# *Proof-relevant Horn Clauses for Dependent Type Inference and Term Synthesis*

František Farka
*University of St Andrews, and Heriot-Watt University*
(*e-mail:* `ff12@st-andrews.ac.uk`)

Ekaterina Komendantskaya
*Heriot-Watt University*
(*e-mail:* `ek19@hw.ac.uk`)

Kevin Hammond
*University of St Andrews*
(*e-mail:* `kevin@kevinhammond.net`)

*submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003*

## Abstract

First-order resolution has been used for type inference for many years, including in Hindley-Milner type inference, type-classes, and constrained data types. Dependent types are a new trend in functional languages. In this paper, we show that proof-relevant first-order resolution can play an important role in automating type inference and term synthesis for dependently typed languages. We propose a calculus that translates type inference and term synthesis problems in a dependently typed language to a logic program and a goal in the proof-relevant first-order Horn clause logic. The computed answer substitution and proof term then provide a solution to the given type inference and term synthesis problem. We prove the decidability and soundness of our method.

*KEYWORDS*: Proof-relevant logic, Horn clauses, Dependent types, Type Inference, Proof-relevant resolution

## 1 Introduction

First-order resolution is well known for supporting a range of automated reasoning methods for type inference. Simple types have been a part of mainstream languages since the 1960s; polymorphic types have been available in advanced languages such as ML and Haskell since the 1980s; and type classes were introduced from the 1990s onwards. Logic programming has had a role to play in each of these stages. Hindley and Milner (1978) were the first to notice that type inference in simply typed lambda calculus can be expressed as a first-order unification problem. For example, the rule for term application in this calculus

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \textsc{App}$$

gives rise to a type inference problem encoded by the following Horn clause:

$$\mathit{type}(\Gamma, \mathrm{app}(M, N), B) \leftarrow \mathit{type}(\Gamma, M, A \to B) \wedge \mathit{type}(\Gamma, N, A)$$

Given a term $E$, the query $\mathit{type}(\Gamma, E, T)$ infers a type $T$ in a context $\Gamma$ such that the typing judgement $\Gamma \vdash E : T$ holds. This general scheme allows a multitude of extensions. For example, in $HM(X)$ type inference (Odersky et al. 1999), constrained types must be accounted for. For this extension, a constraint logic programming $CLP(X)$ (Sulzmann and Stuckey 2008), was suggested, in which constraint solving over a domain $X$ was added to the existing first-order unification and resolution algorithms. Haskell type classes are another example of the application of Horn clause resolution. Consider the following instances of the Haskell equality type class:

```
instance Eq Int where ...
instance(Eq x,Eq y) ⇒ Eq(x,y) where ...
```

They can be encoded by the following Horn clauses, annotated with names $\kappa_{\mathtt{Int}}$ and $\kappa_{\mathtt{Pair}}$:

$$\begin{aligned}\kappa_{\mathtt{Int}} &: eq(int)\\ \kappa_{\mathtt{Pair}} &: eq(X,Y) \leftarrow eq(X) \wedge eq(Y)\end{aligned}$$

Type class instance resolution is then implemented as first-order resolution on Horn clauses. There is only one caveat—a *dictionary* (that is, a proof term) needs to be constructed (Peyton Jones et al. 1997). For example, $Eq\ (Int, Int)$ is inferred to have a dictionary $\kappa_{\mathtt{Pair}}(\kappa_{\mathtt{Int}}, \kappa_{\mathtt{Int}})$. This records the resolution trace and is treated as a witness of the type class instance $Eq\ (Int, Int)$. Horn clause resolution is thus extended to *proof-relevant resolution* (Fu et al. 2016). This line of work is on-going: various extensions to the syntax of type classes are still being investigated (Karachalias and Schrijvers 2017). In recent years, the idea of relational type inference has been taken further by miniKanren (Hemann et al. 2016). This offers a range of relational domain specific languages for ML, Rust, Haskell and many other languages. As Ahn and Vezzosi (2016) point out, a relational language can be very convenient in encodings of type inference problems.

In the last decade, dependent types (Weirich et al. 2017; Brady 2013) have gained popularity in the programming language community. They allow reasoning about program values within the types, and thus give more general, powerful and flexible mechanisms to enable verification of both the functional (correctness, compliance, etc.) and the non-functional (execution time, space, energy usage etc.) properties of code. Automation of type inference represents a big challenge for these languages. Most dependently typed languages, such as Coq, Agda or Idris, incorporate a range of algorithms that automate various aspects of type inference (*cf.* Pientka (2013)). Some use reflection (Slama and Brady 2017), some are based on algorithms that are similar to first-order resolution (Gonthier and Mahboubi 2010), and others (*e.g.* Liquid Haskell) incorporate third-party SMT solvers (Vazou et al. 2018). However, to the best of our knowledge, logic programming has not yet made its definitive mark in this domain.

This paper fills this gap: we propose a first systematic approach to logic-programming based type inference for a dependently-typed language. We demonstrate that Horn clause logic provides a convenient formal language to express type inference problems while staying very close to the formal specification of the dependently-typed language. Proof-relevant resolution then computes proof terms that capture well-formedness derivations of objects in the language. We present a method to synthesise terms of the dependently typed language from such proof terms. This method can be applied in a more complex setting where a small kernel of a verified compiler off-loads proof-relevant resolution to an external, non-verified resolution engine and then verifies synthesised derivations internally.

In the next section, we explain our main idea by means of an example.

## *Overview of Results by Means of an Example*

We rely on an abstract syntax that closely resembles existing functional programming languages with dependent types. We will call it the *surface language*. Using the syntax we define $\mathtt{maybe}_{\mathtt{A}}$, an option type over a fixed type $\mathtt{A}$, indexed by a Boolean:

$$\begin{array}{ll}\textbf{data}\ \mathtt{maybe}_{\mathtt{A}}\ (a : \mathtt{A}) & : \mathtt{bool} \to \mathtt{type}\ \ \textbf{where}\\ \quad \mathtt{nothing} & : \mathtt{maybe}_{\mathtt{A}}\ \mathtt{ff}\\ \quad \mathtt{just} & : \mathtt{A} \to \mathtt{maybe}_{\mathtt{A}}\ \mathtt{tt}\end{array}$$

Here, `nothing` and `just` are the two *constructors* of the `maybe` type. The type is indexed by `ff` when the `nothing` constructor is used, and by `tt` when the `just` constructor is used (`ff` and `tt` are constructors of `bool`). A function `fromJust` extracts the value from the `just` constructor:

$$\begin{aligned}&\texttt{fromJust} \;:\; \texttt{maybe}_{\texttt{A}}\;\texttt{tt} \to \texttt{A}\\ &\texttt{fromJust}\;(\texttt{just}\;x) = x\end{aligned}$$

Note that the value `tt` appears within the type $\texttt{maybe}_{\texttt{A}}\;\texttt{tt} \to \texttt{A}$ of this function (the type *depends* on the value), allowing for a more precise function definition that omits the redundant case when the constructor of type $\texttt{maybe}_{\texttt{A}}$ is `nothing`. The challenge for the type checker is to determine that the missing case `fromJust nothing` is contradictory (rather than being omitted by mistake). Indeed, the type of `nothing` is $\texttt{maybe}_{\texttt{A}}\;\texttt{ff}$. However, the function specifies its argument to be of type $\texttt{maybe}_{\texttt{A}}\;\texttt{tt}$.

To type check such functions, the compiler translates them into terms in a type-theoretic calculus. In this paper, we will rely on the calculus LF (Harper and Pfenning 2005), a standard and well-understood first-order dependent type theory. We call this calculus the *internal language* of the compiler. For our example, the signature of the internal language is as follows:

```
A        : type                          maybe_A     : bool → type
bool     : type                          nothing     : maybe_A ff
ff tt    : bool                          just        : A → maybe_A tt
                                         elim_maybe_A : Π(b:bool).maybe_A b
(≡_bool) : bool → bool → type                        → (b ≡_bool ff → A)
refl     : Π(b:bool).b ≡_bool b                      → (b ≡_bool tt → A → A)
elim_≡bool: tt ≡_bool ff → A                         → A
```

We use $A \to B$ as an abbreviation for $\Pi(a : A).B$ where $a$ does not occur free in $B$. The final goal of type checking of the function `fromJust` in the surface language is to obtain the following encoding in the internal language:

$$\begin{aligned}\texttt{t}_{\texttt{fromJust}} \;:=\; &\lambda\,(m{:}\texttt{maybe}_{\texttt{A}}\;\texttt{tt}).\texttt{elim}_{\texttt{maybe}_{\texttt{A}}}\;\texttt{tt}\;m\\ &(\lambda\;(w{:}\texttt{tt}\equiv_{\texttt{bool}}\texttt{ff}).\texttt{elim}_{\equiv_{\texttt{bool}}}\;w)\\ &(\lambda\;(w{:}\texttt{tt}\equiv_{\texttt{bool}}\texttt{tt}).\lambda\;(x{:}\texttt{A}).x)\end{aligned}$$

The missing case for `nothing` must be accounted for (*cf.* the line $(\lambda\,(w{:}\texttt{tt}\equiv_{\texttt{bool}}\texttt{ff}).\texttt{elim}_{\equiv_{\texttt{bool}}} w)$ above). In this example (as is generally the case), only partial information is given in the surface language. To address this problem, we extend the internal language with term level metavariables, denoted by $?_a$, and type level metavariables, denoted by $?_A$. These stand for the parts of a term in the internal language that are not yet known. Using metavariables, the term that directly corresponds to `fromJust` is:

$$\begin{aligned}\texttt{t}_{\texttt{fromJust}} \;:=\; &\lambda\,(m{:}\texttt{maybe}_{\texttt{A}}\;\texttt{tt}).\texttt{elim}_{\texttt{maybe}_{\texttt{A}}}\;?_a\;m\\ &(\lambda\;(w{:}\;?_A\;).\;?_b\;)\\ &(\lambda\;(w{:}\;?_B\;).\lambda\;(x{:}\texttt{A}).x)\end{aligned} \tag{I}$$

The missing information comprises the two types $?_A$ and $?_B$ and the term $?_b$ for the constructor `nothing`. Obtaining types $?_A$, $?_B$ amounts to type inference (in the internal language, as opposed to checking in the surface language), whereas obtaining the term $?_b$ amounts to term synthesis. In this paper, we are interested in automating such reasoning. We use the notion *refinement* to refer to the combined problem of type inference and term synthesis. We make use of *proof-relevant* Horn clause logic (Fu and Komendantskaya 2017) that was initially inspired by type class resolution, as described above. In this logic, Horn clauses are seen as types and proof witnesses — as terms inhabiting the types. Given a proposition—a *goal*—and a set of Horn clauses—a *logic program*—the resolution process is captured by an explicit proof term construction. We translate *refinement problems* into the syntax of logic programs. The *refinement* algorithm that we propose takes a signature and a term with metavariables in the extended internal language to a logic program and a goal in proof-relevant Horn clause logic. The unifiers that are computed by resolution give an assignment of types to type-level metavariables. At the same time, the computed proof terms are interpreted as an assignment of terms to term-level metavariables. We illustrate the process in the following paragraphs.

Consider the inference rule Π-T-ELIM in LF, which generalises the inference rule APP given on page 1 of this paper:

$$\frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}\ \Pi\text{-T-ELIM}$$

When type checking the term $t_{\texttt{fromJust}}$ (defined in (I)) an application of $\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m$ to the term $\lambda(w :?_A).?_b$ in the context $m : \texttt{maybe}_\texttt{A}\ \texttt{tt}$ needs to be type checked. This amounts to providing a derivation of the typing judgement that contains the following instance of the rule Π-T-ELIM:

$$\frac{\begin{array}{l} m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash \texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m \\ \qquad : (\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \to \texttt{A}) \to \cdots \to \texttt{A} \end{array} \qquad m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash \lambda(w :?_A).?_b :?_A \to ?_B}{m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash (\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m)\ (\lambda(w :?_A).?_b) : (\texttt{tt} \equiv_{\texttt{bool}} \texttt{tt} \to \texttt{A} \to \texttt{A}) \to \texttt{A}}$$

For the above inference step to be a valid instance of the inference rule Π-T-ELIM, it is necessary that $(\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) =\ ?_A$ and $\texttt{A} =\ ?_B$. This is reflected in the goal:

$$((\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) =\ ?_A) \wedge (\texttt{A} =\ ?_B) \wedge G_{(\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m)} \wedge G_{\lambda(w:?_A).?_b)} \qquad \text{(II)}$$

The additional goals $G_{(\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m)}$ and $G_{\lambda(w:?_A).?_b}$ are recursively generated by the algorithm for the terms $\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m$ and $\lambda(w :?_A).?_b$, respectively. Similarly, assuming the term $\lambda(w :?_A).?_b$ is of type $(\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) \to \texttt{A}$, type checking places restrictions on the term $?_b$:

$$\frac{m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} : \texttt{type} \qquad m : \texttt{maybe}_\texttt{A}\ \texttt{tt}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \vdash ?_b : \texttt{A}}{m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash \lambda(w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}).?_b : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \to \texttt{A}}$$

That is, $?_b$ needs to be a well-typed term of type $\texttt{A}$ in a context consisting of $m$ and $w$. Recall that in the signature there is a constant $\texttt{elim}_{\equiv_{\texttt{bool}}}$ of type $\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \to \texttt{A}$. Our translation will turn this constant into a clause in the generated logic program. There will be a clause that corresponds to the inference rule for elimination of a Π type as well:

$$\begin{aligned} \kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}} &: \mathit{term}(\texttt{elim}_{\equiv_{\texttt{bool}}}, \Pi x : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \,.\, \texttt{A}, ?_\Gamma) \leftarrow \\ \kappa_{\texttt{elim}} &: \mathit{term}(?_M ?_N, ?_B, ?_\Gamma) \leftarrow \mathit{term}(?_M, \Pi x :?_A.?_{B'}, ?_\Gamma) \wedge \mathit{term}(?_N, ?_A, ?_\Gamma) \wedge ?_{B'}[?_N/x] \equiv ?_B \end{aligned}$$

The above clauses are written in the proof-relevant Horn clause logic, and thus $\kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}}$ and $\kappa_{\texttt{elim}}$ now play the role of proof-term symbols ("witnesses" for the clauses). In this clause, $?_M$, $?_N$, $?_A$, $?_B$, $?_{B'}$ and $?_\Gamma$ are *logic variables*, i.e. variables of the first-order logic. By an abuse of notation, we use the same symbols for metavariables of the internal language and logic variables in the logic programs generated by the refinement algorithm. We also use the same notation for objects of the internal language and terms of the logic programs. This is possible since we represent variables using *de Bruijn indices*.

The presence of $w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}$ in the context allows us to use the clause $\texttt{elim}_{\equiv_{\texttt{bool}}}$ to resolve the goal $\mathit{term}(?_M ?_N, \texttt{A}, [m : \texttt{maybe}_\texttt{A}\ \texttt{tt}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}])$:

$$\begin{array}{l} \mathit{term}(?_M ?_N, \texttt{A}, [m : \texttt{maybe}_\texttt{A}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}]) \rightsquigarrow_{\kappa_{\texttt{elim}}} \\ \quad \mathit{term}(?_M, \Pi x :?_A .\, \texttt{A}, [\ldots]) \wedge \mathit{term}(?_N, ?_A, [\ldots, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}]) \wedge \texttt{A}[?_N/x] \equiv ?_B \rightsquigarrow_{\kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}}} \\ \quad \mathit{term}(?_N, \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}, [\ldots, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}]) \wedge \texttt{A}[?_N/x] \equiv ?_B \rightsquigarrow_{\kappa_{\texttt{proj}_w}} \\ \quad \texttt{A}[?_N/x] \equiv ?_B \rightsquigarrow_{\kappa_{\texttt{subst}_\texttt{A}}} \bot \end{array} \qquad \text{(III)}$$

The resolution steps are denoted by $\rightsquigarrow$. Each step is indexed by the name of the clause that was used. First, the goal is resolved in one step using the clause $\kappa_{\texttt{elim}}$. A clause $\kappa_{\texttt{proj}_w}$ is used to project the variable $w$ from the context. We postpone further discussion of the exact shape of the clauses until Section 3, since it depends on the representation we use. For the moment, we are just interested in composing the proof terms occurring in these resolution steps into one composite proof term: $\kappa_{\texttt{elim}}\ \kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}}\ \kappa_{\texttt{proj}_w}\kappa_{\texttt{subst}_{\texttt{A}}}$. Note that, by resolving the goal (II), we obtain a substitution $\theta$ that assigns the type $A$ to the logic variable $?_B$, *i.e.* $\theta(?_B) = A$. At the same time, the proof term computed by the the derivation (III) is interpreted as a solution ($\texttt{elim}_{\equiv_{\texttt{bool}}}\ w$) for the term-level metavariable $?_b$. However, the proof term can be used to reconstruct the derivation of well-typedness of the judgement $m : \texttt{maybe}_{\texttt{A}}\ \ \texttt{tt}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \vdash \texttt{elim}_{\equiv_{\texttt{bool}}}\ w : \texttt{A}$ as well. In general, a substitution is interpreted as a solution to a type-level metavariable and a proof term as a solution to a term-level metavariable. The remaining solution for $?_A$ is computed using similar methodology, and we omit the details here. Thus, we have computed values for all metavariables in function (I), *i.e.* we inferred all types and synthesised all terms.

## *Contributions*

Our main contributions are:

1. We present a novel approach to refinement for a first-order type theory with dependent types that is simpler than existing methods, *e.g.* Pientka and Dunfield (2010).
2. We prove that generation of goals and logic programs from the extended language is decidable.
3. We show that proof-relevant first-order Horn clause resolution gives an appropriate inference mechanism for dependently typed languages: firstly, it is sound with respect to type checking in LF; secondly, the proof term construction alongside the resolution trace allows to reconstruct the derivation of well-typedness judgement.

This paper is structured as follows. Section 2 gives a nameless formulation of LF, the chosen first-order dependent type theory. We then present the described refinement algorithm by means of a formal calculus in Section 3 and show that it is decidable. Section 4 establishes interpretation of answer substitutions and proof terms as solutions to refinement problems and states soundness of the interpretation. Finally, in Sections 5 and 6 we discuss related and future work and conclude.

## 2 Nameless LF

Standard expositions of a type theory use variable names. However, variable names carry a burden when implementing such a type theory. For example, types need to be checked up to $\alpha$-equivalence of bound variables and fresh names need to be introduced in order to expand terms to $\eta$-long form. In order to avoid the burden, existing implementations use de Bruijn indices. We use de Bruijn indices directly in our exposition as it allows us to avoid the above problems when checking the equality of terms and types and when synthesising new terms and types.

We use natural numbers in $\mathbb{N}$ for de Bruijn indices $\iota, \iota_1, \ldots$, and we denote successor by $\sigma(-)$. We assume countably infinite disjoint sets $\mathcal{C}$ of *term constants*, and $\mathcal{B}$ of *type constants*. We denote elements of $\mathcal{C}$ by $c$, $c'$, *etc.*, and elements of $\mathcal{B}$ by $\alpha$, $\beta$, *etc.*

*Definition 1* (*Nameless LF*)
The *terms*, *types*, and *kinds* as well as *signatures* and *contexts* are:

$$\begin{array}{lrl@{\qquad}lrl}
\text{Kinds} & K ::= & \text{type} \mid \Pi T.K & \text{Signatures} & \mathit{Sig} ::= & \cdot \mid \mathit{Sig}, \mathcal{C} : T \mid \mathit{Sig}, \mathcal{B} : K \\
\text{Types} & T ::= & \mathcal{B} \mid Tt \mid \Pi T.T & \text{Contexts} & \mathit{Con} ::= & \cdot \mid \mathit{Con}, T \\
\text{Terms} & t ::= & \mathcal{C} \mid \mathbb{N} \mid \lambda T.t \mid tt & & &
\end{array}$$

We use identifier $L$ to denote kinds in $K$, identifiers $A$, $B$ to denote types in $T$ and identifiers $M$, $N$ to denote terms in $t$. We use $\mathcal{S}$ for signatures and $\Gamma$ for contexts. In line with standard practice, we define two operations. *Shifting* recursively traverses a term, a type, or a kind and increases all indices by one.

*Definition 2* (*Shifting*)
Term and type shifting, denoted by $(-)\uparrow^{\iota}$ is defined as follows:

$$\begin{array}{ccc} c\uparrow^{\iota} = c & \iota\uparrow^{0} = \sigma\iota & \alpha\uparrow^{\iota} = \alpha \\ (\lambda A.M)\uparrow^{\iota} = \lambda A\uparrow^{\iota}.M\uparrow^{\sigma\iota} & 0\uparrow^{\sigma\iota} = 0 & (\Pi A.B)\uparrow^{\iota} = \lambda A\uparrow^{\iota}.B\uparrow^{\sigma\iota} \\ (MN)\uparrow^{\iota} = M\uparrow^{\iota}.N\uparrow^{\iota} & \sigma\iota\uparrow^{\sigma\iota'} = \sigma(\iota\uparrow^{\iota'}) & (AM)\uparrow^{\iota} = A\uparrow^{\iota}.M\uparrow^{\iota} \end{array}$$

*Substitution* with a term $N$ and index $\iota$ replaces indices that are bound by the $\iota$-th binder while updating remaining indices. The index $\iota$ is increased when traversing under a binder.

*Definition 3* (*Substitution*)
Term and type substitution, denoted by $(-)[N/\iota]$ is defined as follows:

$$\begin{array}{ccc} c[N/\iota] = c & 0[N/0] = N & \alpha[N/\iota] = \alpha \\ (\lambda A.M)[N/\iota] = \lambda A[N/\iota].M[N\uparrow^{0}/\sigma\iota] & 0[N/\sigma\iota] = 0 & (\Pi A.B)[N/\iota] = \lambda A[N/\iota].B\uparrow^{0}[N/\sigma\iota] \\ (M_1M_2)[N/\iota] = M_1[N/\iota].M_2[N/\iota] & \sigma\iota[N/0] = \sigma\iota & (AM)[N/\iota] = A[N/\iota].M[N/\iota] \\ & \sigma\iota[N/\sigma\iota'] = \sigma(\iota[N/\iota']) & \end{array}$$

Shifting with a greater index than 0 and substitution for other indices than 0 is not necessary in the inference rules of neither the internal language nor refinement. For the sake of readability we introduce the following abbreviations:

$$A\uparrow \overset{def}{=} A\uparrow^{0} \qquad M\uparrow \overset{def}{=} M\uparrow^{0} \qquad A[N] \overset{def}{=} A[N/0] \qquad M[N] \overset{def}{=} M[N/0]$$

Well-formedness of objects introduced by Definition 1 is stated by means of several judgements. In particular, we give equality in nameless LF as algorithmic, following Harper and Pfenning (Harper and Pfenning 2005). In order to do so we define simple kinds, simple types, simple signatures, and simple contexts:

*Definition 4*
The *simple kinds*, *simple types*, and *simple signatures* are:

$$\begin{array}{rrcl} \text{Simple kinds} & K^{-} & ::= & \mathtt{type}^{-} \mid T^{-} \to K^{-} \\ \text{Simple types} & T^{-} & ::= & \mathcal{B} \mid T^{-} \to T^{-} \\ \text{Simple signatures} & Sig^{-} & ::= & \cdot \mid Sig^{-}, \mathcal{C} : T^{-} \mid Sig^{-}, \mathcal{B} : K^{-} \\ \text{Simple contexts} & Con^{-} & ::= & \cdot \mid Con^{-}, T^{-} \end{array}$$

We use identifiers $\kappa$ for simple kinds, $\tau$ for simple types, $\mathcal{S}^{-}$ for simple signatures and $\Delta$ for simple contexts. The erasure from objects to corresponding simple objects, denoted $(-)^{-}$ is defined as follows:

*Definition 5* (*Erasure*)

$$\begin{array}{cc} (\mathtt{type})^{-} = \mathtt{type} & (\alpha)^{-} = \alpha \\ (\Pi A.L)^{-} = (A)^{-} \to (L)^{-} & (\Pi A.B)^{-} = (A)^{-} \to (B)^{-} \\ & (AM)^{-} = (A)^{-} \end{array}$$

The well-formedness of judgements for kinds, types and terms, weak algorithmic equality of types, algorithmic and structural equality of terms, and weak head reduction of terms are:

$$\begin{array}{cccc} \mathcal{S};\Gamma \vdash L : \mathtt{kind} & \mathcal{S};\Gamma \vdash A : L & \mathcal{S};\Gamma \vdash M : A & \\ \mathcal{S}^{-};\Delta \vdash A \rightleftharpoons A' : \kappa & \mathcal{S};\Gamma \vdash M \leftrightarrow M' : \tau & \mathcal{S};\Gamma \vdash M \Leftrightarrow M' : \tau & M \xrightarrow{\text{whr}} M' \end{array}$$

The inference rules for well-formedness of kinds, types, and terms are listed in Figures 1, 2,

$\boxed{\mathcal{S};\Gamma\vdash L:\texttt{kind}}$

$$\frac{\mathcal{S}\vdash\Gamma\ \mathrm{ctx}}{\mathcal{S};\Gamma\vdash\texttt{type}:\texttt{kind}}\ \textrm{K-TY}\qquad\frac{\mathcal{S};\Gamma\vdash A:\texttt{type}\qquad\mathcal{S};\Gamma,A\vdash L:\texttt{kind}}{\mathcal{S};\Gamma\vdash\Pi A.L:\texttt{kind}}\ \textrm{K-}\Pi\textrm{-INTRO}$$

Fig. 1. Well-formedness of nameless kinds

$\boxed{\mathcal{S};\Gamma\vdash A:L}$

$$\frac{\mathcal{S}\vdash\Gamma\ \mathrm{ctx}\qquad\alpha:L\in\mathcal{S}}{\mathcal{S};\Gamma\vdash\alpha:L}\ \textrm{T-CON}\qquad\frac{\mathcal{S};\Gamma\vdash A:\texttt{type}\qquad\mathcal{S};\Gamma,A\vdash B:\texttt{type}}{\mathcal{S};\Gamma\vdash\Pi A.B:\texttt{type}}\ \textrm{T-}\Pi\textrm{-INTRO}$$

$$\frac{\mathcal{S};\Gamma\vdash A:\Pi B.L\qquad\mathcal{S};\Gamma\vdash M:B'\qquad\mathcal{S}^-;\Gamma^-\vdash B\rightleftharpoons B':\texttt{type}^-}{\mathcal{S};\Gamma\vdash AM:L[M]}\ \textrm{T-}\Pi\textrm{-ELIM}$$

Fig. 2. Well-formedness of nameless types

and 3. The inference rules of well-formedness judgements for signatures and contexts as well as for definitional equality are standard (*cf.* Harper and Pfenning (2005)). The inference rules for weak head reduction are listed in Figure 4. The well-formedness of signatures and contexts is defined in Figure 5. Algorithmic equality of terms, structural equality of terms and weak algorithmic equality of types are defined in Figures 6, 7, and 8 respectively.

## 3 Refinement in Nameless LF

Following the ideas we sketched in Section 1, we present the translation of a refinement problem into Horn clause logic with explicit proof terms. Firstly, we extend the language of nameless LF with metavariables, which allows us to capture incomplete terms. Secondly, we describe the language of Horn clause logic with explicit proof terms. Finally, we give a calculus for transformation of an incomplete term to a goal and a program.

### *3.1 Refinement Problem*

We capture missing information in nameless LF terms by metavariables. We assume infinitely countable disjoint sets $?_{\mathcal{B}}$ and $?_{\mathcal{V}}$ that stand for omitted types and terms and we call elements of these sets type-level and term-level metavariables respectively. We use identifiers $?_a$, $?_b$, *etc.* to denote elements of $?_{\mathcal{V}}$ and identifiers $?_A$, $?_B$, *etc.* to denote elements of $?_{\mathcal{B}}$. The extended syntax is defined as follows:

*Definition 6* (*Extended Nameless LF*)
We define *extended nameless types, terms and contexts* as follows:

$$\begin{array}{lll} \text{Types} & T ::= & \cdots \mid ?_{\mathcal{B}} \\ \text{Terms} & t ::= & \cdots \mid ?_{\mathcal{V}} \end{array}\qquad \text{Contexts}\quad Con ::= \quad \cdots \mid Con, ?_{\mathcal{V}} : T$$

The ellipsis in the definition are to be understood as the appropriate syntactic constructs of Definition 1. Note that we do not define an extended signature. We assume that the signature is always fixed and does not contain any metavariables. This does not pose any problem since well-typedness of signature does not depend on the term being refined. We use mtvar(−) and mvar(−) to denote the sets of type-level and term-level metavariables respectively. The well-formedness judgements of the nameless LF are then defined on a subset of extended objects.

*Lemma 1*
Let $L$ be an extended nameless kind, $A$ an extended nameless type and $M$ an extended nameless term. Let $\mathcal{S}$ and $\Gamma$ be contexts.

$\boxed{\mathcal{S};\Gamma \vdash M : A}$

$$\frac{\mathcal{S} \vdash \Gamma \text{ ctx} \qquad c : A \in \mathcal{S}}{\mathcal{S};\Gamma \vdash c : A}\ \textsc{con} \qquad \frac{\mathcal{S} \vdash \Gamma, A \text{ ctx}}{\mathcal{S};\Gamma, A \vdash 0 : A\!\uparrow}\ \textsc{zero} \qquad \frac{\mathcal{S};\Gamma \vdash \iota : A}{\mathcal{S};\Gamma, B \vdash \sigma\iota : A\!\uparrow}\ \textsc{succ}$$

$$\frac{\mathcal{S};\Gamma \vdash A : \texttt{type} \qquad \mathcal{S};\Gamma, A \vdash M : B}{\mathcal{S};\Gamma \vdash \lambda A.M : \Pi A.B}\ \Pi\text{-}\textsc{intro}$$

$$\frac{\mathcal{S};\Gamma \vdash M : \Pi A.B \qquad \mathcal{S};\Gamma \vdash N : A' \qquad \mathcal{S}^{-};\Gamma^{-} \vdash A \rightleftharpoons A' : \texttt{type}}{\mathcal{S};\Gamma \vdash MN : B[N]}\ \Pi\text{-}\textsc{elim}$$

Fig. 3. Well-formedness of nameless terms

$\boxed{M \xrightarrow{\text{whr}} M'}$

$$\frac{}{(\lambda A.M)N \xrightarrow{\text{whr}} M[N]} \qquad \frac{M \xrightarrow{\text{whr}} M'}{MN \xrightarrow{\text{whr}} M'N'}$$

Fig. 4. Weak head reduction of terms

- If $\mathcal{S};\Gamma \vdash L : \texttt{kind}$ then $\text{mvar}(L) = \emptyset$ and $\text{mtvar}(L) = \emptyset$,
- if $\mathcal{S};\Gamma \vdash A : L$ then $\text{mvar}(A) = \emptyset$ and $\text{mtvar}(A) = \emptyset$, and
- if $\mathcal{S};\Gamma \vdash M : A$ then $\text{mvar}(M) = \emptyset$ and $\text{mtvar}(M) = \emptyset$.

*Proof*
By induction on the derivation of judgements. □

A *refinement problem* is defined as a term in the extended syntax. A signature and a context of the term are kept implicit.

*Example 1* (*Refinement Problem*)
Taking the problem from Introduction, the term $M'$ given by $(\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ 0)(\lambda ?_A.?_b)$ is a refinement problem. The appropriate context is $\Gamma_1 = \cdot, \texttt{maybe}_\texttt{A}\ \texttt{tt}$. This signature in Introduction is adjusted to nameless signature $\mathcal{S}$.

A *refinement* of a term is a pair of assignments $(\rho, R)$ such that $\rho : ?_\mathcal{V} \to t$ is an assignment of (extended) terms to term-level metavariables and $R : ?_\mathcal{B} \to T$ is an assignment of (extended) types to type-level metavariables. We define application of refinement $(\rho, R)(-)$ to terms, types and kinds by induction on definition of the syntactic object.

$\boxed{\vdash \mathcal{S} \text{ sig}}$

$$\frac{}{\vdash \cdot \text{ sig}} \qquad \frac{\vdash \mathcal{S} \text{ sig} \qquad \mathcal{S};\cdot \vdash L : \texttt{kind} \qquad a \notin \mathcal{S}}{\vdash \mathcal{S}, a : L \text{ sig}} \qquad \frac{\vdash \mathcal{S} \text{ sig} \qquad \mathcal{S};\cdot \vdash A : \texttt{type} \qquad c \notin \mathcal{S}}{\vdash \mathcal{S}, c : A \text{ sig}}$$

$\boxed{\mathcal{S} \vdash \Gamma \text{ ctx}}$

$$\frac{\vdash \mathcal{S} \text{ sig}}{\mathcal{S} \vdash \cdot \text{ ctx}} \qquad \frac{\mathcal{S} \vdash \Gamma \text{ ctx} \qquad \mathcal{S};\Gamma \vdash A : \texttt{type}}{\vdash \mathcal{S};\Gamma, A \text{ ctx}}$$

Fig. 5. Well-formedness of signatures and contexts

$\boxed{\mathcal{S}^-; \Delta \vdash M \Leftrightarrow M' : \tau}$

$$\frac{M \xrightarrow{\text{whr}} M' \qquad \mathcal{S}^-; \Delta \vdash M' \Leftrightarrow N}{\mathcal{S}^-; \Delta \vdash M \Leftrightarrow N} \qquad \frac{N \xrightarrow{\text{whr}} N' \qquad \mathcal{S}^-; \Delta \vdash M \Leftrightarrow N'}{\mathcal{S}^-; \Delta \vdash M \Leftrightarrow N}$$

$$\frac{\mathcal{S}^-; \Delta \vdash M \leftrightarrow N : \tau}{\mathcal{S}^-; \Delta \vdash M \Leftrightarrow N : \tau} \qquad \frac{\mathcal{S}^-; \Delta, \tau_1 \vdash (M\!\uparrow)0 \Leftrightarrow (N\!\uparrow)0 : \tau_2}{\mathcal{S}^-; \Delta \vdash M \Leftrightarrow N : \tau_1 \to \tau_2}$$

Fig. 6. Algorithmic equality of terms

$\boxed{\mathcal{S}^-; \Delta \vdash M \leftrightarrow N : \tau}$

$$\frac{\vdash \mathcal{S}^- \ \text{ssig}}{\mathcal{S}^-; \Delta, \tau \vdash 0 \leftrightarrow 0 : \tau} \qquad \frac{\mathcal{S}^-; \Delta \vdash \iota \leftrightarrow \iota' : \tau}{\mathcal{S}^-; \Delta, \tau' \vdash \sigma\iota \leftrightarrow \sigma\iota' : \tau} \qquad \frac{\vdash \mathcal{S}^- \ \text{ssig} \qquad c : \tau \in \mathcal{S}^-}{\mathcal{S}^-; \Delta \vdash c \leftrightarrow c : \tau}$$

$$\frac{\mathcal{S}^-; \Delta \vdash M_1 \leftrightarrow N_1 : \tau_2 \to \tau_1 \qquad \mathcal{S}^-; \Delta \vdash M_2 \Leftrightarrow N_2 : \tau_2}{\mathcal{S}^-; \Delta \vdash M_1 M_2 \leftrightarrow N_1 N_2 : \tau_1}$$

Fig. 7. Structural equality of terms

*Definition 7* (*Refinement application*)
Let $\rho : \ ?_{\mathcal{V}} \to t$ be an assignment of terms and $R : \ ?_{\mathcal{B}} \to T$ be an assignment of types. Application of the refinement $(\rho, R)$ to kinds, types and terms is defined by:

$$\begin{aligned}
(\rho, R)(\texttt{type}) &= \texttt{type} \\
(\rho, R)(\Pi A.L) &= \Pi(\rho, R)(A).(\rho, R)(L) \\
(\rho, R)(\alpha) &= \alpha \\
(\rho, R)(?_A) &= R(?_A) \\
(\rho, R)(\Pi A.B) &= \Pi(\rho, R)(A).(\rho, R)(B) \\
(\rho, R)(AN) &= (\rho, R)(A)(\rho, R)(N)
\end{aligned}
\qquad
\begin{aligned}
(\rho, R)(c) &= c \\
(\rho, R)(\iota) &= \iota \\
(\rho, R)(?_a) &= \rho(?_a) \\
(\rho, R)(\lambda x : A.M) &= \lambda x : (\rho, R)(A).(\rho, R)(M) \\
(\rho, R)(MN) &= (\rho, R)(M)(\rho, R)(N)
\end{aligned}$$

A *solution* to a refinement problem $t$ is a refinement $(\rho, R)$ such that $(\rho, R)(t)$ is a well-formed term of nameless LF. That is, by Lemma 1, $(\rho, R)(t)$ does not contain neither term- nor type-level metavariables.

Horn clause logic is usually presented using a signature that comprises sets of function and predicate symbols and the appropriate grammar for atomic and Horn formulae (*cf.* Miller and Nadathur (2012)). Although the same presentation can be given for Horn clause logic with explicit proof terms, for the sake of brevity we resort to a simpler presentation that is sufficient for our purposes. We define atomic formulae using objects of nameless LF and we list all the predicates that are needed for refinement translation explicitly in the grammar. However, we make an exception in the case of contexts and use the usual list notation. Metavariables of

$\boxed{\mathcal{S}^-; \Delta \vdash A \rightleftharpoons A' : \kappa}$

$$\frac{\vdash \mathcal{S}^- \ \text{ssig} \qquad \alpha : \kappa \in \mathcal{S}^-}{\mathcal{S}^-; \Delta \vdash \alpha \rightleftharpoons \alpha : \kappa} \qquad \frac{\mathcal{S}^-; \Delta \vdash A \rightleftharpoons B : \tau \to \kappa \qquad \mathcal{S}^-; \Delta \vdash M \Leftrightarrow N : \tau}{\mathcal{S}^-; \Delta \vdash AM \rightleftharpoons BN : \kappa}$$

$$\frac{\mathcal{S}^-; \Delta \vdash A_1 \rightleftharpoons B_1 : \texttt{type} \qquad \mathcal{S}^-; \Delta, (A_1)^- \vdash (A_2\!\uparrow) \rightleftharpoons (B_2\!\uparrow) : \texttt{type}}{\mathcal{S}^-; \Delta \vdash (\Pi A_1.A_2) \rightleftharpoons (\Pi B_1.B_2) : \texttt{type}}$$

Fig. 8. Weak algorithmic equality of types

$\boxed{\mathcal{S};\Gamma;M \vdash (G \mid A)}$

$$\frac{c : A \in \mathcal{S}}{\mathcal{S};\Gamma;c \vdash (\top \mid A)}\ \text{R-CON} \qquad \frac{}{\mathcal{S};\Gamma;?_a \vdash (?_a : \mathit{term}(?_{a'}, ?_A, \Gamma) \mid ?_A)}\ \text{R-T-META}$$

$$\frac{}{\mathcal{S};\Gamma,A;0 \vdash (A\!\uparrow\, \equiv\, ?_A \mid ?_A)}\ \text{R-ZERO} \qquad \frac{\mathcal{S};\Gamma;\iota \vdash (G \mid A)}{\mathcal{S};\Gamma,B;\sigma\iota \vdash (G \wedge (A\!\uparrow\, \equiv\, ?_A) \mid ?_A)}\ \text{R-SUCC}$$

$$\frac{\mathcal{S};\Gamma;A \vdash (G_A \mid L) \qquad \mathcal{S};\Gamma,A;M \vdash (G_M \mid B)}{\mathcal{S};\Gamma;\lambda A.M \vdash (G_A \wedge G_M \wedge \mathit{eq}_K(L, \mathtt{type}, \Gamma) \mid \Pi A.B)}\ \text{R-}\lambda\text{-INTRO}$$

$$\frac{\mathcal{S};\Gamma;M \vdash (G_M \mid A) \qquad \mathcal{S};\Gamma;N \vdash (G_N \mid A_2)}{\mathcal{S};\Gamma;MN \vdash (G_M \wedge G_N \wedge \mathit{eq}_T(A, \Pi A_2.?_B, \mathtt{type}, \Gamma) \wedge (?_B[N] \equiv ?_{B'}) \mid ?_{B'})}\ \text{R-}\lambda\text{-ELIM}$$

Fig. 9. Refinement of terms

extended nameless LF are seen as logic variables. Furthermore, we assume a finite set $\mathcal{K}$ of *proof-term symbols* and a countable set $\mathcal{D}$ of goal variables. We denote elements of $\mathcal{K}$ by $\kappa$ with indices and elements of $\mathcal{D}$ by $\gamma$ with indices. For technical reasons, we also use metavariables in positions of kinds, denoted $?_L$, and in position of indicies, denoted $?_\iota$. The syntax is defined as follows:

*Definition 8* (*Syntax of Horn Clause Logic with Explicit Proof Terms*)
*Atomic formulae*, *Horn formulae*, *programs*, *proof terms*, and *goals* are generated as follows:

$$\begin{array}{lll} \text{Atomic formulae} & At ::= & \mathit{eq}^a_t(t,t,T,\mathit{Con}) \mid \mathit{eq}^s_t(t,t,T,\mathit{Con}) \mid \\ & & \mathit{eq}_T(T,T,K,\mathit{Con}) \mid \mathit{eq}_K(K,K,\mathit{Con}) \mid \\ & & \mathit{type}(T,K,\mathit{Con}) \mid \mathit{term}(t,T,\mathit{Con}) \mid T\!\uparrow\, \equiv\, T \mid T[t] \equiv T' \mid \top \end{array}$$

$$\begin{array}{llllll} \text{Horn clauses} & HC ::= & At \leftarrow At \wedge \ldots \wedge At & \text{Programs} & \mathcal{P} ::= & \cdot \mid P, \mathcal{K} : HC \\ \text{Proof terms} & PT ::= & \mathcal{K}\ PT \ldots PT & \text{Goals} & \mathcal{G} ::= & \mathcal{D} : At \wedge \ldots \wedge \mathcal{D} : At \end{array}$$

The atomic formula $\top$ is intended to stand for a formula that is always true. The predicates $\mathit{eq}^a_t$ and $\mathit{eq}^s_t$ denote algorithmic and structural equality respectively of terms of a certain simple type in a context, the predicates $\mathit{eq}_T$ and $\mathit{eq}_K$ denote equality of terms of a certain simple kind, and equality of kinds in a context respectively. The predicates *term* and *type* denote, respectively, that a term or a type is well-formed in a context. We use $A\!\uparrow\, \equiv\, A'$ to denote that a type $A'$ is the result of shifting of $A$; and we use $A[M] \equiv A'$ to denote that $A'$ is the result of substitution of $A$ with $M$. We use the identifier $H$ to denote Horn clauses in *HC*. Goals in $\mathcal{G}$ are denoted by $G$. We use $P$ with indices to refer to programs. Proof terms in $PT$ are denoted by $\delta$, $\delta_1$, *etc.*

Note that, by Definition 8, atoms in goals are assigned variables. Later, proof terms computed by resolution are identified by these variables (Definition 12). We omit explicit mention of goal variables whenever we do not need to refer to proof terms that the variables identify (fresh variables are assumed in such cases).

### *3.2 From a Refinement Problem to a Logic Program*

In this section, we explain how a term with metavariables is transformed into a goal, and the signature into a logic program. At the end of the section we state that, for a refinement problem, either a goal and a program exist or else the problem cannot be refined to a well-formed term.

We define a calculus with two kinds of judgements, one for transforming refinement problems into goals and the other – for transforming signatures into logic programs. These judgements are defined mutually in a similar way to the well-formedness judgements of nameless LF in Figures

$\boxed{\mathcal{S}; \Gamma; A \vdash (G \mid L)}$

$$\frac{a : L \in \mathcal{S}}{\mathcal{S}; \Gamma; a \vdash (\top \mid L)}\ \textsc{r-tcon} \qquad \frac{}{\mathcal{S}; \Gamma; ?_A \vdash (\mathit{type}(?_A, ?_L, \Gamma) \mid ?_L)}\ \textsc{r-T-meta}$$

$$\frac{\mathcal{S}; \Gamma; A \vdash (G_A \mid L_1) \qquad \mathcal{S}; \Gamma, A; B \vdash (G_B \mid L_2)}{\mathcal{S}; \Gamma; \Pi A.B \vdash (G_A \wedge G_B \wedge \mathit{eq}_K(L_1, \mathtt{type}, \Gamma) \wedge \mathit{eq}_K(L_2, \mathtt{type}, \Gamma) \mid \mathtt{type})}\ \textsc{r-}\Pi\textsc{-intro}$$

$$\frac{\mathcal{S}; \Gamma; A \vdash (G_A \mid L) \qquad \mathcal{S}; \Gamma; M \vdash (G_M \mid B)}{\mathcal{S}; \Gamma; AM \vdash (G_A \wedge G_M \wedge \mathit{eq}_K(L, \Pi B.?_L, \Gamma) \wedge (?_L[M] \equiv ?_{L'}) \mid ?_{L'})}\ \textsc{r-}\Pi\textsc{-elim}$$

Fig. 10. Refinement of types

2 and 3. We use $\mathcal{S}; \Gamma; M \vdash (G \mid A)$ to denote the transformation of a term $M$ in a signature $\mathcal{S}$ and a context $\Gamma$ to a goal $G$. The judgement also synthesises a type $A$ of the term $M$. Similarly, $\mathcal{S}; \Gamma; A \vdash (G \mid K)$ denotes a transformation of a type $A$ in $\mathcal{S}$ and $\Gamma$ to a goal $G$ while synthesising a kind $K$.

*Definition 9* (*Refinement Goals*)

The judgements $\mathcal{S}; \Gamma; M \vdash (G \mid A)$ and $\mathcal{S}; \Gamma; A \vdash (G \mid L)$ are given by inference rules in Figures 9 and 10. Metavariables that do not occur among assumptions have an implicit freshness condition.

The inference judgement for a logic program generation is denoted by $\mathcal{S} \vdash_{\text{Prog}} P$ where $\mathcal{S}$ is a signature and $P$ is a generated logic program. A generated logic program contains clauses that represent inference rules of type theory and clauses that are generated from a signature $\mathcal{S}$. The clauses that represent inference rules of LF are the same for all programs and Definition 10 gives a minimal program $P_e$ that contains only these clauses.

*Definition 10*

Let $P_e$ be a program with clauses that represent inference rules for well-formedness of terms and types:

$$\begin{aligned}
\kappa_{true} :&\quad \top \leftarrow \\
\kappa_0 :&\quad \mathit{proj}(0, ?_A, ?'_A : ?_\Gamma) \leftarrow ?_{A'}\!\uparrow\, \equiv ?_A \\
\kappa_\sigma :&\quad \mathit{proj}(\sigma(?_\iota), ?_A, ?_B : ?_\Gamma) \leftarrow \mathit{proj}(?_\iota, ?'_A, ?_\Gamma) \wedge ?_{A'}\!\uparrow\, \equiv ?_A \\
\kappa_{proj} :&\quad \mathit{type}(?_\iota, ?_A, \mathtt{type}, ?_\Gamma) \leftarrow \mathit{proj}(?_\iota, ?_A, ?_\Gamma) \\
\kappa_{\text{T-elim}} :&\quad \mathit{type}(?_A ?_M, ?_L, ?_\Gamma) \leftarrow \mathit{type}(?_A, \Pi ?_{A_1}.?_{L'}, ?_\Gamma) \wedge \mathit{term}(?_M, ?_{A_2}, ?_\Gamma) \wedge \\
&\qquad \mathit{eq}_T(?_{A_1}, ?_{A_2}, \mathtt{type}, ?_\Gamma) \wedge ?_{L'}[?_M] \equiv ?_L \\
\kappa_{\text{T-intro}} :&\quad \mathit{type}(\Pi ?_A.?_B, \mathtt{type}, ?_\Gamma) \leftarrow \mathit{type}(?_A, \mathtt{type}, ?_\Gamma) \wedge \mathit{type}(?_B, \mathtt{type}, ?_B : ?_\Gamma) \\
\kappa_{\text{t-elim}} :&\quad \mathit{term}(?_M ?_N, ?_B, ?_\Gamma) \leftarrow \mathit{term}(?_M, \Pi ?_{A_1}.?_{B'}, ?_\Gamma) \wedge \mathit{term}(?_N, ?_{A_2}, ?_\Gamma) \wedge \\
&\qquad \mathit{eq}_T(?_{A_1}, ?_{A_2}, \mathtt{type}, ?_\Gamma) \wedge ?_{B'}[?_N] \equiv ?_B \\
\kappa_{\text{t-intro}} :&\quad \mathit{term}(\lambda ?_A.?_M, \Pi ?_A.?_B, ?_\Gamma) \leftarrow \mathit{type}(?_A, \mathtt{type}, ?_\Gamma) \wedge \mathit{term}(?_M, ?_B, ?_\Gamma)
\end{aligned}$$

Further, there are clauses that represent weak algorithmic equality of types, algorithmic and structural equality of termss, and weak head reduction of terms:

$$\begin{aligned}
\kappa_{\text{eqTintro}} :\ & eq_T(\Pi ?_{A_1}.?_{A_2}, \Pi ?_{B_1}.?_{B_2}, type, ?_\Gamma) \leftarrow eq_T(?_{A_1}, ?_{B_1}, type, ?_\Gamma) \wedge \\
& \quad eq_T(?_{A_2}, ?_{B_2}, type, ?_{A_1} : ?_\Gamma) \\
\kappa_{\text{eqTelim}} :\ & eq_T(?_A ?_M, ?_B ?_N, ?_L, ?_\Gamma) \leftarrow eq_T(?_A, ?_B, \Pi ?_C.?_L, ?_\Gamma) \wedge eq_t^a(?_M, ?_N, ?_C, ?_\Gamma) \\
\kappa_{\text{eqtzero}} :\ & eq_t^s(0_\Gamma, 0_\Gamma, ?_A, ?_A : ?_\Gamma)) \leftarrow \\
\kappa_{\text{eqtsucc}} :\ & eq_t^s(\sigma ?_{\iota_\Gamma}, \sigma ?_{\iota'_\Gamma}, ?_A, ?_B : ?_\Gamma) \leftarrow eq_t^s(?_{\iota_\Gamma}, ?_{\iota'_\Gamma}, ?_A, ?_\Gamma) \\
\kappa_{\text{eqtrefl}} :\ & eq_t^s(?_a, ?_a, ?_A, ?_\Gamma) \leftarrow \\
\kappa_{\text{eqtelim}} :\ & eq_t^s(?_{M_1} ?_{M_2}, ?_{N_1} ?_{N_2}, ?_B, ?_\Gamma) \leftarrow eq_t^s(?_{M_1}, ?_{N_1}, \Pi ?_A.?_B, ?_\Gamma) \wedge eq_t^a(?_{M_2}, ?_{N_2}, ?_B, ?_\Gamma) \\
\kappa_{\text{eqtwhrl}} :\ & eq_t^a(?_M, ?_N, ?_A, ?_\Gamma) \leftarrow whr(?_M, ?_{M'}), eq_t(?_{M'}, ?_N, ?_A, ?_\Gamma)
\end{aligned}$$

$$\begin{aligned}
\kappa_{\text{eqtwhrr}} :\ & eq_t^a(?_M, ?_N, ?_A, ?_\Gamma) \leftarrow whr(?_N, ?_{N'}), eq_t^a(?_M, ?_{N'}, ?_A, ?_\Gamma) \\
\kappa_{\text{eqtstr}} :\ & eq_t^a(?_M, ?_N, ?_A, ?_\Gamma) \leftarrow eq_t^s(?_M, ?_N, ?_A, ?_\Gamma) \\
\kappa_{\text{eqtexp}} :\ & eq_t^a(?_M, ?_N, \Pi ?_A.?_B, ?_\Gamma) \leftarrow ?_M\uparrow \equiv ?_{M'}, ?_N\uparrow ?_{N'} \wedge eq_t^a(?_{M'}0, ?_{N'}0, ?_B, ?_A : ?_\Gamma) \\
\kappa_{\text{eqsimpl}} :\ & eq_t^a(?_M, ?_{M'}, ?_A ?_N, ?_\Gamma) \leftarrow eq_t^a(?_M, ?_{M'}, ?_A, ?_G) \\
\kappa_{\text{whrs}} :\ & whr((\lambda ?_A.?_M)?_N, ?_{M'}) \leftarrow ?_M[?_N/0] \equiv ?_{M'} \\
\kappa_{\text{whrh}} :\ & whr(?_M ?_N, ?_{M'} ?_N) \leftarrow whr(?_M, ?_{M'})
\end{aligned}$$

Finally, there are clauses that represent shifting and substitution on terms and types:

$$\begin{aligned}
\kappa_{\text{shiftTtintro}} :\ & (\Pi ?_A.?_M)\uparrow^\iota \equiv\ (\Pi ?_{A'}.?_{M'}) \leftarrow ?_A\uparrow^\iota \equiv ?_{A'} \wedge ?_M\uparrow^{\sigma\iota} \equiv ?_{M'} \\
\kappa_{\text{shiftTtintro}} :\ & (\lambda ?_A.?_M)\uparrow^\iota \equiv\ (\lambda ?_{A'}.?_{M'}) \leftarrow ?_A\uparrow^\iota \equiv ?_{A'} \wedge ?_M\uparrow^{\sigma\iota} \equiv ?_{M'} \\
\kappa_{\text{shifttelim}} :\ & (?_M ?_N)\uparrow^\iota \equiv\ (?_{M'} ?_{N'}) \leftarrow ?_M\uparrow^\iota \equiv ?_{M'} \wedge ?_N\uparrow^\iota \equiv ?_{N'} \\
\kappa_{\text{shifttgt}} :\ & \iota\uparrow^0 \equiv\ \sigma\iota \leftarrow \\
\kappa_{\text{shifttpred}} :\ & 0\uparrow^{\sigma\iota} \equiv\ 0 \leftarrow \\
\kappa_{\text{shifttstep}} :\ & \sigma\iota\uparrow^{\sigma\iota'} \equiv\ \sigma\iota'' \leftarrow \iota\uparrow^{\iota'} \equiv\ \iota'' \\
\kappa_{\text{substTtintro}} :\ & (\Pi ?_A.?_M)[?_N/\iota] \equiv (\Pi ?_{A'}.?_{M'}) \leftarrow (?_A[?_N/\iota] \equiv ?_{A'}) \wedge (?_N\uparrow^0 \equiv ?'_N) \wedge ?_M[?'_N/\sigma\iota] \equiv ?_{M'} \\
\kappa_{\text{substintro}} :\ & (\lambda ?_A.?_M)[N/\iota] \equiv (\lambda ?_{A'}.?_{M'}) \leftarrow (?_A[\iota/?_{A'}] \equiv) \wedge (?_N\uparrow^0 \equiv ?'_N) \wedge ?_M[?'_N/\sigma\iota] \equiv ?_{M'} \\
\kappa_{\text{substtelim}} :\ & (?_{M_1} ?_{M_2})[?_N/\iota] \equiv (?_{M'_1} ?_{M'_2}) \leftarrow ?_{M_1}[?_N/\iota] \equiv ?_{M'_1} \wedge ?_{M_2}[?_N/\iota] \equiv ?_{M'_2} \\
\kappa_{\text{substz}} :\ & 0[?_N/0] \equiv ?_N \leftarrow \\
\kappa_{\text{substs}} :\ & 0[?_N/\sigma\iota] \equiv 0 \leftarrow \\
\kappa_{\text{substgt}} :\ & \sigma\iota[?_N/0] \equiv \sigma\iota \leftarrow \\
\kappa_{\text{substpred}} :\ & \sigma\iota[?_N/\sigma\iota'] \equiv \sigma\iota'' \leftarrow \iota[?_N/\iota'] \equiv \iota''
\end{aligned}$$

The clauses in Definition 10 correspond to judgements in Figures 2–5. They are direct translations of the inference rules of nameless LF in these figures. The judgement $\mathcal{S} \vdash_{\text{Prog}} P$ extends $P_e$ with a clause for each type and term constant in $\mathcal{S}$ and initialises shifting and substitution with term and type-level constants as constant under the operation.

*Definition 11* (*Refinement Program*)
The judgement $\mathcal{S} \vdash_{\text{Prog}} P$ is given by the inference rules of Figure 11.

The Figure 11 gives definition of signature refinement. The refinement judgement of a signature into a program concludes our transformation of refinement problem into a goal and a program.

*Theorem 1* (*Decidability of Goal Construction*)
Let $M$ be a refinement problem in a well-formed signature $\mathcal{S}$ and a well-formed context $\Gamma$ such that a solution $(\rho, R)$ exists. Then inference rules in Figures 9 and 10 construct the goal $G$ and the extended type $A$ such that $\mathcal{S}; \Gamma; M \vdash (G \mid A)$.

$\boxed{\mathcal{S} \vdash_{\text{Prog}} P}$

$$\frac{}{\cdot \vdash_{\text{Prog}} P_{\text{e}}} \qquad \frac{\mathcal{S} \vdash_{\text{Prog}} P}{\begin{array}{c}\mathcal{S}, c : A \vdash_{\text{Prog}} P, \quad \kappa_c : \mathit{term}(c, A, ?_\Gamma) \leftarrow, \quad \kappa_{\text{shift}_c} : (c{\uparrow}^0 \equiv\ c) \leftarrow, \\ \kappa_{\text{subst}_c} : c[?_M/0]] \equiv c \leftarrow, \quad \kappa_{eq^s_c} : eq^s(c, c, A, ?_\Gamma) \leftarrow\end{array}}$$

$$\frac{\mathcal{S} \vdash_{\text{Prog}} P}{\begin{array}{c}\mathcal{S}, a : L \vdash_{\text{Prog}} P, \quad \kappa_{\text{shift}_\alpha} : (\alpha{\uparrow}^0 \equiv\ \alpha) \leftarrow, \kappa_{\text{subst}_\alpha} : \alpha[?_M/0] \equiv \alpha \leftarrow, \quad \kappa_{eq_T} : eq_T(\alpha, \alpha, L, ?_\Gamma) \leftarrow, \\ \kappa_{{eq^a}_\alpha} : eq^a(?_N, ?_M, \alpha, ?_\Gamma) \leftarrow eq^s(?_M, ?_N, \alpha, ?_\Gamma)\end{array}}$$

Fig. 11. Refinement of signatures, with operations

*Proof*
By induction on the derivation of the well-formedness judgement of $(\rho, R)(M)$. □

The next example illustrates the construction of a refinement goal.

*Example 2* (*From an Extended Nameless Term to a Goal*)
Let us take the refinement problem $M' = (\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ \ 0)(\lambda ?_A . ?_b)$ and the implicit context and signature from Example 1. By Theorem 1 we can generate $G$ such that the judgement $\mathcal{S}; \Gamma_1; M' \vdash (G \mid ?_{B_7})$ holds:

$$\begin{gathered} G = \top \wedge \top \wedge eq_T(\Pi\, \texttt{bool}\, .(\Pi(\texttt{maybe}_\texttt{A}\, 0_T).(\Pi(\Pi(2_T \equiv_{\texttt{bool}} \texttt{ff}).\, \texttt{A}).(\Pi(\Pi(3_T \equiv_{\texttt{bool}} \texttt{tt}).(\Pi\, \texttt{A}\, .\, \texttt{A})).\, \texttt{A}))), \\ \Pi\, \texttt{bool}\, .?_{B_1}, \Pi\, \texttt{type}\, .?_{L_1}, \Gamma_1) \wedge ?_{B_1}[\texttt{tt}\, /0_T] \equiv ?_{B_2} \wedge \top \wedge \\ eq_T(?_{B_2}, \Pi(\texttt{maybe}_\texttt{A}\, \texttt{tt}).?_{B_3}, \Pi\, \texttt{type}\, .?_{L_2}, \Gamma_1) \wedge ?_{B_3}[0_\Gamma/0_T] \equiv ?_{B_4} \wedge \\ \mathit{type}(?_A, ?_{L_3}, \Gamma_1) \wedge ?_b : \mathit{term}(?_b, ?_{A_1}, ?_A : \Gamma_1) \wedge eq_K(?_{L_3}, \texttt{type}, \Gamma_1) \wedge ?_{A_1}[0_T/0_\Gamma] \equiv ?_{B_5} \wedge \\ eq_T(?_{B_4}, \Pi(\Pi ?_A . ?_{B_5}).?_{B_6}, \Pi \mathit{type}.?_{L_5}, \Gamma) \wedge ?_{B_6}[(\lambda ?_A . ?_b)/0_T] \equiv ?_{B_7} \end{gathered}$$

That is, the type of $M'$ will be computed as a substitution for logic variable $?_{B_7}$ and resolving the goal computes assignments to $?_A$ and $?_b$ as well.

*Proposition 1* (*Decidability of Program Construction*)
Let $\mathcal{S}$ be a signature. Then inference rules in Figure 11 construct the program $P$ such that $\mathcal{S} \vdash_{Prog} P$.

We develop our running example further to illustrate the proposition:

*Example 3* (*From a Signature to a Program*)
The signature $\mathcal{S}$ contains $\texttt{elim}_{\equiv_{\texttt{bool}}}$ hence the generated program contains the clause:

$$\kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}} : \mathit{term}(\texttt{elim}_{\equiv_{\texttt{bool}}}, \Pi\, \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}\, .\, \texttt{A}, \texttt{type}, ?_\Gamma) \leftarrow$$

The following clauses come from the program $P_e$ and represent inference rules of the internal language:

$$\begin{aligned} \kappa_0 : \mathit{term}(0, ?_A, ?_{A'} : ?_\Gamma) \quad & \leftarrow ?_{A'}{\uparrow} \equiv ?_A \\ \kappa_{\texttt{elim}} : \mathit{term}(?_a ?_b, ?_B, ?_\Gamma) \quad & \leftarrow \mathit{term}(?_a, \Pi ?_A . ?_{B'}, ?_\Gamma) \wedge \mathit{term}(?_b, ?_A, ?_\Gamma) \wedge \\ & \qquad eq_T(?_B, ?_{B'}, \texttt{type}, ?_\Gamma) \wedge ?_{B'}[?_b] \equiv ?_B \end{aligned}$$

Example 2 shows unresolved meta-variables in the goal, and Example 3 gives a program against which to resolve the goal. Now the proof-relevant resolution comes into play.

## 4 Proof-Relevant Resolution and Soundness

We utilise a variant of proof-relevant resolution (Fu and Komendantskaya 2017) as the inference engine for solving refinement problems. A substitution of logic variables as well as substitution composition is defined in the usual way. We use $\theta$, $\theta'$ to denote substitutions and $\theta \circ \theta'$ to denote composition of substitutions $\theta$ and $\theta'$. We use $[\delta_1/\gamma_1, \ldots, \delta_n/\gamma_n]$ to denote an assignment that assigns, in order, proof terms $\delta_1$ to $\delta_n$ to proof-term variables $\gamma_1$ to $\gamma_n$. The resolution judgement $P \vdash^{\theta}_{[\delta_1/?_{\gamma_1}, \ldots, \delta_n/?_{\gamma_n}]} G$ states that a goal $G$ is resolved by a program $P$ while computing an answer substitution $\theta$ and an assignment of proof terms. The judgement makes use of an auxiliary judgement for resolution of atomic goals with a proof term $\delta$, denoted $P \vdash^{\theta}_{[\delta_1/?_{\gamma_1}, \ldots, \delta_n/?_{\gamma_n}]} \delta : At$.

*Definition 12* (*Proof-Relevant Resolution*)
Let $P$ be a program and $G$ a goal, $At$, $At_1, \ldots, At_n$ be atomic formulae, $\delta, \delta_1, \ldots, \delta_n$ be proof terms and $\gamma_1, \ldots, \gamma_n$ proof-term variables. The judgements $P \vdash^{\theta}_{[\delta_1/\gamma_1, \ldots, \delta_n/\gamma_n]} G$ and $P \vdash^{\theta} \delta : At$ are defined by the inference rules

$$\frac{P \vdash^{\theta} \delta_1 : At_1 \qquad \ldots \qquad P \vdash^{\theta} \delta_n : At_n}{P \vdash^{\theta}_{[\delta_1/\gamma_1, \ldots, \delta_n/\gamma_n]} \gamma_1 : At_1 \wedge \cdots \wedge \gamma_n : At_n}$$

and

$$\kappa : At' \leftarrow At_1 \wedge \cdots \wedge At_n \in P \; \frac{P \vdash^{\theta'} \delta_1 : \theta At_1 \qquad \ldots \qquad P \vdash^{\theta'} \delta_n : \theta At_n}{P \vdash^{\theta \circ \theta'} \kappa \delta_1 \ldots \delta_n : At}$$

assuming that $\theta At' = \theta At$.

We do not discuss a particular resolution strategy here, but instead refer the reader to the work of Fu and Komendantskaya (2017).

Assume that $G$ and $P$ are a goal and a program that originate from a refinement problem $M$ in signature $\mathcal{S}$. An answer substitution for $G$ computed by $P$ provides a solution to the type-level metavariables in $M$. Similarly the computed assignment of proof terms to proof variables provides a solution to the term-level metavariables in $M$.

We continue with our running example, building upon Examples 1–3.

*Example 4* (*Proof-relevant Resolution Trace*)
The resolution trace of our example is rather long, and we show only a fragment. Suppose that, in several resolution steps denoted by $\leadsto^*$, the goal $G$ given in Example 2 resolves as follows:

$$G \leadsto^* \quad ?_b : \mathit{term}(?_b, \mathtt{A}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} : \Gamma_1)$$

The computed substitution assigns $(\Pi(\Pi(\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{tt}).(\Pi\, \mathtt{A} \,.\, \mathtt{A})).\, \mathtt{A}$ to the logic variable $?_{B_7}$. We now show the trace for the remaining goal $?_b : \mathit{term}(?_b, \mathtt{A}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} : \Gamma_1)$. Given the clauses of Example 3, a resolution trace that computes a proof term for $?_b$ can be given as follows:

$$\begin{aligned}
&\mathit{term}(?_a, \mathtt{A}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} : \Gamma_1) \leadsto_{\kappa_{\mathtt{elim}}} \\
&\quad \mathit{term}(?_{a_1} ?_{a_2}, \Pi ?_A . ?_{B'}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} : \Gamma_1) \wedge \mathit{term}(?_{a'_2}, ?_A, \Gamma_1)) \\
&\qquad\qquad\qquad\qquad \wedge\, \mathit{eq}_T(?_{B_4}, ?_{B'}, \mathtt{type}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} : \Gamma_1) \leadsto^{[?_A \mapsto \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}, ?_{B'} \mapsto \mathtt{A}]}_{\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}} \\
&\quad \mathit{term}(?_{a'_2}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}, (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} : \Gamma_1)) \wedge \mathit{eq}_T(\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}, \mathtt{type}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} : \Gamma_1)) \leadsto_{\kappa_0} \\
&\quad \mathit{eq}_T(\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}, \mathtt{type}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} : \Gamma_1)) \leadsto^* \bot
\end{aligned}$$

Above, we omit writing full proof terms, but record the witnesses as indices of $\leadsto$. The assignment to the logic variable $?_A$ is $\mathtt{A}$ and the subterm of the computed proof term that is bound to $?_b$ is $\kappa_{\mathtt{elim}} \kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}} \kappa_0 \delta_{eq_T}$ where the subterm $\delta_{eq_T}$ is a witness of the appropriate type equality.

Since we have used the types and terms of nameless LF to define our atomic formulae, the computed substitution can be used directly. The interpretation of the computed assignment of proof terms depends on assignment of atomic proof term symbols in the program $P_e$. We define a mapping that gives the intended interpretation:

*Definition 13* (*Interpretation of Proof Terms*)
We define interpretation of proof terms $\ulcorner - \urcorner : PT \to T$ as follows:

$$\begin{array}{ccc} \ulcorner \kappa_{\sigma}\delta\delta_{\iota}\delta' \urcorner = \sigma\ulcorner \delta_{\iota} \urcorner & \ulcorner \kappa_{\text{T-intro}}\delta_A\delta\delta_B \urcorner = \Pi\ulcorner \delta_A \urcorner.\ulcorner \delta_B \urcorner & \ulcorner \kappa_0 \urcorner = 0 \\ \ulcorner \kappa_{proj}\delta_{\iota} \urcorner = \ulcorner \delta_{\iota} \urcorner & \ulcorner \kappa_{\text{t-elim}}\delta_A\delta_M\delta\delta' \urcorner = \ulcorner \delta_A \urcorner\ulcorner \delta_M \urcorner & \ulcorner \kappa_c \urcorner = c \\ \ulcorner \kappa_{\text{T-elim}}\delta_M\delta_N\delta\delta' \urcorner = \ulcorner \delta_M \urcorner\ulcorner \delta_N \urcorner & \ulcorner \kappa_{\text{t-intro}}\delta_A\delta\delta_M\delta' \urcorner = \lambda\ulcorner \delta_A \urcorner.\overrightarrow{\ulcorner \delta_M \urcorner} & \ulcorner \kappa_a \urcorner = a. \end{array}$$

We extend $\ulcorner - \urcorner$ to assignments of proof terms by composition and we use $\ulcorner R \urcorner$ to denote the composition $(\ulcorner - \urcorner \circ R)$.

*Example 5*
In Example 4, the computed proof term bound to $?_b$ will be interpreted as follows:

$$\ulcorner \kappa_{\mathtt{elim}}\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\kappa_0\delta_{eqT} \urcorner = \mathtt{elim}_{\equiv_{\mathtt{bool}}}\,0$$

Hence, the original problem is refined to $\mathtt{elim}_{\mathtt{maybe}_\mathtt{A}}\ \ \mathtt{tt}\ \ 0\ (\lambda\,\mathtt{A}\,.\,\mathtt{elim}_{\equiv_{\mathtt{bool}}}\,0)$ while the computed type is $((\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{tt}) \to \mathtt{A} \to \mathtt{A}) \to \mathtt{A}$ .

Finally, the above interpretation allows us to state the soundness of our system:

*Theorem 2* (*Soundness of Proof-Relevant Resolution for Generated Goals and Programs*)
Let $M$ be a term in the extended syntax with signature $\mathcal{S}$. Let $P$ and $G_M$ be a program and a goal such that $\mathcal{S}, \cdot \vdash (G_M|A)$ and $\mathcal{S} \vdash_{\text{Prog}} P$ respectively. Let $\rho$, $R$ be a substitution and a proof term assignment computed by proof-relevant resolution such that $P \vdash_R^{\rho} G_M$. Then if there is a solution for a well-formed term, then there are solutions $(\rho', R')$ and $(\rho'', R'')$ such that $(\rho', R')M$ is a well-formed term and

$$(\rho'', R'')((\rho, \ulcorner R \urcorner)M) = (\rho', R')M$$

*Proof*
Generalise the statement of the theorem for an arbitrary well-formed context $\Gamma$. By simultaneous induction on derivation of the well-formedness judgement of $(\rho', R')M$ and derivation of $P \vdash_R^{\rho} G$. The theorem follows from the generalisation. $\square$

Theorem 2 guarantees that the refinement computed in Examples 2–5 is well-typed in the internal language. That is, there is a derivation of the following judgement:

$$\mathcal{S}; \cdot, \mathtt{maybe}_\mathtt{A}\,\mathtt{tt} \vdash \mathtt{elim}_{\mathtt{maybe}_\mathtt{A}}\ \ \mathtt{tt}\ \ 0\ (\lambda\,\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}\,.\,\mathtt{elim}_{\equiv_{\mathtt{bool}}}\,0) : (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{tt}) \to \mathtt{A} \to \mathtt{A}) \to \mathtt{A}$$

We omit the actual derivation of the judgement. However, note that it can be easily reconstructed in a similar way as the intended interpretation of proof terms is computed in Definition 13. For example, in case of our running example, the subterm $\delta_{eqT}$ of the proof term gives derivation of the definitional equality that is necessary to verify application of $\mathtt{elim}_{\equiv_{\mathtt{bool}}}$ to index 0.

## *Implementation*

We have formalised the results in this paper using the Ott tool and the Coq theorem prover. The source code can be found online[1]. Most importantly, we formalise definitions of nameless LF. The exported Coq definitions are then used in formalisation of decidability of the refinement calculus. An implementation of translation from the extended language to logic programs and goals is obtained from the formal proof via code extraction into OCaml. A parser is extracted from the formalisation of the grammar as well. The translation outputs logic programs and goals suitable for an external resolution engine.

[1] `https://github.com/frantisekfarka/slepice`

## 5 Related and Future Work

Ideas underlying our work originate in the work of Stuckey and Sulzmann (2002) on HM(X) type inference as (constraint) logic programming. There are two key differences. First, in our work we consider dependent types. Other approaches, such as that of Sulzmann and Stuckey do not give a motivation for the shape of generated logic goals and programs. We make explicit that atomic formulae represent judgements of the type theory and that the program originates on one hand from inference rules of the type theory and on the other from a signature of a term. We believe that a clear identification of this interpretation of generated goals and programs makes it feasible to adjust the refinement calculus for different type theories.

Currently implemented systems (*cf.* Pientka (2013)) make use of a bidirectional approach to type checking. That is, there are separate type checking and type synthesis phases. The key difference between these systems and our own work is that we do not explicitly discuss bidirectionality. Combining this with a clear identification of atomic formulae with judgements, and Horn clauses with inference rules, in our opinion, makes the presentation significantly more accessible. However, bidirectionality in our system is still implicitly present, albeit postponed to the resolution phase. As future work, we intend to analyse structural resolution (Fu and Komendantskaya 2017) for the generated goals. We intend to show that the matching steps in the resolution correspond to type checking in the bidirectional approach whereas resolution steps by unification correspond to type synthesis.

In future work, we would like to consider additional constructs in the surface language. One example of such a construct would be a type-class mechanism, as found in *e.g.* Haskell. Fu et al. (2016) have previously demonstrated that type class resolution in Haskell can be addressed by proof-relevant resolution in Horn clause logic with explicit proof terms. This result suggests a natural extension of our work by adding a new atomic formula that represents type-class judgement and by adjusting refinement calculus with inference rules for the translation of type-class judgements. Recently, Bottu et al. (2017) argued for quantification on type-class constraints. Although such quantification escapes the Horn clause fragment as discussed in the work of Fu *et al.*, it can be addressed in the logic of hereditary Harrop formulae. Our approach scales well by replacing the target logic by the logic of first-order hereditary Harrop formulae. Finally, we believe that our suggested approach to refinement can also serve as a viable method for proving the soundness of the surface language w.r.t. a semantic model. We have already presented some initial results (Farka et al. 2016) that show that proof-relevant type class resolution is sound w.r.t. to both inductive and coinductive interpretation of type class instances.

## 6 Conclusion

Functional programming languages are a convenient tool for developing software. Dependent types in particular allow various semantic properties to be encoded in types. However, as evidenced by languages such as Agda, Coq or Idris, software development in a functional programming language with dependent types is a complex task. The usability of such languages critically depends on the amount of assistance that an automated tool provides to a programmer: at a minimum, some type refinement is necessary. Current descriptions are implementation dependent and hard to understand. This complicates the reuse of existing approaches in the development of tools for new languages. Ultimately, it is problematic for a programmer as they need to understand the elaboration process to some extent. We present a description of refinement in LF that is significantly simpler than the existing ones. We show a translation of an incomplete term with metavariables to a goal and a program in Horn clause logic by a syntactic traversal of the term. The inference is then performed by proof-relevant resolution that is an extension of the standard first-order resolution with proof term construction. Moreover, the generated goal and program have a straightforward interpretation as judgements of type theory and inference rules and hence can be easier to understand and work with.

Our discussion of related work shows how our approach links to the state of the art in refinement in functional languages. We have suggested two different areas for future work. Firstly, a more detailed analysis of resolution on generated goals and programs may recover bidirectionality. Secondly, we have discussed a possible extension to the surface language, higher order features and type classes, and argued that our approach scales well with extensions of the surface language. However, extensions of the surface language may require a stronger logic as a target logic of the refinement calculus.

*Acknowledgements* This work has been supported by the EPSRC grant "Coalgebraic Logic Programming for Type Inference" EP/K031864/1-2, EU Horizon 2020 grant "RePhrase: Refactoring Parallel Heterogeneous Resource-Aware Applications - a Software Engineering Approach" (ICT-644235), and by COST Action IC1202 (TACLe), supported by COST (European Cooperation in Science and Technology). We would like to thank Andreas Abel, Greg Michaelson, and anonymous referees for helpful comments on earlier versions of the paper.

## References

Ahn, K. Y. and Vezzosi, A. 2016. Executable relational specifications of polymorphic type systems using prolog. See Kiselyov and King (2016), 109–125.

Bottu, G., Karachalias, G., Schrijvers, T., d. S. Oliveira, B. C., and Wadler, P. 2017. Quantified class constraints. See Diatchki (2017), 148–161.

Brady, E. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program. 23,* 5, 552–593.

Diatchki, I. S., Ed. 2017. *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017.* ACM.

Farka, F., Komendantskaya, E., and Hammond, K. 2016. Coinductive soundness of corecursive type class resolution. In *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, M. V. Hermenegildo and P. López-García, Eds. Lecture Notes in Computer Science, vol. 10184. Springer, 311–327.

Fu, P. and Komendantskaya, E. 2017. Operational semantics of resolution and productivity in horn clause logic. *Formal Asp. Comput. 29,* 3, 453–474.

Fu, P., Komendantskaya, E., Schrijvers, T., and Pond, A. 2016. Proof relevant corecursive resolution. See Kiselyov and King (2016), 126–143.

Gonthier, G. and Mahboubi, A. 2010. An introduction to small scale reflection in Coq. *J. Formalized Reasoning 3,* 2, 95–152.

Harper, R. and Pfenning, F. 2005. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Log. 6,* 1, 61–101.

Hemann, J., Friedman, D. P., Byrd, W. E., and Might, M. 2016. A small embedding of logic programming with a simple complete search. In *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, R. Ierusalimschy, Ed. ACM, 96–107.

Karachalias, G. and Schrijvers, T. 2017. Elaboration on functional dependencies: functional dependencies are dead, long live functional dependencies! See Diatchki (2017), 133–147.

Kiselyov, O. and King, A., Eds. 2016. *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings.* Lecture Notes in Computer Science, vol. 9613. Springer.

Miller, D. and Nadathur, G. 2012. *Programming with Higher-Order Logic.* Cambridge University Press.

Milner, R. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17,* 348–375.

Odersky, M., Sulzmann, M., and Wehr, M. 1999. Type inference with constrained types. *TAPOS 5,* 1, 35–55.

Peyton Jones, S., Jones, M., and Meijer, E. 1997. Type classes: an exploration of the design space. In Haskell workshop.

Pientka, B. 2013. An insider's look at LF type reconstruction: everything you (n)ever wanted to know. *J. Funct. Program. 23,* 1, 1–37.

Pientka, B. and Dunfield, J. 2010. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, J. Giesl and R. Hähnle, Eds. Lecture Notes in Computer Science, vol. 6173. Springer, 15–21.

Slama, F. and Brady, E. 2017. Automatically proving equivalence by type-safe reflection. In *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke, Eds. Lecture Notes in Computer Science, vol. 10383. Springer, 40–55.

Stuckey, P. J. and Sulzmann, M. 2002. A theory of overloading. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, M. Wand and S. L. P. Jones, Eds. ACM, 167–178.

Sulzmann, M. and Stuckey, P. J. 2008. HM(X) type inference is CLP(X) solving. *J. Funct. Program. 18,* 2, 251–283.

Vazou, N., Tondwalkar, A., Choudhury, V., Scott, R. G., Newton, R. R., Wadler, P., and Jhala, R. 2018. Refinement reflection: complete verification with SMT. *PACMPL 2,* POPL, 53:1–53:31.

Weirich, S., Voizard, A., de Amorim, P. H. A., and Eisenberg, R. A. 2017. A specification for dependent types in haskell. *PACMPL 1,* ICFP, 31:1–31:29.