tau_1))\} \vdash_W e_2 : \tau_2, F_2 \\
& \Gamma \vdash_W \text{let } \{g = e_1\} \text{ in } e_2 : \tau_2, \theta'(F'_1) \wedge F_2 \\
& \vdash_s \text{simple}(\theta'(F'_1) \wedge F_2) : \theta
\end{aligned}$$

We must show that:

$$\exists P. \epsilon, \theta(\Gamma) \vdash_r \text{let } \{g = e_1\} \text{ in } e_2 : \theta(\tau_2), P$$

Firstly, because of the induction hypothesis, we have that (a):

$$\exists P_1. \epsilon, \theta_1(\Gamma) \cup \{g : \theta_1(\alpha)\} \vdash_r e_1 : \theta_1(\tau_1), P_1$$

as well as: $\exists \theta'_1. (\theta_1 \cdot \theta' \cdot \theta'_1) \equiv \theta$. Secondly, because of the induction hypothesis, we also have that (b):

$$\begin{aligned}
& \exists \theta_2, \theta'_2. \\
& (\exists P_2. \epsilon, \theta_2(\Gamma) \cup \{g : \theta_2(\forall \bar{b}. \theta'(\theta_1(\tau_1)))\} \vdash_r e_2 : \theta_2(\tau_2), P_2) \\
& \wedge \theta_2 \cdot \theta'_2 \equiv \theta_2
\end{aligned}$$

Let $P \equiv \theta'_1(\theta'(P_1)) \cup \theta'_2(P_2)$ and then apply rule (R-LET) to the proof obligation. We get three new obligations:

$$\epsilon, \theta(\Gamma) \cup \{g : \theta(\theta'(\theta_1(\tau_1)))\} \vdash_r e_1 : \theta(\theta'(\theta_1(\tau_1))), \theta(P_1) \quad (1)$$

$$\bar{b} = \text{fv}(\theta(\theta'(\theta_1(\tau_1)))) - \text{fv}(\theta(\Gamma)) \quad (2)$$

$$\epsilon, \theta(\Gamma) \cup \{g : \forall \bar{b}. \theta(\theta'(\theta_1(\tau_1)))\} \vdash_r e_2 : \theta(\tau_2), \theta(P_2) \quad (3)$$

The first of these follows from (a). The second of these is satisfied by the definition of \bar{b} . The third of these follows from (b).

- let $\{g :: \forall \bar{a}. \tau_1\}$ in e_2 : We have:

$$\begin{aligned}
& \Gamma \cup \{g : \forall \bar{a}. \tau_1\} \vdash_W e_1 : \tau'_1, F_1 \\
& \Gamma \cup \{g : \forall \bar{a}. \tau_1\} \vdash_W e_2 : \tau_2, F_2 \\
& \Gamma \vdash_W \text{let } \{g :: \tau_1 = e_1\} \text{ in } e_2 : \tau_2, F_2 \wedge [\text{fiv}(\Gamma)](\forall \bar{a}. \epsilon \supset F_1 \wedge \tau_1 \sim \tau'_1) \\
& \vdash_s \text{simple}(F_2 \wedge [\text{fiv}(\Gamma)](\forall \bar{a}. \epsilon \supset F_1 \wedge \tau_1 \sim \tau'_1)) : \theta
\end{aligned}$$

We must show that:

$$\exists P. \epsilon, \theta(\Gamma) \vdash_r \text{let } \{g :: \tau_1 = e_1\} \text{ in } e_2 : \theta(\tau_2), P$$

Because of the first and third assumption, using the induction hypothesis, we can conclude:

$$\exists \theta_1, \theta'_1. (\theta'_1 \cdot \theta_1 \equiv \theta'_1) \wedge (\exists P_1. \epsilon, \theta_1(\Gamma \cup \{\{g : \forall \bar{a}. \tau_1\}\}) \vdash_r e_1 : \theta_1(\tau'_1), P_1)$$

If we apply the substitution θ_1 to the rightmost conjunct, we get because of the leftmost conjunct:

$$(\exists P_1. \epsilon, \theta(\Gamma \cup \{\{g : \forall \bar{a}. \tau_1\}\}) \vdash_r e_1 : \theta(\tau'_1), P_1)$$

Because θ is a solution for $\tau_1 \sim \tau'_1$, the above is equal to:

$$(\exists P_1. \epsilon, \theta(\Gamma \cup \{\{g : \forall \bar{a}. \tau_1\}\}) \vdash_r e_1 : \theta(\tau_1), P_1)$$

which is the second antecedent of the (R-LETA) rule. The Third antecedent follows in a similar manner. Finally, $\bar{a} = \text{fv}(\theta(\tau_1)) - \text{fv}(\theta(\Gamma))$ holds by construction. Hence, the (R-LETA) rule discharges the proof obligation.

- case e of $[p_i \rightarrow e_i]_{i \in I}$: The proof is similar in style to the previous cases. However, it relies on mutual induction with the Deferring Branch Soundness.

LEMMA F.5 (Deferring Branch Soundness). *The simple generated constraints yield a solution for the deferring branch typing:*

$$\begin{aligned}
& \forall p, e, \Gamma, \tau_p, \tau_e, F, \theta. \\
& (\Gamma \vdash_W p \rightarrow e : \tau_p \rightarrow \tau_e, F) \wedge (\vdash_s \text{simple}(F) : \theta) \\
& \implies \exists P. \epsilon, \theta(\Gamma) \vdash_r p \rightarrow e : \theta(\tau_p \rightarrow \tau_e), P
\end{aligned}$$

Proof The proof proceeds by mutual induction with the previous lemma. This proof only consists of inductive cases. The base cases only arise in the proof of the previous lemma.

The pattern p is of the form $K x_1 \dots x_p$ where K has the signature $\forall \bar{a}, \bar{b}. D \Rightarrow \tau_1 \rightarrow \dots \rightarrow \tau_p \rightarrow T \bar{a}$.

We distinguish two cases based on D :

- $D \equiv \epsilon$: The constraint F is a degenerate implication constraint. This is handled by rule (S-SIMPL) in the solver. The solver ensures that none of the type variables \bar{b} ends up in the environment as is required by rule (R-VPAT). The rest of rule (R-VPAT) follows by induction.
- $D \neq \epsilon$: The constraint F is a genuine implication constraint. Hence the substitution θ is empty. It is trivial to see that the lemma holds.

LEMMA F.6 (Deferring Principality). *The simple generated constraints yield a principal solution for the deferring typing:*

$$\begin{aligned}
& \forall e, \Gamma, \tau, F. (\Gamma \vdash_W e : \tau, F) \wedge (\vdash_s \text{simple}(F) : \theta) \\
& \implies \exists P. \epsilon, \Gamma \vdash_r e : \theta(\tau), P \wedge \\
& \quad (\forall \tau', P', \theta'. \epsilon, \theta'(\Gamma) \vdash_r e : \tau', P') \\
& \implies \exists \theta''. (\theta'' \cdot \theta \equiv \theta') \wedge (\models \theta'(\tau) \sim \tau') \wedge \theta''(P) \equiv P'
\end{aligned}$$

Proof This is easy but tedious to see.

LEMMA F.7 (Monotonicity Lemma).

$$\forall C, \Gamma, e, \tau. C, \Gamma \vdash_r e : \tau \implies \forall \theta. \theta(C), \theta(\Gamma) \vdash_r e : \theta(\tau)$$

Proof This is easy but tedious to see.

F.3 Deferred Typing Lemmas

Genuine Implications The genuine implications $\text{pimpl}(F)$ of a constraint F is defined as:

$$\begin{aligned}
\text{pimpl}(C) &= \epsilon \\
\text{pimpl}(F_1 \wedge F_2) &= \text{pimpl}(F_1) \wedge \text{pimpl}(F_2) \\
\text{pimpl}(\forall \bar{b}. F) &= \text{pimpl}(F) \\
\text{pimpl}(\forall \bar{b}. C \supset F) &= \forall \bar{b}. C \supset F \quad , C \neq \epsilon
\end{aligned}$$

LEMMA F.8 (Deferred Soundness).

$$\begin{aligned}
& \forall \Gamma, e, \tau, F, \theta_s, \theta_p, P : \\
& (\Gamma \vdash_W e : \tau, F) \\
& \wedge (\vdash_s \text{simple}(F) : \theta_s) \\
& \wedge (\epsilon, \theta_s(\Gamma) \vdash_r e : \theta_s(\tau), P) \\
& \wedge (\vdash_s \theta_s(\text{pimpl}(F)) : \theta_p) \\
& \implies \forall \langle C_i, \Gamma_i, e_i, \tau_i \rangle \in P : \\
& \quad (C_i, \Gamma_i \vdash_r e_i : \tau_i)
\end{aligned}$$

Proof The proof connects the generated implication constraints to the deferred typings. It relies on the subsequent auxiliary lemma.

LEMMA F.9 (Consistency). *Any well-typing in \vdash_R requires that the given constraint C is consistent.*

$$\forall C, \Gamma, e, \tau. (C, \Gamma \vdash_R e : \tau) \implies \text{consistent}(C)$$

Proof This is so for the top-level typing judgement, where $C = \epsilon$ and consistency is trivial. The constraint C is only extended in the (R-GPAT) rule, where consistency is enforced. explicitly.

LEMMA F.10 (Constraint-Free Typing).

$$\forall C, \Gamma, e, \tau. (C, \Gamma \vdash_R e : \tau) \iff \exists \phi. (\phi = \text{mgu}(C)) \wedge (\epsilon, \phi(\Gamma) \vdash_R \phi(e) : \phi(\tau))$$

Proof Firstly, from the Consistency Lemma we know that C must be consistent for the left-hand side to hold. From the definition of consistency, rule (CONSISTENT), we know that hence the substitution ϕ must exist.

For the well-typing derivation, any derivation step that does not rely on C still works. Only two rules rely on C , namely (R-CONS) and (R-EQ). The difference boils down to showing that $\forall D. C \models D \iff \phi(D)$. This is easy to see.

F.4 Proof of the Main Soundness Theorem

Proof The first part of overall soundness, i.e. the first antecedent of rule (R-MAIN)

$$\epsilon, \theta(\Gamma) \vdash_r e : \theta(\tau), P$$

, follows from the Deferring Soundness Lemma.

From the Deferred Soundness Lemma, we have also:

$$\forall \langle C_i, \Gamma_i, e_i, \tau_i \rangle \in P : C_i, \Gamma_i \vdash_r e_i, \tau_i$$

From the Monotonicity Lemma of \vdash_r we know that any instance of θ also yields a deferring well-typing. Because of the Deferring Principality Lemma, we know that all other deferring well-typings are instances.

It is important to not here that the substitution produced by solving the proper implication constraints $\text{pimpl}(F)$ does not bind any unification variables in τ_i . These unification variables are recorded in the $[\bar{\alpha}]$ part of the implications, and the condition in rule (S-PIMPL) prevents them from being instantiated.

Hence, the full second antecedent holds:

$$\forall \tau', P'. (C, \Gamma \vdash_r e : \tau', P') \implies \forall \langle C_i, \Gamma_i, e_i, \tau_i \rangle \in P'. C_i, \Gamma_i \vdash e_i : \tau_i$$

We may conclude:

$$\epsilon, \theta(\Gamma) \vdash_r e : \tau$$

F.5 Proof of the Main Principality Theorem

Proof Overall principality coincides with deferred principality. Hence, it follows from the Deferred Principality Lemma.

G. Proof of Completeness

G.1 The Deferred and Deferring Completeness Theorems

LEMMA G.1 (Deferring Completeness). *If an expression e has a deferring typing, then the simple generated constraints and constraint solving yield a solution.*

$$\implies \exists F, \tau'. (\Gamma \vdash_W e : \tau', F) \wedge \exists \theta. (\vdash_s F : \theta)$$

Proof We construct a proof by contradiction. Assume that $\epsilon, \Gamma \vdash_r e : \tau, P$ holds, and $\Gamma \vdash_W e : \tau', F$. However, $\vdash_s \text{simple}(F) : \theta$ does not hold.

However, we can show that: there exists a substitution ϕ such that $\tau = \phi(\tau')$, and $\models \phi(\text{simple}(F))$. Because of the completeness of \vdash_s it is thus impossible that $\vdash_s \text{simple}(F) : \theta$ does not hold.

LEMMA G.2 (Deferred Completeness).

$$\begin{aligned} & \forall \Gamma, e, \tau, F, \theta_s, P, \langle C_i, \Gamma_i, e_i, \tau_i \rangle \in P : \\ & (\Gamma \vdash_W e : \tau, F) \\ & \wedge (\vdash_s \text{simple}(F) : \theta_s) \\ & \wedge (\epsilon, \theta_s(\Gamma) \vdash_r e : \theta_s(\tau), P) \\ & \wedge (C_i, \Gamma_i \vdash_r e_i : \tau_i) \\ \implies & \exists \theta_p : (\vdash_s \theta_s(\text{pimpl}(F)) : \theta_p) \end{aligned}$$

Proof The proof is similar in style.

LEMMA G.3 (Partitioning).

$$\forall F, \theta_s, \theta_p : \vdash_s \text{simple}(F) : \theta_s \wedge \vdash_s \theta_s(\text{pimpl}(F)) : \theta_p \implies \vdash_s F : \theta_p \cdot \theta_s$$

Proof The proof proceeds by induction on F . The base cases are:

- ϵ : We have that $\text{simple}(F) = \text{pimpl}(F) = \epsilon$ and $\theta_s = \theta_p = \theta_p \cdot \theta_s = \emptyset$. Indeed, $\vdash \epsilon : \emptyset$.
- $\tau_1 \sim \tau_2$: We have that $\text{simple}(F) = F$ and $\text{pimpl}(F) = \epsilon$. Thus $\theta_p = \emptyset$ and $\theta_p \cdot \theta_s = \theta_s$. Indeed, $\vdash_s \tau_1 \sim \tau_2 : \theta_s$.

The inductive cases are:

- $F_1 \wedge F_2$: The induction assumption tells us that $\vdash_s F_1 : \theta_{p,1} \cdot \theta_{s,1}$ and $\vdash_s F_2 : \theta_{p,2} \cdot \theta_{s,2}$. Moreover, $\vdash_s \text{simple}(F_1) \wedge \text{simple}(F_2) : \theta_s$. There is only two possibilities for the latter to be true: (S-SPLIT). Also, $\vdash_s \text{pimpl}(F_1) \wedge \text{pimpl}(F_2) : \theta_p$. There is only one possibility for this to be true, by means of rule (S-SPLIT). Hence, w.o.l.g., we have that $\theta_p = \theta_{p,2} \cdot \theta_{p,1}$. From these, it follows that $\vdash_s F : \theta_p \cdot \theta_s$.
- $\forall \bar{b}. F'$: We have that $\text{simple}(F) = \forall \bar{b}. \text{simple}(F')$ and $\text{pimpl}(F) = \text{pimpl}(F')$. Our induction assumption tells us that $\vdash_s F' : \theta_p \cdot \theta_s$. By means of rule (S-SIMPL) we may conclude that $\vdash_s \forall \bar{b}. F' : \theta_p \cdot \theta_s$.
- $[\bar{\alpha}](\forall \bar{b}. C \supset F')$: We have that $\text{simple}(F) = \epsilon$ and $\text{pimpl}(F) = F$. Thus $\theta_s = \emptyset$ and $\theta_p \cdot \theta_s = \theta_p$. Hence indeed $\vdash_s [\bar{\alpha}](\forall \bar{b}. C \supset F') : \theta_p$.

G.2 Proof of the Main Completeness Theorem

Proof Given $\epsilon, \Gamma \vdash_R e : \tau$, we know by rule (R-MAIN) that there exists a P such that $C, \Gamma \vdash_r e : \tau, P$. Hence, by the Deferring Completeness Lemma, we know that there exists an F, τ' and θ_s such that

$$\Gamma \vdash_W e : \tau', F \wedge \vdash_s \text{simple}(F) : \theta_s$$

Similarly, it follows from the Deferred Completeness Lemma that there exists a θ_p such that

$$\vdash_s \theta_s(\text{pimpl}(F)) : \theta_p$$

By the Partitioning Lemma we conclude:

$$\vdash_s F : \theta_p \cdot \theta_s$$