Sulzmann, Martin and Wang, Meng (2004) ***A Systematic Translation of Guarded Recursive Data Types to Existential Types.*** **Technical report. National University of Singapore (Unpublished)**

A Systematic Translation of Guarded Recursive Data Types to Existential Types

Martin Sulzmann
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

Meng Wang
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
wangmeng@comp.nus.edu.sg

ABSTRACT

Guarded recursive data types (GRDT) are a new language feature which allows to type check the different branches of case expressions under different type assumptions. We show that GRDT can be translated to type classes with existential types (TCET). The translation to TCET might be problematic in the sense that common implementations such as the Glasgow Haskell Compiler (GHC) fail to accept the translated program. We establish some sufficient conditions under which we can provide for a refined translation from TCET to existential types (ET) based on a novel proof term construction method. The resulting ET program is accepted by GHC. The sufficient conditions are met by all GRDT examples we have found in the literature. Our work can be seen as the first formal investigation to relate the concepts of guarded recursive data types and (type classes with) existential types.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism,Constraints*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms

Languages, Theory

Keywords

Type systems, type-directed translation, proof-term construction, constraint solving.

1. INTRODUCTION

Guarded recursive data types (GRDT) [28] introduced by Xi, Chen and Chen are a new language feature which allows to type check more programs. The basic idea is to use different type assumptions for each branch of a case expression. E.g., consider the following (toy) GRDT program. We will use Haskell-style syntax [8] throughout the paper.

Example 1 We introduce a GRDT $Erk\ a$ where a may be refined depending on the constructor. Function `f` takes advantage of the temporary equality assumptions enabled by pattern matching.

```
data Erk a = (a=Int) => I a
           | forall b.(a=[b]) => L a
f :: Erk a -> a
f (I x) = x + 1
f (L x) = tail x
```

In detail, the data type definition introduces two constructors belonging to data type $Erk\ a$. The novelty of GRDT is that in case of constructor `I` we refine the type to $Erk\ Int$. We present type refinement in terms of equations. In case of `L` we refine the type to $Erk\ [b]$ for some b. Note that GRDT imply existential types [14]. Constructor `L` has type $\forall a, b.(a = [b]) \Rightarrow a \rightarrow Erk\ a$. Therefore, all variables not appearing in the result type are bound by the `forall` keyword. Note that some presentations [4] write `I a with (a=Int)` instead of `(a=Int) => I a`. The important point is that when pattern matching over values we can make use of these additional type assumptions. Consider the function definition where in the first clause we temporarily add $a = Int$ to our assumptions (assuming that `x` has type a). Thus, we can verify that the `x+1` has type a. A similar observation applies to the second clause. Hence, function `f` is type correct. □

GRDT have been recognized as a very useful language feature, .e.g. consider [20, 17, 18]. Hence, it is desirable to extend existing languages with GRDT. In fact, a number of authors [1, 2, 3, 27] have recognized that GRDT-style behavior can be expressed in terms of some existing language features already available in Haskell. All of these encodings share the same idea and represent type equalities by Haskell terms.

Example 2 Here is an encoding of Example 1 in terms of existential types [14]. We introduce a special data type $E\ a\ b$ to represent equality assumption among types. E.g., we represent $a = Int$ by $E\ a\ Int$ where the associated value `E` (g,h) implies functions g and h to convert a's to and from Int's.

```
data E a b = E (a->b,b->a)
data Erk_H' a = I_H' a (E a Int)
              | forall b. L_H' a (E a [b])
f_H' :: Erk_H' a -> a
f_H' (I_H' x (E (g,h))) = h ((+) (g x) 1)
f_H' (L_H' x (E (g,h))) = h (tail (g x))
```

Note that we use function notation for addition. Operationally, the conversion functions are assumed to represent the identity. Hence, the above program is equivalent to Example 1. The above program makes only use of existential types and is therefore accepted by GHC [6]. However, the programmer has to do now more work when defining the function body. In the first clause, we turn x into a value of type Int by making use the explicitly provided conversion function g of type $a \to Int$. Then, we apply (+) which is assumed to have type $Int \to Int \to Int$. Finally, we apply h to obtain a value of type a such that the type annotation is matched. □

Clearly, such a style of programming is rather tedious and should be best performed by an automatic tool. To the best of our knowledge, we are the first to propose a *systematic* translation method from GRDT to ET (existential types) by means of a source-to-source translation. We see our work as a more principled answer to the many examples we have seen so far in the literature [1, 2, 3, 16, 27]. The essential task is to construct proof terms for type equalities out of logical statements of the form $C \supset t_1 = t_2$ where C consists of a set of type equations and $\supset$ denotes Boolean implication. One of our main technical contribution is a decidable proof term construction method for (directed) type equalities. Under the assumption that type assumptions are decomposable we achieve a translation from GRDT to existential types (ET) which is accepted by GHC. In our experience, the decomposable assumption is satisfied by all GRDT examples we have seen in the literature.

We continue in Section 2 where we introduce some basic notations. In Section 3 we define the set of well-typed GRDT programs. Section 4 provides for an (intermediate) translation from GRDT to type classes with existential types (TCET). Section 5 provides for a translation scheme from GRDT to ET based on a proof system for type equalities. The translation scheme is complete if types are decomposable. In Section 6 we show that the proof system is decidable. In Section 7 we show how to combine our proof term construction method with a novel inference method. Related work is discussed in Section 8. We conclude in Section 9. Due to space limitations proofs for all results stated have been moved to the Appendix.

2. PRELIMINARIES

We write $\bar{o}$ to denote a sequence of objects $o_1, ..., o_n$. We write $fv(o)$ to denote the set of free variables in some object o.

We assume that the reader is familiar with the concepts of substitution, unifiers, most general unifiers (m.g.u.) etc [12]. E.g., $[t/a]$ denotes the substitution which has the effect of replacing each occurrence of a by t. Often, we abbreviate $[t_1/a_1, ..., t_n/a_n]$ by $[\bar{t}/\bar{a}]$.

We make use of constraints C consisting of conjunction of primitive constraints such as $t_1 = t_2$ describing equality among t_1 and t_2. We often treat constraints as sets, therefore, we use "," as a short-hand for Boolean conjunction.

We also assume basic familiarity with first-order logic. We write $\models$ to denote the model-theoretic entailment relation, $\supset$ to denote Boolean implication and $\leftrightarrow$ to denote Boolean equivalence. We let $\bar{\exists}_W F$ denote the formula $\exists \alpha_1 \ldots \exists \alpha_n F$ where $\{\alpha_1, \ldots, \alpha_n\} = fv(F) - W$. We refer to [21] for details.

3. GUARDED RECURSIVE DATA TYPES

In this section, we define the set of well-typed GRDT programs. Note that there exist several variations of GRDT such as Cheney's and Hinze's first-class phantom types [4], Peyton-Jones's, Washburn's and Weirich's generalized algebraic data types [10] and equality-qualified types by Sheard and Pasalic [19]. Our formulation is closest to the system described by Simonet and Pottier [22].

First, we define the set of expressions and types.

Expressions	e	$::=$	$K \mid x \mid \lambda x.e \mid e\ e \mid \mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i \in I}$
Patterns	p	$::=$	$x \mid (p,p) \mid K\ \bar{p}$
Types	t	$::=$	$a \mid t \to t \mid T\ \bar{t}$
Type Schemes	σ	$::=$	$t \mid \forall \bar{a}.C \Rightarrow t$

For simplicity, we leave out let-definitions and type annotations but may make use of them in examples. Note that pattern matching syntax used in examples can be straightforwardly expressed in terms of case expressions.

GRDT definitions in example programs such as

```
data Erk a = (a=Int) => I a | forall b.(a=[b]) => L a
```

imply constructors $I : \forall a.a = Int \Rightarrow a \to Erk\ a$ and $L : \forall a, b.a = [b] \Rightarrow a \to Erk\ a$. We prohibit "invalid" definitions such as `data Unsat a = (a=(a,Int)) => U a` which yields a constructor with an unsatisfiable set of equations. We assume that booleans, integers, pairs and lists are predefined.

The typing rules describing well-typing of GRDT expressions are in Figure 1. We introduce judgments $C, \Gamma \vdash^G e : t$ to denote that expression e has type t under constraint C and environment Γ. We assume that C consists of conjunction of equations. A judgment is valid if we find a derivation w.r.t. the typing rules. Note that in Γ we record the types of lambda-bound variables and primitive functions such as $head : \forall a.[a] \to a$, $tail : \forall a.[a] \to [a]$ etc. Rules (Abs), (App) and (Var-x) are standard. Rule (K) seems somewhat redundant and could be modeled by rules (App) and (Var-x) assuming that constructors are recorded in Γ_{init}. Our intention is that constructors are always fully applied. Rule (Case) deals with case expression. Nothing unusual so far. Next, we consider the GRDT specific rules. In rule (Eq) we are able to change the type of an expression. Note that the side condition $C \supset t_1 = t_2$ holds iff (1) C does not have a unifier, or (2) for any unifier ϕ of C we have

$$(\text{Eq})\ \frac{C,\Gamma \vdash^G e : t \quad C \supset t = t'}{C,\Gamma \vdash^G e : t'} \qquad (\text{App})\ \frac{C,\Gamma \vdash^G e_2 : t_2 \quad C,\Gamma \vdash^G e_1 : t_2 \to t}{C,\Gamma \vdash^G e_1\ e_2 : t} \qquad (\text{Abs})\ \frac{C,\Gamma.x : t_1 \vdash^G e : t_2}{C,\Gamma \vdash^G \lambda x.e : t_1 \to t_2}$$

$$(\text{Var-x})\ \frac{(x : \forall \bar{a}.t) \in \Gamma}{C,\Gamma \vdash^G x : [\bar{t}/\bar{a}]t} \qquad (\text{Case})\ \frac{C,\Gamma \vdash^G e : t_1 \quad C,\Gamma \vdash^G p_i \to e_i : t_1 \to t_2 \quad \text{for } i \in I}{C,\Gamma \vdash^G \mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i\in I} : t_2}$$

$$(\text{K})\ \frac{K : \forall \bar{a},\bar{b}.D \Rightarrow t \to T\ \bar{a} \quad C,\Gamma \vdash^G e : [\bar{t}/\bar{a},\bar{t'}/\bar{b}]t \quad C \supset [\bar{t}/\bar{a},\bar{t'}/\bar{b}]D}{C,\Gamma \vdash^G K\ e : T\ \bar{t}} \qquad (\text{Pat})\ \frac{p : t_1 \vdash^G \forall \bar{b}.(D \,|\, \Gamma_p) \quad \bar{b} \cap fv(C,\Gamma,t_2) = \emptyset \quad C \wedge D, \Gamma \cup \Gamma_p \vdash^G e : t_2}{C,\Gamma \vdash^G p \to e : t_1 \to t_2} \qquad (\text{P-Var})\ \ x : t \vdash^G (True \,|\, \{x : t\})$$

$$(\text{P-Pair})\ \frac{p_1 : t_1 \vdash^G \forall \bar{b_1}.(D_1 \,|\, \Gamma_{p_1}) \quad p_2 : t_2 \vdash^G \forall \bar{b_2}.(D_2 \,|\, \Gamma_{p_2})}{(p_1,p_2) : (t_1,t_2) \vdash^G \forall \bar{b_1},\bar{b_2}.(D_1 \wedge D_2 \,|\, \Gamma_{p_1} \cup \Gamma_{p_1})} \qquad (\text{P-K})\ \frac{K : \forall \bar{a},\bar{b}.D \Rightarrow t \to T\ \bar{a} \quad \bar{b} \cap \bar{a} = \emptyset \quad p : [\bar{t}/\bar{a}]t \vdash^G \forall \bar{b'}.(D' \,|\, \Gamma_p)}{K\ p : T\ \bar{t} \vdash^G \forall \bar{b'},\bar{b}.(D' \wedge [\bar{t}/\bar{a}]D \,|\, \Gamma_p)}$$

Figure 1: GRDT Typing Rules

that $\phi(t_1) = \phi(t_2)$ holds. In rule (Pat) we make use of an auxiliary judgment $p : t \vdash \forall \bar{b}.(D \,|\, \Gamma_p)$ which establishes a relation among pattern p of type t and the binding Γ_p of variables in p. Variables $\bar{b}$ refer to all "existential" variables. Logically, these variables must be considered as universally quantified. Hence, we write $\forall \bar{b}$. The side condition $\bar{b} \cap fv(C,\Gamma,t_2) = \emptyset$ prevents existential variables from escaping. In rule (P-Pair), we assume that there are no name clashes between variables $\overline{b_1}$ and $\overline{b_2}$. Constraint D arises from constructor uses in p. The other rules are standard.

Let's consider the first clause of `f` in Example 1 again. According to rule (Pat), the pattern `I x` provides the additional type assumption $a = Int$ which is used in typing of the body `x+1`. Note that because of this additional assumption, rule (Eq) is able to turn the type of `x` from a to Int. Thus, the expression `x+1` is well typed. Similarly, rule (Eq) also turns the type of `x+1` to a. Hence, the annotation given to `f` is correct. Rule (Eq) has some other surprising consequences.

Example 3 Consider the following variation of Example 1

```
data Erk a = (a=Int) => I a
g :: Erk Bool -> b
g (I x) = x + 'a'
```

We make use of $Bool = Int$ which is equivalent to $False$ to type the body of the clause. Hence, we can derive anything. Hence, `g` has type $Erk\ Bool \to b$ for any b. Note that we only temporarily make use of $False$. The constraint in the final judgment is satisfiable. □

As already observed by Cheney and Hinze [4] such meaningless programs can always be replaced by "undefined". Note that we never ever construct a value of type $Erk\ Bool$. Hence, w.l.o.g. we slightly restrict the set of typable programs and replace logical by constructive entailment. Effectively, we rule out GRDT programs where $False$ occurs in (intermediate) typing judgments. The definition of constructive entailment among type equality is as follows:

$$\frac{t = t' \in C}{C \vdash^{=_c} t = t'} \qquad \frac{C \vdash^{=_c} t_1 = t_2 \quad C \vdash^{=_c} t_2 = t_3}{C \vdash^{=_c} t_1 = t_3}$$

$$\frac{C \vdash^{=_c} t_1 = t_2 \quad C \vdash^{=_c} t_3 = t_4}{C \vdash^{=_c} t_1 \to t_3 = t_2 \to t_4} \qquad \frac{C \vdash^{=_c} t_i = t'_i \quad \text{for } i = 1,...,n}{C \vdash^{=_c} T\ t_1 ... t_n = T\ t'_1 ... t'_n}$$

We obtain the constructive GRDT system $\vdash^{G_c}$ by replacing (Eq) with the following rule.

$$(\text{Eq}_c)\ \frac{C,\Gamma \vdash^{G_c} e : t \quad C \vdash^{=_c} t = t'}{C,\Gamma \vdash^{G_c} e : t'}$$

Note that Example 3 is not typable anymore in the constructive system.

4. TRANSLATING GRDT TO TCET

The main result of this section is that GRDT can be encoded by type classes with existential types (TCET). This will form an important intermediate step in our translation to ET. For this purpose, we introduce a type class $Ct\ a\ b$ to convert a term of type a into a term of type b. In essence, we model directed equality. The following instance declarations implement this idea.

```
class Ct a b where cast :: a->b
instance Ct a a where cast x = x -- (Id)
instance (Ct b1 a1, Ct a2 b2) => Ct (a1->a2) (b1->b2)
   where cast f x = cast (f (cast x))      -- (Arrow)
instance (Ct a1 a2, Ct a2 a3) => Ct a1 a3
   where cast a1 =   cast (cast a1)        -- (Trans)
```

Operationally, the conversion functions performs the identity operation for all monomorphic instances derivable w.r.t. the above rules.

We translate GRDT programs to TCET by replacing each equation $t_1 = t_2$ in a data type definition by $Ct\ t_1\ t_2$ and $Ct\ t_2\ t_1$ Additionally, we apply `cast` to all sub-expressions.

Example 4 Here is the translation of Example 1.

```
data Erk_H a = (Ct a Int, Ct Int a) => I_H a
             | forall b.(Ct a [b], Ct [b] a) => L_H a
f_H :: Erk_H a -> a f_H (I_H x) =
  cast ((cast ((cast (+)) (cast x))) (cast 1))
f_H (L_H x) = cast ((cast tail) (cast x))
```

When typing the second clause we temporarily make use of $Ct\ a\ [b]$ and $Ct\ [b]\ a$. Thus, `cast x` can be given type $[b]$. We make use of instance (Id) to show that `cast tail` has type $[b] \to [b]$. Hence, `(cast tail) (cast x)` has type $[b]$. Hence, `cast ((cast tail) (cast x))` can be given type a. A similar reasoning applies to the first clause where we make use of instance (Arrow). Hence, function `f_H` is type correct. □

The connection between GRDT and TCET becomes obvious when considering their underlying formal systems. A formal description of TCET covering the single-parameter case is given by Läufer [13]. In our own work [23], we recently formalized the general case including multi-parameter type classes which we will make use of in the following.

Briefly, in the TCET system we find now type (multi-paramter) class constraints $TC\ t_1 ... t_n$ instead of equality constraints $t_1 = t_2$. For simplicity, we assume that instance declarations are preprocessed and the relations they describe are translated to logic formulae. We commonly denote these logic formulae by P_p and refer to P_p as the *program theory.* E.g., the instance declarations from above can be described by the following first-order formulae.

$$\forall a.(Ct\ a\ a \leftrightarrow True)$$
$$\forall a_1, a_2, b_1, b_2.(Ct\ (a_1 \to a_2)\ (b_1 \to b_2) \leftrightarrow Ct\ b_1\ a_1 \wedge Ct\ a_2\ b_2)$$
$$\forall a_1, a_3.(Ct\ a_1\ a_3 \leftrightarrow \exists a_2.(Ct\ a_1\ a_2 \wedge Ct\ a_2\ a_3)$$

where $\leftrightarrow$ denotes Boolean equivalence. We refer the interested reader to [24] for more details on the translation of instances to logic formulae.

For each class declaration `class TC a1...an where m::t` we assume a new primitive $m : \forall \bar{a}.TC\ \bar{a} \Rightarrow t$. For simplicity, we restrict ourselves to *monomorphic methods*. That is, we require that $fv(t) \subseteq \bar{a}$. Note that the restriction to monomorphic methods is sufficient for the purpose of translating GRDT to TCET.

The typing rules for TCET are almost the same as those for GRDT in Figure 1. We adopt rules (App), (Abs), (Var-x), (Case), (Pat), (P-Var), (P-Pair) and (P-K) from Figure 1. However, we drop rule (Eq). Furthermore, we adjust rule (K) and introduce a new rule (M) to take care of class methods.

$$\text{(K)}\quad \frac{\begin{array}{c} K : \forall \bar{a}, \bar{b}.D \Rightarrow t \to T\ \bar{a} \\ C, \Gamma \vdash^T e : [\bar{t}/\bar{a}]t \quad P_p \models C \supset [\bar{t}/\bar{a}, \bar{t'}/\bar{b}]D \end{array}}{C, \Gamma \vdash^T K\ e : T\ \bar{t}}$$

$$\text{(M)}\quad \frac{m : \forall \bar{a}.TC\ \bar{a} \Rightarrow t \quad fv(t) \subseteq \bar{a} \quad P_p \models C \supset TC\ \bar{t}}{C, \Gamma \vdash^T m : [\bar{t}/\bar{a}]t}$$

Note that entailment is now defined w.r.t. the program theory. The side condition $P_p \models C \supset [\bar{t}/\bar{a}, \bar{t'}/\bar{b}]D$ denotes that any model satisfying P_p and C also satisfies $[\bar{t}/\bar{a}, \bar{t'}/\bar{b}]D$.

To distinguish the two systems we write $C, \Gamma \vdash^T e : t$ to denote that expression e has type t under constraint C and environment Γ in the TCET system. In case of $True, \Gamma \vdash^T e : t$ we sometimes write $\Gamma \vdash^T e : t$ for short.

We are in the position to define the formal translation from GRDT to TCET. In order to model the constructive entailment relation $\vdash^{=_c}$ among equalities we need to impose some conditions on the program theory.

Definition 1 (Full and Faithful) *We say that the program theory P_p is* full and faithful *w.r.t. constructive equality iff (1) for each n-ary type constructor T there is some appropriate instance such that*

$$P_p \models (Ct\ (T\ a_1 ... a_n)\ (T\ b_1 ... b_n) \wedge Ct\ (T\ b_1 ... b_n)\ (T\ a_1 ... a_n)) \supset (Ct\ a_1\ b_1 \wedge Ct\ b_1\ a_1 \wedge ... Ct\ a_n\ b_n \wedge Ct\ b_n\ a_n)$$

and (2) all monomorphic cast instances are equivalent to the identity. Equality among expressions is defined in terms of a standard denotational semantics, e.g., consider [15].

To turn GRDT typable expressions into TCET typable expressions, we perform a syntactic transformation by applying the `cast` function to each (sub-)expression. We write $e[e']$ to denote a occurrence of e' in e.

Definition 2 (Fully Casted) *Let e be an GRDT expression. We construct a fully casted expression e' out of e by applying cast on every subexpression of e. A single transformation step is defined as $e[e_1] \leadsto e[cast\ e_1]$ where e_1 is syntactically different from $cast\ e_2$ for some expression e_2.*

The transformation of GRDT constructors is simple. Each GRDT constructor

$$K : \forall \bar{a}, \bar{b}.(t_1 = t'_1, ..., t_n = t'_n) \Rightarrow t \to T\ \bar{a}$$

implies a TCET constructor

$$K' : \forall \bar{a}, \bar{b}.(Ct\ t_1\ t'_1, Ct\ t'_1\ t_1, ..., Ct\ t_n\ t'_n, Ct\ t'_n\ t_n) \Rightarrow t \to T\ \bar{a}$$

We can state the following formal connection between GRDT and TCET.

Theorem 1 (GRDT to TCET) *Let e be a GRDT expression and e' be its fully casted version. For each GRDT constructor K we introduce its TCET equivalent K'. Let P_p a full and faithful program theory representing all GRDT type constructors mentioned in e. Then, we have that $True, \Gamma \vdash^{G_c} e : t$ iff $True, \Gamma \vdash^T e' : t$.*

A proof can be found in Appendix B.1.

As already pointed out the restriction to the $\vdash^{G_c}$ system is not onerous. Note that in order to directly translate Example 3 the program theory would need to be strengthened by including additional "improvement" rules such as $P_p \models Ct\ Bool\ Int \supset False$, $P_p \models Ct\ Int\ Bool \supset False$ etc.

The above result is constructive in the sense that we can type check the resulting TCET program if the entire GRDT typing derivation (including $C \vdash^{=_c} t_1 = t_2$ derivations) is available. We can also give a meaning to translated TCET program based on the scheme presented in [24]. However, GHC fails to accept the TCET program because instance declarations are potentially "non-terminating". [1] E.g., consider instance (Trans) from above. When performing context reduction [2] we need to guess the intermediate type when applying instance (Trans). Hence, context-reduction may or may not terminate. Hence, the check whether $C \supset Ct\ t_1\ t_2$ holds where C is a set of Ct assumptions may not terminate. On the other hand, $C' \supset t_1 = t_2$ is decidable assuming that C' is derived from C by turning each $Ct\ t\ t'$ into an equation $t = t'$. We conclude that we further need to refine our transformation method for GRDT. The translation to TCET represents an important intermediate step to achieve a translation to ET which is finally accepted by GHC.

5. TRANSLATING GRDT TO ET

The result from the previous section allows us to assume that GRDT programs have been translated to TCET by fully casting expressions and transforming GRDT constructors into TCET constructors. Hence, it is sufficient to consider the translation from TCET to ET. We establish some sufficient conditions under which we achieve a type-directed translation translation scheme from TCET to ET based on a proof system to construct terms connected to type class constraints $Ct\ t\ t'$.

We start off by describing our proof system. We assume that constraints such as $f : Ct\ a\ b$ carry now a proof term f representing "evidence" for $Ct\ a\ b$. We silently drop f in case proof terms do not matter. We introduce judgments of the form $f : Ct\ a\ b \leftrightarrow F$ to denote that f is the proof term corresponding to $Ct\ a\ b$ under the assumption F where F refers to a (possibly existentially quantified) conjunction of type class constraints. The rules describing the valid judgments are in Figure 2. Note that we write the actual definition of f as part of the premise. Rules (Id), (Var) and (Trans) are straightforward. Rules (Arrow) and (Pair) deal with function and pair types. We assume that the proof rules will be extended accordingly for user-defined types. Rule ($\circ$) allows for the structural composition of proof terms. Rules ($\forall$E) and ($\exists$E) deal with universal and existential quantifiers. In essence, we make the construction rules represented by Ct instance declarations explicit.

Example 5 We give the derivation tree for $f : Ct\ a\ (Int, Bool) \leftrightarrow g_1 : Ct\ a\ (b, c), g_2 : Ct\ b\ Int, g_3 : Ct\ c\ Bool$ in Figure 2. For convenience, we combine rule ($\forall$E) with rules (Id), (Var), (Arrow). We conclude that

```
f x = let g4 (x,y) = (g2 x,g3 y)
      in g4 (g1 x)
```

□

A simple observation of our proof rules shows that the proof system is sound w.r.t. the logical reading of instances declarations.

[1]Indeed, GHC will only accept instance (Trans) once we turn on the "undecidable instances" option.

[2]This is the process of resolving type classes w.r.t. a given set of class and instance declarations.

Lemma 1 (Soundness) *Let P_p be the program theory. Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$ such $f : Ct\ a\ b \leftrightarrow C$ is valid. Then, $P_p \models C \supset Ct\ a\ b$.*

We can also state that proof terms are well-typed.

Definition 3 *Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$. We construct an environment Γ out of C, written as $C \rightsquigarrow \Gamma$, by mapping each $g : Ct\ a\ b \in C$ to $g : a \rightarrow b \in \Gamma$.*

Lemma 2 (Well-Typed) *Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$ and Γ such that $C \rightsquigarrow \Gamma$ and $f : Ct\ a\ b \leftrightarrow C$ is valid. Then $\Gamma \vdash f : a \rightarrow b$.*

Proofs can be found in Appendix B.2. Note that the proof term f is equivalent to the identity assuming f_1,...,f_n are equivalent to the identity as well.

As presented, our proof term construction rules in Figure 2 are still non-terminating (see rule (Trans)). In the upcoming Section 6, we give a decidable procedure to compute $f : Ct\ a\ b \leftrightarrow C$ given $Ct\ a\ b$ and C.

We are in the position to systematically translate TCET to ET. Each TCET constructor $K : \sigma$ is turned into a ET constructor $K' : \sigma'$, written $(K : \sigma) \rightsquigarrow (K' : \sigma')$. We have that $(K : \forall \bar{a}, \bar{b}.D \Rightarrow t \rightarrow T\ \bar{a}) \rightsquigarrow (K' : \forall \bar{a}, \bar{b}.t \rightarrow E\ t_1\ t'_1 \rightarrow ... \rightarrow E\ t_n\ t'_n \rightarrow T\ \bar{a})$ where $D = \{Ct\ t_1\ t'_1, Ct\ t'_1\ t_1, ..., Ct\ t_n\ t'_n, Ct\ t'_n\ t_n\}$. Silently, we assume a fixed order among Ct constraints. Note that the type constructor E is defined in Example 2.

For the translation of expressions we introduce judgments of the form $C, \Gamma \vdash^T e : t \rightsquigarrow e'$ where C holds Ct assumptions, e is a TCET expression and e' is a ET expression. The translation rules can be found in Figure 3. Our main tasks are to resolve `cast` functions (see rule (Reduce)) based on our proof system and to explicitly insert proof terms in constructors (see rule (P-K)). In rule (K), we define $P_p \models C \supset \overline{(g, h)} : [\bar{t}/\bar{a}]D$ iff $g_i : Ct\ t_i\ t'_i \leftrightarrow C$ and $h_i : Ct\ t'_i\ t_i \leftrightarrow C$ for $i = 1, .., n$ where $[\bar{t}/\bar{a}]D = \{Ct\ t_1\ t'_1, Ct\ t'_1\ t_1, ..., Ct\ t_n\ t'_n, Ct\ t'_n\ t_n\}$. Note that $P_p \models C \supset \overline{(g, h)} : [\bar{t}/\bar{a}]D$ implies that $P_p \models C \supset [\bar{t}/\bar{a}]D$ (see Lemma 1). As will see the other direction (which is crucial for completeness) does not hold necessarily.

We can state soundness of our translation scheme given that the TCET program is typable. Note that the ET system is a special instance of TCET. We write $\Gamma \vdash^E e : t$ to denote a judgment in the ET system.

Theorem 2 (TCET to ET Soundness) *Let $True, \Gamma \vdash^T e : t$ and $True, \Gamma \vdash^T e : t \rightsquigarrow e'$. Then $\Gamma \vdash^E e' : t$.*

We also find that e and e' are equivalent assuming the program theory and proof system is full and faithful.

In combination with Theorem 1 we obtain a systematic translation from GRDT to ET. We do rely on full type information for the GRDT program such that our proof term construction method is able to insert the appropriate evidence values.

Proof Term Construction Rules:

$$(\text{Id}) \quad \forall a.\lambda x.x : Ct\ a\ a \leftrightarrow True \qquad (\text{Var}) \quad \forall a, b.f : Ct\ a\ b \leftrightarrow f : Ct\ a\ b$$

$$(\text{Trans}) \quad \frac{f = \lambda x.f_2\ (f_1\ x)}{\forall a_1, a_3.f : Ct\ a_1\ a_3 \leftrightarrow \exists a_2.f_1 : Ct\ a_1\ a_2, f_2 : Ct\ a_2\ a_3}$$

$$(\text{Arrow}) \quad \frac{f = \lambda g.\lambda x.f_2\ (g\ (f_1\ x))}{\forall a_1, a_2, b_1, b_2.f : Ct\ (a_1 \to a_2)\ (b_1 \to b_2) \leftrightarrow f_1 : Ct\ b_1\ a_1, f_2 : Ct\ a_2\ b_2}$$

$$(\text{Pair}) \quad \frac{\begin{array}{c} f = \lambda(x,y).(f_1\ x, f_2\ y) \\ C = \{f_1 : Ct\ a_1\ b_1, f_2 : Ct\ a_2\ b_2\} \end{array}}{\forall a_1, a_2, b_1, b_2.f : Ct\ (a_1, a_2)\ (b_1, b_2) \leftrightarrow C}$$

$$(\circ) \quad \frac{f : Ct\ a\ b \leftrightarrow f_1 : c_1, ..., f_n : c_n \quad f_i : c_i \leftrightarrow F_i \quad F \models F_i \quad \text{for } i = 1, ..., n}{f : Ct\ a\ b \leftrightarrow F}$$

$$(\forall \text{E}) \quad \frac{\forall \bar{a}.f : Ct\ t_1\ t_2 \leftrightarrow F \quad \phi = [\bar{t}/\bar{a}]}{f : Ct\ \phi(t_1)\ \phi(t_2) \leftrightarrow \phi(F)} \qquad (\exists \text{E}) \quad \frac{f : c \leftrightarrow \exists a.F}{f : c \leftrightarrow [t/a]F}$$

Example:

$$(\circ) \quad \frac{\begin{array}{c} (\text{Trans}) \quad \dfrac{f = \lambda x.g_4\ (g_1\ x)}{f : Ct\ a\ (Int, Bool) \leftrightarrow g_1 : Ct\ a\ (b,c), g_4 : Ct\ (b,c)\ (Int, Bool)} \\ (\text{Var}) \quad \begin{array}{l} g_1 : Ct\ a\ (b,c) \leftrightarrow \\ g_1 : Ct\ a\ (b,c) \end{array} \qquad (\text{Pair}) \quad \dfrac{g_4(x,y) = (g_2\ x, g_3\ y)}{\begin{array}{c} g_4 : Ct\ (b,c)\ (Int, Bool) \leftrightarrow \\ g_2 : Ct\ b\ Int, g_3 : Ct\ c\ Bool \end{array}} \end{array}}{f : Ct\ a\ (Int, Bool) \leftrightarrow g_1 : Ct\ a\ (b,c), g_2 : Ct\ b\ Int, g_3 : Ct\ c\ Bool}$$

Figure 2: Proof Term Construction Rules and Example

Note that we do not obtain completeness in general. The problem is that proof terms are not "decomposable" in general. This has already been observed by Chen, Zhu and Xi [2].

Example 6 Consider

```
data Foo a = K
instance Ct a b => Ct (Foo a) (Foo b) where cast K = K
```

We have that $P_p \models g : Ct\ (Foo\ a)\ (Foo\ b) \supset h : Ct\ a\ b$ but $h : Ct\ a\ b \leftrightarrow g : Ct\ (Foo\ a)\ (Foo\ b)$ does not exist. Hence, our translation scheme gets possibly stuck in rules (K) and (Reduce). Note that the instance declaration implies that $Ct\ (Foo\,a)\ (Foo\,b)$ iff $Ct\ a\ b$. The instance context seems somewhat redundant but necessary to ensure that the program theory models fully and faithfully the entailment relation $\vdash^{=_c}$. Clearly, we can build `g` on type `Foo a -> Foo b` given `h` on type `a->b` whereas for the other direction we would need to decompose proof terms which is not possible here. □

The above is not surprising. Similar situations arise for simple type class programs. E.g., we cannot decompose $Eq\ [a]$ into $Eq\ a$ for any a. All what we can do is to identify some sufficient conditions which allow us to extend the rules in Figure 2 faithfully.

Definition 4 (Decomposable Types) *Let T be a n-ary type constructor. We say that T is* decomposable *at position i where $i \in \{1, ..., n\}$ iff a proof term construction rule $f_i : Ct\ a_i\ b_i \leftrightarrow g : Ct\ (T\ a_1...a_n)\ (T\ b_1...b_n), h : Ct\ (T\ b_1...b_n)\ (T\ a_1...a_n)$ exists such that (1) f_i is well-typed under $\{g : T\ a_1...a_n \to T\ b_1...b_n, h : T\ b_1...b_n \to T\ a_1...a_n\}$ and (2) f_i is equivalent to the identity if g and h are equivalent to the identity.*

We say that T is decomposable iff T is decomposable at all positions.

We find that pairs are decomposable.

Example 7 We make use of $\perp : \forall a.a$. Consider

$$(\text{Pair1}\!\downarrow) \quad \frac{g_1 = \lambda x.fst\ (f\ (x, \perp))}{g_1 : Ct\ a_1\ b_1 \leftrightarrow f : Ct\ (a_1, a_2)\ (b_1, b_2)}$$

$$(\text{Pair2}\!\downarrow) \quad \frac{g_2 = \lambda x.snd\ (f\ (\perp, x))}{g_2 : Ct\ a_2\ b_2 \leftrightarrow f : Ct\ (a_1, a_2)\ (b_1, b_2)}$$

□

However, function types seem only to be decomposable in their co-variant position under a non-strict semantics.

$$\text{(Abs)} \quad \frac{C, \Gamma.x : t_1 \vdash^T e : t_2 \leadsto e'}{C, \Gamma \vdash^T \lambda x.e : t_1 \to t_2 \leadsto \lambda x.e'} \qquad \text{(App)} \quad \frac{\begin{array}{c} C, \Gamma \vdash^T e_2 : t_2 \leadsto e_2' \\ C, \Gamma \vdash^T e_1 : t_2 \to t \leadsto e_1' \end{array}}{C, \Gamma \vdash^T e_1\ e_2 : t \leadsto e_2'\ e_1'}$$

$$\text{(Var-x)} \quad \frac{(x : \forall \bar{a}.t) \in \Gamma}{C, \Gamma \vdash^T x : [\bar{t}/\bar{a}]t \leadsto x} \qquad \text{(Reduce)} \quad \frac{D \subseteq C \quad f : Ct\ t_1\ t_2 \leftrightarrow D}{C, \Gamma \vdash^T cast : t_1 \to t_2 \leadsto f}$$

$$\text{(Case)} \quad \frac{\begin{array}{c} C, \Gamma \vdash^T e : t_1 \leadsto e' \\ C, \Gamma \vdash^T p_i \to e_i : t_1 \to t_2 \leadsto p_i' \to e_i' \quad \text{for } i \in I \end{array}}{C, \Gamma \vdash^T \textsf{case}\ e\ \textsf{of}\ [p_i \to e_i]_{i \in I} : t_2 \leadsto \textsf{case}\ e'\ \textsf{of}\ [p_i' \to e_i']_{i \in I}}$$

$$\text{(Pat)} \quad \frac{\begin{array}{c} p : t_1 \vdash \forall \bar{b}.(D \mid \Gamma_p \mid p') \quad \bar{b} \cap fv(C, \Gamma, t_2) = \emptyset \\ C \wedge D, \Gamma \cup \Gamma_p \vdash^T e : t_2 \leadsto e' \end{array}}{C, \Gamma \vdash^T p \to e : t_1 \to t_2 \leadsto p' \to e'}$$

$$\text{(K)} \quad \frac{\begin{array}{c} (K : \forall \bar{a}, \bar{b}.Ct\ t_1\ t_1', Ct\ t_1'\ t_1 ..., Ct\ t_n\ t_n', Ct\ t_n'\ t_n \Rightarrow t \to T\ \bar{a}) \\ (K' : \forall \bar{a}, \bar{b}.t \to E\ t_1\ t_1' \to ... \to E\ t_n\ t_n' \to T\ \bar{a}) \\ C, \Gamma \vdash^T e : [\bar{t}/\bar{a}]t \leadsto e' \\ P_p \models C \supset \overline{(g, h)} : [\bar{t}/\bar{a}](Ct\ t_1\ t_1', Ct\ t_1'\ t_1 ..., Ct\ t_n\ t_n', Ct\ t_n'\ t_n) \end{array}}{C, \Gamma \vdash^T Ke : T\ \bar{t} \leadsto K'\ e'\ E\ (g_1, h_1) ... E\ (g_n, h_n)}$$

$$\text{(P-Var)} \quad x : t \vdash (True \mid \{x : t\} \mid x)$$

$$\text{(P-Pair)} \quad \frac{p_1 : t_1 \vdash \forall \bar{b_1}.(D_1 \mid \Gamma_{p_1} \mid p_1') \quad p_2 : t_2 \vdash \forall \bar{b_2}.(D_2 \mid \Gamma_{p_2} \mid p_2')}{(p_1, p_2) : (t_1, t_2) \vdash \forall \bar{b_1}, \bar{b_2}.(D_1 \wedge D_2 \mid \Gamma_{p_1} \cup \Gamma_{p_1} \mid (p_1', p_2'))}$$

$$\text{(P-K)} \quad \frac{\begin{array}{c} (K : \forall \bar{a}, \bar{b}.Ct\ t_1\ t_1', Ct\ t_1'\ t_1 ..., Ct\ t_n\ t_n', Ct\ t_n'\ t_n \Rightarrow t \to T\ \bar{a}) \\ (K' : \forall \bar{a}, \bar{b}.t \to E\ t_1\ t_1' \to ... E\ t_n\ t_n' \to T\ \bar{a}) \\ \bar{b} \cap \bar{a} = \emptyset \quad p : [\bar{t}/\bar{a}]t \vdash \forall \bar{b'}.(D' \mid \Gamma_p \mid p') \quad g_1, h_1, ..., g_n, h_n \text{ fresh} \\ D'' = \{D', g_1 : Ct\ t_1\ t_1', h_1 : Ct\ t_1'\ t_1 ..., g_n : Ct\ t_n\ t_n', h_n : Ct\ t_n'\ t_n\} \end{array}}{K\ p : T\ \bar{t} \vdash \forall \bar{b'}, \bar{b}.(D'' \mid \Gamma_p \mid K'\ p'\ E\ (g_1, h_1) ... E\ (g_n, h_n))}$$

Figure 3: Type-Directed Translation

Example 8

$$\text{(Arrow}\downarrow\text{)} \quad \frac{g = \lambda x.(f\ (\lambda y.x))\ \bot}{g : Ct\ a_2\ b_2 \leftrightarrow f : Ct\ (a_1 \to a_2)\ (b_1 \to b_2)}$$

Note that g is the identity under a non-strict semantics. However, it seems that $h : Ct\ b_1\ a_1 \leftrightarrow f : Ct\ (a_1 \to a_2)\ (b_1 \to b_2)$ does not exist. □

Example 9 The *Either* data type is decomposable:

```
data Either a b = Left a | Right b
```

The construction rules are as follow:

$$\text{(EitherL}\downarrow\text{)} \quad \frac{g = \lambda x.project_L\ (f\ (inject_L\ x))}{g : Ct\ a_1\ b_1 \leftrightarrow f : Ct\ (Either\ a_1\ a_2)\ (Either\ b_1\ b_2)}$$

$$\text{(EitherR}\downarrow\text{)} \quad \frac{g = \lambda x.project_R\ (f\ (inject_R\ x))}{g : Ct\ a_2\ b_2 \leftrightarrow f : Ct\ (Either\ a_1\ a_2)\ (Either\ b_1\ b_2)}$$

where

```
inject_L x = Left x
project_L (left x) = x
inject_R x = Right x
project_R (Right x) = x
```

Note that the decomposition conditions (Definition 4) are satisfied. Consider the (EitherL↓) case. Expressions are well-typed. Assume f is the identity. Then, $f\ (inject_L\ x)$ must yield $L\ x$. Hence, application of $project_L$ is safe. Hence, g is the identity. A similar reasoning applies (EitherR↓). □

Decomposable types ensure that our proof term construction system is not only sound but also complete.

Lemma 3 (Decomposition) *Let P_p be a full and faithful program theory, $Ct\ t_1\ t_2$ a constraint and $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$ such that $P_p \models C \supset Ct\ t_1\ t_2$ and all types appearing in constraints are decomposable. Then, $f : Ct\ t_1\ t_2 \leftrightarrow C$ for some proof term f.*

The proof is straightforward and proceeds by induction over $P_p \models C \supset Ct\ t_1\ t_2$.

We are able to state completeness of our translation from TCET to ET given that the types appearing in assumption constraints are decomposable. By assumption constraints we refer to constraints D in rule (Pat).

Theorem 3 (TCET to ET Completeness) *Let $True, \Gamma \vdash^T e : t$ and all types appearing in assumption constraints in intermediate derivations are decomposable. Then $True, \Gamma \vdash^T e : t \leadsto e'$ for some e'.*

6. DECIDABLE PROOF TERM CONSTRUCTION METHOD

We introduce a method to decide $f : Ct\ t_1\ t_2 \leftrightarrow C$ (see Figure 2). That is, given C and $Ct\ t_1\ t_2$ construct a derivation for some f. The main challenge is to find a decidable representation for rule (Trans). In the above statement, C contains the set of Ct assumptions whereas $Ct\ t_1\ t_2$ refers to a use site (see rule (Reduce) in Figure 3). In order to distinguish between Ct uses and assumptions we write $CtM\ t_1\ t_2$ to refer to a use of Ct. Our task is to construct CtM uses out of a given set of Ct assumptions. Note that Ct constraints can be viewed as directed edges. Hence, the successful construction of a CtM use is equivalent to finding a path in the graph of Ct edges. However, we do not rely our method on graph algorithms. We would like our method to work even under some additional side conditions such as $CtM\ t_1\ t_2, CtM\ t_3\ t_4, t_2 = t_4 \rightarrow a$. That is, construct $CtM\ t_1\ t_2$ and $CtM\ t_3\ t_4$ out of some assumption set C under the side condition that $t_2 = t_4 \rightarrow a$ for some a. Therefore, we view proof term construction as constraint solving where we rewrite constraint stores until all CtMs have been resolved.

The formal development is as follows. We assume that CtM uses are attached to "locations". The idea is that $i : CtM\ a\ b$ refers to some program text $\mathtt{cast}_i$ where `cast` is used at type $a \rightarrow b$ and i refers to the location (e.g., position in the abstract syntax tree). As before, we write $f : Ct\ a\ b$ to refer to the proof term f associated to a $Ct\ a\ b$ assumption.

We employ Constraint Handling Rules (CHRs) [5] to construct CtM uses out of Ct assumptions. CHRs are a rule-based language for specifying transformations among constraints. A CHR *simplification* rule (R) $\bar{c} \Longleftrightarrow \bar{d}$ states that if we find a constraint matching the lhs of a rule we replace this constraint by the rhs. We assume that c_is refer to type class constraints and d_is refer to either type class constraints or equations. Formally, we write $C \rightarrowtail_R C - \bar{c}', \phi(\bar{d})$ where $\bar{c} \in C$ such that $\phi(\bar{c}) = \bar{c}'$ for some substitution ϕ. Silently, we assume the variables in CHRs are renamed before rule application.

A CHR *propagation* rule (R) $\bar{c} \Longleftrightarrow \bar{d}$ states that if we find a constraint matching the lhs of a rule we add the rhs to the store. Formally, we write $C \rightarrowtail_R C, \phi(\bar{d})$ where $\bar{c} \in C$ such that $\phi(\bar{c}) = \bar{c}'$. CHRs also have a logical reading which is not relevant here.

The CHR-based representation of the proof term construction rules can be found in Figure 4. Note that each CHR simplification rule also introduces a transformation rule among expressions written $e \leadsto e'$. We write $C \rightarrowtail^* D'$ to denote an n number of application of CHRs starting with the initial store C yielding store D'. We write $e \leadsto^* e'$ to denote a reduction sequence among expressions.

Proof rules (Arrow) and (Pair) from Figure 2 can be straightforwardly encoded in terms of CHRs. Note that rule (Trans) from Figure 2 has been split into rules (Trans1) and (Id). Our idea is to incrementally build CtM uses out of Ct assumptions. A naive CHR-translation of transitivity such as

$$\begin{array}{lccc} \text{(Trans)} & i : CtM\ a'\ b' & \Longleftrightarrow & j : CtM\ a'\ b, k : CtM\ b\ b' \\ & \mathtt{castm}_i & \leadsto & \mathtt{castm}_k \circ \mathtt{castm}_j \end{array}$$

leads to problems because we need to guess b. In CHR terminology, the above CHR is not range-restricted. We say a CHR is *range-restricted* iff grounding the lhs grounds the rhs. Note that there is no rule (Var). The same effect can be achieved by rule (Trans1) in combination with rule (Id).

Example 10 Here is a sample derivation. We underline constraints involved in rule applications and silently perform equivalence transformations, replacing equals by equals. For brevity, we leave out `castm` transformations.

$$\begin{array}{ll} & \underline{g_1 : Ct\ a\ (b,c)}, g_2 : Ct\ b\ Int, g_3 : Ct\ c\ Bool, \\ & \underline{i : CtM\ a\ (Int, Bool)} \\ \rightarrowtail_{Trans1} & \underline{g_1 : Ct\ a\ (b,c), g_2 : Ct\ b\ Int, g_3 : Ct\ c\ Bool,} \\ & \underline{j : CtM\ (b,c)\ (Int, Bool)} \\ \rightarrowtail_{Pair} & g_1 : Ct\ a\ (b,c), \underline{g_2 : Ct\ b\ Int}, g_3 : Ct\ c\ Bool, \\ & \underline{k : CtM\ b\ Int}, l : CtM\ c\ Bool \\ \rightarrowtail_{Trans1} & g_1 : Ct\ a\ (b,c), g_2 : Ct\ b\ Int, \underline{g_3 : Ct\ c\ Bool}, \\ & m : CtM\ Int\ Int, \underline{l : CtM\ c\ Bool} \\ \rightarrowtail_{Trans1} & g_1 : Ct\ a\ (b,c), g_2 : Ct\ b\ Int, g_3 : Ct\ c\ Bool, \\ & m : CtM\ Int\ Int, n : CtM\ Bool\ Bool \\ \rightarrowtail^*_{Id} & g_1 : Ct\ a\ (b,c), g_2 : Ct\ b\ Int, g_3 : Ct\ c\ Bool \end{array}$$

In the above derivation, $\rightarrowtail^*$ represents n step derivation. □

There is also another set of rules which exclusively manipulates Ct assumptions. In rule (Trans↓) we make use of a CHR propagation rule to build the closure of all available Ct assumptions. Note that we silently avoid to apply propagation rules twice on the same constraints (to avoid infinite propagation). Note that for each "decomposition" rule we introduce a propagation rule. The CHR representation of the rules from Example 7 and 8 can be found in Figure 4.

It should be clear now that simplification rules incrementally resolve CtM uses whereas propagation rules build the closure of all available Ct assumptions. The following example stresses the importance of propagation rules.

CtM Simplification Rules:

$$
\begin{array}{llcl}
\text{(Id)} & i : CtM\ a\ b & \Longleftrightarrow & a = b \\
 & \mathtt{castm}_i & \rightsquigarrow & \lambda x.x \\
\text{(Trans1)} & g : Ct\ a\ b, i : CtM\ a'\ b' & \Longleftrightarrow & g : Ct\ a\ b, a = a', j : CtM\ b\ b' \\
 & \mathtt{castm}_i & \rightsquigarrow & \mathtt{castm}_j \circ g \\
\text{(Arrow)} & i : CtM\ (a_1 \to a_2)\ (b_1 \to b_2) & \Longleftrightarrow & i_1 : CtM\ b_1\ a_1, i_2 : CtM\ a_2\ b_2 \\
 & \mathtt{castm}_i & \rightsquigarrow & \lambda g.\lambda x.\mathtt{castm}_{i_2}\ (g\ (\mathtt{castm}_{i_1}\ x)) \\
\text{(Pair)} & i : CtM\ (a_1, a_2)\ (b_1, b_2) & \Longleftrightarrow & i_1 : CtM\ a_1\ b_1, i_2 : CtM\ a_2\ b_2 \\
 & \mathtt{castm}_i & \rightsquigarrow & \lambda(x, y).((\mathtt{castm}_{i_1}\ x), (\mathtt{castm}_{i_2}\ y))
\end{array}
$$

Ct Propagation Rules:

$$
\begin{array}{llcl}
\text{(Trans}\downarrow\text{)} & g : Ct\ a\ b, h : Ct\ b\ c & \Longrightarrow & h \circ g : Ct\ a\ c \\
\text{(Pair1}\downarrow\text{)} & f : Ct\ (a_1, a_2)\ (b_1, b_2) & \Longrightarrow & (\lambda x.fst\ (f\ (x, \bot))) : Ct\ a_1\ b_1 \\
\text{(Pair2}\downarrow\text{)} & f : Ct\ (a_1, a_2)\ (b_1, b_2) & \Longrightarrow & (\lambda x.snd\ (f\ (\bot, x))) : Ct\ a_2\ b_2 \\
\text{(Arrow}\downarrow\text{)} & f : Ct\ (a_1 \to a_2)\ (b_1 \to b_2) & \Longrightarrow & (\lambda x.(f\ (\lambda y.x))\ \bot) : Ct\ a_2\ b_2
\end{array}
$$

Figure 4: CHR-based Proof Term Construction

Example 11 Consider

$$
\begin{array}{ll}
 & \underline{g : Ct\ (b \to c)\ a}, \underline{h : Ct\ a\ (b \to d)}, i : CtM\ c\ d \\
\rightarrowtail_{Trans\downarrow} & \underline{g : Ct\ (b \to c)\ a, h : Ct\ a\ (b \to d)}, \\
 & \underline{(h \circ g) : Ct\ (b \to c)\ (b \to d)}, i : CtM\ c\ d \\
\rightarrowtail_{Arrow\downarrow} & g : Ct\ (b \to c)\ a, h : Ct\ a\ (b \to d), \\
 & (h \circ g) : Ct\ (b \to c)\ (b \to d), \\
 & (\lambda x.((h \circ g)\ (\lambda y.x))\ \bot) : Ct\ c\ d, i : CtM\ c\ d \\
\rightarrowtail^* & g : Ct\ (b \to c)\ a, h : Ct\ a\ (b \to d), \\
 & (h \circ g) : Ct\ (b \to c)\ (b \to d), \\
 & (\lambda x.((h \circ g)\ (\lambda y.x))\ \bot) : Ct\ c\ d
\end{array}
$$

Note that we can only apply (Arrow↓) after we have applied (Trans↓). □

Another important observation is that CHRs are "indeterministic".

Example 12 Recall Example 10. We find the following alternative derivation.

$$
\begin{array}{ll}
 & g_1 : Ct\ a\ (b, c), g_2 : Ct\ b\ Int, g_3 : Ct\ c\ Bool, \\
 & i : CtM\ a\ (Int, Bool) \\
\rightarrowtail^* & g_1 : Ct\ a\ (b, c), g_2 : Ct\ b\ Int, g_3 : Ct\ c\ Bool, \\
 & b = Int, c = Bool
\end{array}
$$

Note that the final stores differ. Indeed, CHRs are non-confluent. □

We say a set of CHRs is *confluent* iff any sequence of derivation steps on the same initial store leads to the same (logically equivalent) final store. In Figure 4 rules (Id) and (Trans1) overlap and therefore we might discover derivations with same initial store but different final stores.

However, we rule out derivations which yield "bad" final stores. Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$ and $i : CtM\ a\ b, C \rightarrowtail^* D'$. We say that the CHR derivation is *good* iff C and D' are *logically* equivalent, i.e., $\models C \leftrightarrow \exists fv(D') - fv(C).D'$. That is, we rule out derivations yielding stores with unresolved CtM uses, $False$ and further instantiated Ct assumptions. Note that the derivation in Example 12 is bad because the Ct assumptions have been further instantiated in the final store.

We can state that our CHR-based method in Figure 4 is sound w.r.t. the system described in Figure 2. That is, each good derivation implies a valid proof. We can also guarantee to find a good derivation if a proof exists. Furthermore, any good derivation yields equivalent expressions.

Lemma 4 (Sound CHR Construction) *Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$ and $i : CtM\ a\ b, C \rightarrowtail^* D'$ and $castm_i \rightsquigarrow^* e$ such that the CHR derivation is good. Then, $f : Ct\ a\ b \leftrightarrow C$ such that f and e are equivalent.*

Lemma 5 (Complete CHR Construction) *Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$ such that $f : Ct\ a\ b \leftrightarrow C$. Then, $i : CtM\ a\ b, C \rightarrowtail^* C$ such that $castm_i \rightsquigarrow^* e$ and f and e are equivalent.*

Lemma 6 (Sound Term Construction) *Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$, $i : CtM\ a\ b, C \rightarrowtail^* D_1$ and $castm_i \rightsquigarrow^* e_1$ and $i : CtM\ a\ b, C \rightarrowtail^* D_2$ and $castm_i \rightsquigarrow^* e_2$ such that both CHR derivations are good. Then, e_1 and e_2 are equivalent.*

Proofs can be found in Appendix B.5

Note that in order to find a good derivation we might need to back track. See Examples 12 and 10. To obtain a decidable proof method we yet need to rule out certain CHR derivations. E.g., consider

$$
\begin{array}{ll}
 & \underline{g : Ct\ a\ b}, h : Ct\ b\ a, \underline{i : CtM\ a\ b} \\
\rightarrowtail_{Trans1} & g : Ct\ a\ b, \underline{h : Ct\ b\ a}, \underline{j : CtM\ b\ b} \\
\rightarrowtail_{Trans1} & g : Ct\ a\ b, h : Ct\ b\ a, k : CtM\ a\ b \\
 & \ldots
\end{array}
$$

Fortunately, we are able to rule out such non-terminating derivations by imposing stronger restrictions on good derivations. The crucial point is that we disallow "cyclic" Ct assumptions of the form $g : Ct\ a\ (a, b)$. Such assumptions must result from invalid GRDT definitions which we generally rule out.

Lemma 7 *We can impose a complete termination condition on good derivations.*

Details are in Appendix B.6.

We conclude that we obtain a decidable CHR-based proof term construction method. Our method is exponential in the worst-case. However, we believe that such cases will rarely appear in practice. An advantage of our method is that we can perform proof term construction under side conditions. This feature allows us to integrate our method with a general solving method for constructing typing derivations. Details are discussed in the next section.

7. COMBING PROOF TERM CONSTRUCTION AND BUILDING TYPING DERIVATIONS

Our current translation method assumes full type annotations for the GRDT program. Type inference for GRDT is a challenging problem. However, it is mostly sufficient to provide annotations for function definitions only and omit type annotations for sub-expressions. In [23], we introduced a general type inference method for type classes with existential types. The idea is to generate "implication" constraints out of the program text. Solving of these constraints allows us to construct a typing derivation. The solving procedure for implication constraints is phrased as an extension to CHR solving. Hence, we can easily combine the inference method introduced in [23] with our CHR-based proof term construction method. Due to space limitations, we explain the approach by example only.

Consider the following TCET program from Example 4. For simplicity, we only consider one clause.

```
data Erk_H a = forall b.(Ct a [b], Ct [b] a) => L_H a
f_H ::  Erk_H a -> a
f_H (L_H x) = cast ((cast tail) (cast x))
```

In a first step, we translate data types and patterns according to Figure 3 and replace all occurrences of `cast` in the program text by `castm` where each `castm` occurrences are attached to distinct locations.

```
data Erk_H' a = forall b.L_H' a (E a [b])
f_H ::  Erk_H' a -> a
f_H (L_H' x (E (g,h))) = castm1 ((castm2 tail) (castm3 x))
```

According to [23], we generate the following "implication" constraint out of the above program text.

$$\begin{array}{l} t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, \\ \quad (g : Ct\ a\ [b], h : Ct\ [b]\ a \supset \ (1 : CtM\ a_1\ b_1, b_1 = a, \\ \qquad\qquad\qquad\qquad\qquad\quad 2 : CtM\ a_2\ b_2, a_2 = [a'_2] \rightarrow [a'_2], \\ \qquad\qquad\qquad\qquad\qquad\quad 3 : CtM\ a_3\ b_3, a_3 = a, \\ \qquad\qquad\qquad\qquad\qquad\qquad b_2 = b_3 \rightarrow a_1)) \end{array} \tag{1}$$

Annotation `f_H::Erk_H a->a` implies f_H::$\forall a.Erk\_H\ a \rightarrow a$. Hence, we substitute a by the skolem constructor Sk_1. Similarly, we substitute b by $Sk_2\ t$. Each $\texttt{castm}_i$ expression gives rise to $i : CtM\ a\ b$ where $\texttt{castm}_i :: a \rightarrow b$. To each Ct assumption we attach proof terms (see rule (P-K)). We make use of the TCET representation of GRDT but connect the constraints to ET proof terms. The interesting bit is the use of Boolean implication $\supset$ to state that under the Ct assumptions we can derive the CtM uses.

The constraint in (1) represents all possible typing derivations. We simply solve this constraint by applying CHRs defined in Figure 4 until all CtM uses have been resolved. Thus, all locations in the function body referring to proof terms are defined in terms of proof terms attached to Ct assumptions. In general, we solve $C_0, (D \supset C)$ by running $C_0, D \rightarrowtail^* D'$ and $C_0, D, C \rightarrowtail^* C'$ and check that D' and C' are logically equivalent (modulo variables in the initial store). We refer the interested reader to [23] for more details.

For the above constraint (1) we proceed as follows. We find that $t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, g : Ct\ a\ [b], h : Ct\ [b]\ a$ (2) is immediately final.Consider,

$$\begin{array}{ll} & t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, g : Ct\ a\ [b], \\ & h : Ct\ [b]\ a, 1 : CtM\ a_1\ b_1, b_1 = a, \\ & 2 : CtM\ a_2\ b_2, a_2 = [a'_2] \rightarrow [a'_2], \\ & 3 : CtM\ a_3\ b_3, a_3 = a, b_2 = b_3 \rightarrow a_1 \\ & \texttt{f\_H' (L\_H' x (E (g,h))) =} \\ & \quad \texttt{castm}_1\ \texttt{((castm}_2\ \texttt{tail) (castm}_3\ \texttt{x))} \\[1em] \leftrightarrow & t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, \underline{g : Ct\ a\ [b]}, \\ & h : Ct\ [b]\ a, b_1 = a, a_2 = [a'_2] \rightarrow [a'_2], a_3 = a, \\ & b_2 = b_3 \rightarrow a_1, 1 : CtM\ a_1\ a, \\ & 2 : CtM\ ([a'_2] \rightarrow [a'_2])\ (b_3 \rightarrow a_1), \underline{3 : CtM\ a\ b_3} \\ \rightsquigarrow & \texttt{f\_H (L\_H x (E (g,h))) =} \\ & \quad \texttt{castm}_1\ \texttt{((castm}_2\ \texttt{tail) (castm}_3\ \texttt{x))} \\[1em] \rightarrowtail_{Trans1} & t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, g : Ct\ a\ [b], \\ & \underline{h : Ct\ [b]\ a}, b_1 = a, a_2 = [a'_2] \rightarrow [a'_2], \\ & a_3 = a, b_2 = b_3 \rightarrow a_1, \underline{1 : CtM\ a_1\ a}, \\ & 2 : CtM\ ([a'_2] \rightarrow [a'_2])\ (b_3 \rightarrow a_1), 4 : CtM\ [b]\ b_3 \\ \rightsquigarrow & \texttt{f\_H' (L\_H' x (E (g,h))) =} \\ & \quad \texttt{let castm}_3\ \texttt{= castm}_4 \circ \texttt{g} \\ & \quad \texttt{in}\ \ \texttt{castm}_1\ \texttt{((castm}_2\ \texttt{tail) (castm}_3\ \texttt{x))} \\[1em] \rightarrowtail_{Trans1} & t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, g : Ct\ a\ [b], \\ & h : Ct\ [b]\ a, b_1 = a, a_2 = [a'_2] \rightarrow [a'_2], \\ & a_3 = a, b_2 = b_3 \rightarrow a_1, a_1 = [b], \\ & \underline{5 : CtM\ a\ a, 2 : CtM\ ([a'_2] \rightarrow [a'_2])\ (b_3 \rightarrow [b]),} \\ & \underline{4 : CtM\ [b]\ b_3} \\ \rightsquigarrow & \texttt{f\_H' (L\_H' x (E (g,h))) =} \\ & \quad \texttt{let castm}_3\ \texttt{= castm}_4 \circ \texttt{g} \\ & \qquad\ \ \texttt{castm}_1\ \texttt{= castm}_5 \circ \texttt{h} \\ & \quad \texttt{in}\ \ \texttt{castm}_1\ \texttt{((castm}_2\ \texttt{tail) (castm}_3\ \texttt{x))} \\[1em] \rightarrowtail_{Id*} & t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, g : Ct\ a\ [b], \\ & h : Ct\ [b]\ a, \qquad (3) \\ & b_1 = a, a_2 = [a'_2] \rightarrow [a'_2], a_3 = a, b_2 = b_3 \rightarrow a_1, \\ & a_1 = [b], ([a'_2] \rightarrow [a'_2]) = (b_3 \rightarrow [b]), [b] = b_3 \\ \rightsquigarrow & \texttt{f\_H' (L\_H' x (E (g,h))) =} \\ & \quad \texttt{let castm}_3\ \texttt{= castm}_4 \circ \texttt{g} \\ & \qquad\ \ \texttt{castm}_1\ \texttt{= castm}_5 \circ \texttt{h} \\ & \qquad\ \ \texttt{castm}_2\ \texttt{x = x} \\ & \qquad\ \ \texttt{castm}_4\ \texttt{x = x} \\ & \qquad\ \ \texttt{castm}_5\ \texttt{x = x} \\ & \quad \texttt{in}\ \ \texttt{castm}_1\ \texttt{((castm}_2\ \texttt{tail) (castm}_3\ \texttt{x))} \end{array}$$

Note that we simultaneously transform constraints and program text. Constraints involved in rule applications are underlined. Silently, we extend $e' \rightsquigarrow e''$ to $e[e'] \rightsquigarrow e[e'']$ where $e[\cdot]$ denotes an expression with a hole. For clarity, we use let definitions instead of textually replacing expressions. Note that final constraints (2) and (3) are logically equivalent.

Hence, the translation is successful. Note that the final program text for the second derivation can be simplified to the second clause in Example 2. We note that several other derivations are possible. E.g., consider the following where we apply rule (Id) instead of (Trans1).

$$
\begin{array}{ll}
 & t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, g : Ct\ a\ [b], \\
 & h : Ct\ [b]\ a, 1 : CtM\ a_1\ b_1, b_1 = a, \\
 & 2 : CtM\ a_2\ b_2, a_2 = [a_2'] \rightarrow [a_2'], \\
 & 3 : CtM\ a_3\ b_3, a_3 = a, b_2 = b_3 \rightarrow a_1 \\
\leftrightarrow & t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, g : Ct\ a\ [b], \\
 & h : Ct\ [b]\ a, b_1 = a, a_2 = [a_2'] \rightarrow [a_2'], a_3 = a, \\
 & b_2 = b_3 \rightarrow a_1, 1 : CtM\ a_1\ a, \\
 & 2 : CtM\ ([a_2'] \rightarrow [a_2'])\ (b_3 \rightarrow a_1), \underline{3 : CtM\ a\ b_3} \\
\rightarrowtail_{Id} & t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, g : Ct\ a\ [b], \\
 & h : Ct\ [b]\ a, b_1 = a, a_2 = [a_2'] \rightarrow [a_2'], \\
 & a_3 = a, b_2 = b_3 \rightarrow a_1, a = b_3, 1 : CtM\ a_1\ a, \\
 & \underline{2 : CtM\ ([a_2'] \rightarrow [a_2'])\ (a \rightarrow a_1)} \\
\rightarrowtail_{Id} & t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2\ a, g : Ct\ a\ [b], \\
 & h : Ct\ [b]\ a, b_1 = a, a_2 = [a_2'] \rightarrow [a_2'], a_3 = a, \\
 & b_2 = b_3 \rightarrow a_1, a = b_3, ([a_2'] \rightarrow [a_2']) = (a \rightarrow a_1) \\
 & 1 : CtM\ a_1\ a \\
\leftrightarrow & False
\end{array}
$$

Note that skolem variable Sk_1 is unified with $[a_2']$ which immediately yields failure. That is, we obtain a "bad" final store (see Appendix B.6 for details). However, there might be other derivations which yield "good" final stores. Each of them corresponds to a valid solution and all of them are equivalent (see Lemma 6). The following is another possible translation of Example 2.

```
f_H' (L_H' x (E (g,h))) =
    let castm2 g x = castm5 (g (castm4 x))
        castm4 = g
        castm5 = h
        castm1 x = x
        castm3 x = x
    in castm1 ((castm2 tail) (castm3 x))
```

8. RELATED WORK

Our systematic translation method is inspired by the work by Baars and Swierstra [1], Chen, Zhu and Xi [2], Hinze and Cheney [3]. These works showed by example how to express GRDT-style behavior by representing type equalities by Haskell terms and insert appropriate conversion functions into the program text. We note that none of these works considers a systematic translation scheme.

Note that in [1, 3, 16] equality is represented in terms of the following definition.

```
newtype EQ a b = EQ (forall f. f a->f b)
```

The above encodes Leibnitz' law which states that if a and b are equivalent then we may substitute one for the other in any context. By construction this ensures that the only inhabitant of `EQ a b` is the identity (excluding non-terminating functions which might break this property). Our representation of equality makes it necessary to postulate that all values attached to monomorphic instances of `E t t` represent the identity to ensure preservation of the semantics of programs (see Definition 1). On the other hand, the `EQ` representation faces problems when trying to manipulate proof terms. E.g., there are situations where we need to "decompose" a value of type `EQ (a,b) (c,d)` into a value of type `EQ a c` which is impossible based on the above definition. Example 6 shows that our representation of type equality shares the same problem. However, we believe that our representation is more likely to be decomposable.

Weirich [27] also considered a type class encoding based on single-parameter type classes. Our use of multi-parameter type classes in combination with extential types appears to be novel and more natural to mimic GRDT-style behavior.

Kiselyov [11] suggests an alternative type class encoding of GRDT. The gist of his idea is to turn each (value) pattern clause into an (type class) instance declaration. We believe that in addition to the already "problematic" instance declaration for transitivity such an encoding scheme may create further potentially non-terminating instances. We are not aware of any formal results which match the results stated in this paper.

Pottier and Gauthier [17] give a type-preserving defunctionalization of polymorphic programs to System F extended with GRDT. Their formal results (proofs of Lemmas 4.1 and 4.2 in [17]) let us conjecture that resulting GRDT programs can be translated to ET based on our translation method.

Our proof term construction method can be seen as a refined version of the type-directed evidence-translation scheme [7] for Haskell. We could achieve a decidable construction for a seemingly non-terminating set of instances. There are some connections to methods for finding paths in graphs and "ask" constraints which appear in the context of constraint-logic programming [9]. We yet need to work out the exact details.

9. CONCLUSION

The primary goal of our work was to concisely study and relate the concepts of guarded recursive data types (GRDT), existential types (ET) and type classes (TCET). We could achieve this goal by giving for the first time a systematic translation method from GRDT to ET (Section 5) based on an intermediate translation to TCET (Section 4). For the translation method to be complete we require that types appearing in assumption constraints must be decomposable (Definition 4). We also assume full GRDT type information but are able to construct ET expressions automatically based on a novel CHR-based proof term construction method (Section 6). We can even combine our method with an independently developed type inference scheme for GRDT (Section 7). Hence, we obtain a fully automatic tool to translate GRDT to ET where the final program is accepted by GHC. In our experience, the decomposition condition which is crucial for translation is met by all GRDT examples found in the literature. A comprehensive list of examples can be found under [3]

`http://www.comp.nus.edu.sg/~wangmeng/trans-grdt`

An issue we yet need to investigate is how expensive proof term manipulations are in practice. Note that conversion functions represent the identity, however, we may have to

[3] Examples are also part of the technical report version [25].

repeatedly apply such functions to elements of lists etc. A "smart" compiler may be able to avoid such redundant computations (either statically or dynamically). In this context, we would like to mention that GRDT have been recently added to Haskell. Implementations are available in the latest release of GHC [6] and Chameleon [26] (experimental version of Haskell). In case of GHC, the Core back-end has been extended with GRDT as a primitive feature. Clearly, we expect "native" GRDT code to run faster than "source-to-source translated" GRDT code. However, the advantage of our work is that we could identify a large class of GRDT programs which can be implemented by a source-to-source translation. Thus, our work offers a light-weight approach to write GRDT-style programs based on some existing language features.

Our proof term construction method is of independent interest and my prove to be useful to advance the state of art in type-directed translations for languages such as Haskell. This is another interesting avenue which we plan to explore in the future.

Acknowledgements

We thank Gregory Duck, Simon Peyton-Jones and Jeremy Wazny for their comments. In particular, we would like to thank James Cheney and Oleg Kiselyov for fruitful discussions.

10. REFERENCES

[1] A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *Proc. of ICF'02*, pages 157–166. ACM Press, 2002.

[2] C. Chen, D. Zhu, and H. Xi. Implementing cut elimination: A case study of simulating dependent types in Haskell. In *Proc. of PADL'04*, volume 3057 of *LNCS*, pages 239–254. Springer-Verlag, 2004.

[3] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. of Haskell Workshop'02*, pages 90–104. ACM Press, 2002.

[4] J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[5] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

[6] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[7] C. V. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. In *ESOP'94*, volume 788 of *LNCS*, pages 241–256. Springer-Verlag, April 1994.

[8] Haskell 98 language report. http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/.

[9] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19(20):503–581, 1994.

[10] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types, 2004. Submitted to POPL'05.

[11] O. Kiselyov. Typed lambda-expressions without gadts. http://www.haskell.org//pipermail/haskell-cafe/2005-January/008212.html, 2005. Haskell-Cafe Mailing List.

[12] J. Lassez, M. Maher, and K. Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.

[13] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, 1996.

[14] K. Läufer and M. Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 78–91, 1992.

[15] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[16] E. Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health & Science University, OGI School of Science & Engineering, September 2004.

[17] F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *Proc. of POPL'04*, pages 89–98. ACM Press, January 2004.

[18] Franois Pottier and Yann Rgis-Gianas. Towards efficient, typed LR parsers. Draft paper, September 2004.

[19] T. Sheard and E. Pasalic. Meta-programming with built-in type equality. In *Fourth International Workshop on Logical Frameworks and Meta-Languages*, 2004.

[20] Tim Sheard. Languages of the future. *SIGPLAN Not.*, 39(10):116–119, 2004.

[21] J.R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[22] V. Simonet and F. Pottier. Constraint-based type inference with guarded algebraic data types. Submitted to *ACM Transactions on Programming Languages and Systems*, June 2004.

[23] P. J. Stuckey and M. Sulzmann. A unifying inference framework for Hindley/Milner with extensions. http://www.comp.nus.edu.sg/˜ sulzmann, 2004.

[24] P.J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems*, 2004. To appear.

[25] M. Sulzmann and M. Wang. A systematic translation of guarded recursive data types to existential types. Technical Report TR22/04, The National University of Singapore, 2004.

[26] M. Sulzmann and J. Wazny. Chameleon. http://www.comp.nus.edu.sg/˜ sulzmann/chameleon.

[27] S. Weirich. Type-safe cast: (functional pearl). In *Proc. of ICFP'00*, pages 58–67. ACM Press, 2000.

[28] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proc. of POPL'03*, pages 224–235. ACM Press, 2003.

APPENDIX

A. SEMANTICS OF EXPRESSIONS

We follow the ideal semantics of MacQueen, Plotkin and Sethi [15]. The meaning of a term is a value in the CPO $\mathcal{V}$, where $\mathcal{V}$ contains all continuous functions from $\mathcal{V}$ to $\mathcal{V}$ and an error element $\mathbf{W}$, usually pronounced "wrong". Depending on the concrete type system used, $\mathcal{V}$ might contain other elements as well. We assume that the values of additional type constructors are representable in the CPO $\mathcal{V}$. Then $\mathcal{V}$ is the least solution of the equation

$$\mathcal{V} \;=\; \mathbf{W}_\perp \,+\, \mathcal{V} \to \mathcal{V}.$$

The meaning function on terms is as follows:

$$\begin{array}{lcl}
[\![x]\!]\eta & = & \eta(x) \\
[\![\lambda u.e]\!]\eta & = & \lambda v.[\![e]\!]\eta[u := v] \\
[\![e\,e']\!]\eta & = & \mathsf{if}\,[\![e]\!]\eta \in \mathcal{V} \to \mathcal{V} \,\wedge\, [\![e']\!]\eta \neq \mathbf{W} \\
& & \mathsf{then}\,([\![e]\!]\eta)\,([\![e']\!]\eta) \\
& & \mathsf{else}\,\mathbf{W}
\end{array}$$

$$\begin{array}{lcl}
[\![\mathsf{let}\,x = e\,\mathsf{in}\,e']\!]\eta & = & \mathsf{if}\,[\![e]\!]\eta \neq \mathbf{W} \\
& & \mathsf{then}\,[\![e']\!]\eta[x := [\![e]\!]\eta] \\
& & \mathsf{else}\,\mathbf{W}
\end{array}$$

Note that the above semantics is call–by value.

B. PROOFS

B.1 Proof of Theorem 1 (GRDT to TCET)

First, we introduce a auxilliary definition and lemma to establish a connection between constructive type equality entailment and entailment among type classes.

Definition 5 *Let C be a set of term equality constraints and C' be a set of type class constraints. We say that C is equivalent to C', written as $C \sim C'$, iff $(\forall t\ t'.t = t' \in C$ iff $(Ct\ t\ t' \in C' \wedge Ct\ t'\ t \in C'))$. We call C' the "Ct" equivalent of C; and C the "Eq" equivalent of C'.*

Lemma 8 *Let P_p be a full and faithful type class theory. Let C be a set of equality constraints and C' its "Ct" equivalent. We have $C \vdash^{=_c} t_1 = t_2$ iff $P_p \models C' \supset (Ct\ t_1\ t_2, Ct\ t_2\ t_1)$.*

Proof. The proof is done in two directions. (*Direction* $\Rightarrow$): We proof by induction on derivations.

∘ *Case*:

$$\frac{t = t' \in C}{C \vdash^{=_c} t = t'}$$

Because we have $t = t' \in C$, we know $Ct\ t\ t' \in C'$ and $Ct\ t'\ t \in C'$. Thus $P_p \models C' \supset (Ct\ t\ t', Ct\ t'\ t)$.

∘ *Case*:

$$\frac{C \vdash^{=_c} t_1 = t_2 \quad C \vdash^{=_c} t_2 = t_3}{C \vdash^{=_c} t_1 = t_3}$$

By induction, we have

$$P_p \models C' \supset (Ct\ t_1\ t_2, Ct\ t_2\ t_1, Ct\ t_2\ t_3, Ct\ t_3\ t_2)$$

By the type class instance

$$\forall a_1, a_3.(Ct\ a_1\ a_3 \leftrightarrow \exists a_2.(Ct\ a_1\ a_2 \wedge Ct\ a_2\ a_3))$$

We conclude

$$\begin{array}{l} P_p \models C' \supset (Ct\ t_1\ t_2, Ct\ t_2\ t_1, Ct\ t_2\ t_3, Ct\ t_3\ t_2) \\ \qquad\qquad \supset (Ct\ t_1\ t_3, Ct\ t_3\ t_1) \end{array}$$

Other cases are similar.

(*Direction* $\Leftarrow$):

∘ *Case*: Suppose the type class instance

$$\forall a.(Ct\ a\ a \leftrightarrow True)$$

is applied. Then we have

$$P_p \models True \supset Ct\ t\ t$$

We also have

$$True \vdash^{=_c} t = t$$

∘ *Case*: Suppose the type class instance

$$\forall a_1, a_3.(Ct\ a_1\ a_3 \leftrightarrow \exists a_2.(Ct\ a_1\ a_2 \wedge Ct\ a_2\ a_3))$$

is applied. Then we have

$$P_p \models \exists t_2.(Ct\ t_1\ t_2 \wedge Ct\ t_2\ t_3) \supset Ct\ t_1\ t_3$$

Easily, we also obtain

$$t_1 = t_2 \wedge t_2 = t_3 \vdash^{=_c} t_1 = t_3$$

Other cases are similar. □

The next lemma follows immediately from the rule (M).

Lemma 9 $C, \Gamma \vdash^T$ *$cast : t \to t'$ iff $P_p \models C \supset Ct\ t\ t'$*

We obtain Theorem 1 as a special instance from the following lemma.

Lemma 10 *Let e be a GRDT expression and e' be its fully casted version. Let P_p a full and faithful program theory representing all GRDT type constructors mentioned in e. Silently, we transform the GRDT constructors mentioned in e to TCET constructors. We have that $C, \Gamma \vdash^{G_c} e : t$ iff $C', \Gamma \vdash^T e' : t$ where C' is the "Ct" equivalent of C.*

Proof. The proof is done in two directions.

(*Direction* $\Rightarrow$):We proof by induction on derivation.

∘ *Case* (*Eq*):

$$\frac{C, \Gamma \vdash^{G_c} e : t \quad C \vdash^{=_c} t = t'}{C, \Gamma \vdash^{G_c} e : t'}$$

By the induction hypothesis, we have

$$C', \Gamma \vdash^T e' : t \qquad (1)$$

Also by Lemma 8 and $C \vdash^{=_c} t = t'$ we have

$$P_p \models C' \supset (Ct\ t\ t', Ct\ t'\ t) \quad (2)$$

From (1) and (2), we conclude that

$$C', \Gamma \vdash^T (cast\ e') : t'$$

W.l.o.g. We can assume $e' \equiv (cast\ e'')$. Thus we obtain

$$C', \Gamma \vdash^T ((cast \circ cast)\ e'') : t'$$

We assume $C', \Gamma \vdash^T e'' : t''$. In the above case, the first $cast$ is of type $t \to t'$ and the second $t'' \to t$. Thus by Lemma 9, we know that $Ct\ t\ t'$ and $Ct\ t''\ t$ can be derived from the context. By the (Trans) type class instance, we can derive $Ct\ t''\ t'$. Then by Lemma 9, we know there exists a $cast$ of type $t'' \to t'$. After replacing the cast composition $cast \circ cast$ in the above judgement by the new $cast$, we obtain

$$C', \Gamma \vdash^T (cast\ e'') : t'$$

This is equivalent to

$$C', \Gamma \vdash^T e' : t'$$

$\circ$ *Case (App)*:

$$\frac{C, \Gamma \vdash^{G_c} e_1 : t_2 \to t \quad C, \Gamma \vdash^{G_c} e_2 : t_2}{C, \Gamma \vdash^{G_c} e_1\ e_2 : t}$$

By the induction hypothesis, we have

$$C', \Gamma \vdash^T e_1' : t_2 \to t \quad C', \Gamma \vdash^T e_2' : t_2$$

By application of rule (App), we obtain

$$C', \Gamma \vdash^T (e_1'\ e_2') : t \quad (1)$$

Note that we always have $C \vdash^{=_c} t = t$. Thus we conclude

$$C', \Gamma \vdash^T (cast\ (e_1'\ e_2')) : t$$

Other cases are similar.

(*Direction* $\Leftarrow$): We proceed by structural induction. We denote by $[\![e']\!]$ the "erasure" of expression e', i.e. we erase all $cast$ occurrences from e'. W.l.o.g. We can assume $e' \equiv (cast\ e'')$.

$\circ$ $e'' = x$

$$\frac{C', \Gamma \vdash^T cast : t \to t' \quad C', \Gamma \vdash^T e'' : t}{C', \Gamma \vdash^T (cast\ e'') : t'}$$

Because $e'' = x$, then $[\![e'']\!] = e''$. Therefore, we have

$$C, \Gamma \vdash^{G_c} [\![e'']\!] : t \quad (1)$$

By $C', \Gamma \vdash^T cast : t \to t'$ and Lemma 9, we obtain

$$P_p \models C' \supset (Ct\ t\ t', Ct\ t'\ t)$$

Together with Lemma 8, we have

$$C \vdash^{=_c} t = t' \quad (2)$$

By (1), (2) and rule (Eq), we conclude

$$C, \Gamma \vdash^{G_c} [\![e'']\!] : t'$$

Because $[\![cast\ e'']\!] = [\![e'']\!]$, then we have

$$C, \Gamma \vdash^{G_c} [\![cast\ e'']\!] : t'$$

This is equivalent to

$$C, \Gamma \vdash^{G_c} [\![e']\!] : t'$$

$\circ$ $e'' = \lambda x.e'''$

$$\frac{C', \Gamma \vdash^T cast : t \to t' \quad \dfrac{C', \Gamma.x : t_1 \vdash^T e''' : t_2}{C', \Gamma \vdash^T e'' : t}}{C', \Gamma \vdash^T (cast\ e'') : t'}$$

In the above derivation $t = t_1 \to t_2$. By the induction hypothesis, we have

$$C, \Gamma.x : t_1 \vdash^{G_c} [\![e''']\!] : t_2$$

By applying the (Abs) rule, we obtain

$$C, \Gamma \vdash^{G_c} [\![e'']\!] : t \quad (1)$$

By $C', \Gamma \vdash^T cast : t \to t'$ and Lemma 9, we obtain

$$P_p \models C' \supset (Ct\ t\ t', Ct\ t'\ t)$$

Together with Lemma 8, we have

$$C \vdash^{=_c} t = t' \quad (2)$$

By (1), (2) and rule (Eq), we conclude

$$C, \Gamma \vdash^{G_c} [\![e'']\!] : t'$$

Because $[\![cast\ e'']\!] = [\![e'']\!]$, then we have

$$C, \Gamma \vdash^{G_c} [\![cast\ e'']\!] : t'$$

This is equivalent to

$$C, \Gamma \vdash^{G_c} [\![e']\!] : t'$$

$\circ$ $e'' = (e_1'''\ e_2''')$

$$\frac{C', \Gamma \vdash^T Ct : t \to t' \quad \dfrac{C', \Gamma \vdash^T e_1''' : t_2 \to t \quad C', \Gamma \vdash^T e''' : t_2}{C', \Gamma \vdash^T e'' : t}}{C', \Gamma \vdash^T (Ct\ e'') : t'}$$

By the induction hypothesis, we have

$$C, \Gamma \vdash^{G_c} [\![e_1''']\!] : t_2 \to t$$
$$C, \Gamma \vdash^{G_c} [\![e_2''']\!] : t_2$$

By applying the (App) rule, we obtain

$$C, \Gamma \vdash^{G_c} [\![[\![e_1''']\!]\ [\![e_2''']\!]]\!] : t \quad (1)$$

By $C', \Gamma \vdash^T cast : t \to t'$ and Lemma 9, we obtain

$$P_p \models C' \supset (Ct\ t\ t', Ct\ t'\ t)$$

Together with Lemma 8, we have

$$C \vdash^{=_c} t = t' \quad (2)$$

By (1) and (2), we conclude

$$C, \Gamma \vdash^{G_c} [\![e'']\!] : t'$$

Because $[\![cast\ e'']\!] = [\![e'']\!]$, then we have

$$C, \Gamma \vdash^{G_c} [\![cast\ e'']\!] : t'$$

This is equivalent to

$$C, \Gamma \vdash^{G_c} [\![e']\!] : t'$$

Other cases are similar. $\square$

B.2 Proof of Lemma 2 (Well-Typed)

Our assumptions are: Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$ and Γ such that $C \rightsquigarrow \Gamma$ and $f : Ct\ a\ b \leftrightarrow C$ is valid. Then $\Gamma \vdash f : a \rightarrow b$.

PROOF. The proof proceeds by induction over the proof term construction derivation. W.l.o.g we combine rule ($\forall$ E) with rules (Id),(Var),(Arrow) et. We also combine ($\exists$ E) with (Trans).

∘ *Case*(Id):

$$\lambda x.x : Ct\ a\ a \leftrightarrow True$$

We know that $\Gamma = \emptyset$. Thus we conclude $\Gamma \vdash \lambda x.x : a \rightarrow a$.

∘ *Case* (Var):

$$f : Ct\ a\ b \leftrightarrow f : Ct\ a\ b$$

We know that $\Gamma = \{f : a \rightarrow b\}$. Thus we conclude $\Gamma \vdash f : a \rightarrow b$.

∘ *Case* (Trans):

$$\frac{f = \lambda g.\lambda x.f_2\ (g\ (f_1\ x))}{f : Ct\ a_1\ a_3 \leftrightarrow f_1 : Ct\ a_1\ a_2, f_2 : Ct\ a_2\ a_3}$$

We know that $\Gamma = \{f_1 : a_1 \rightarrow a_2, f_2 : a_2 \rightarrow a_3\}$. Thus by typing derivation we can easily conclude $\Gamma \vdash f : a_1 \rightarrow a_3$.

∘ *Case* (Arrow): Similar to (Trans).

∘ *Case* ($\circ$):

$$\frac{\begin{array}{c} f : Ct\ a\ b \leftrightarrow f_1 : c_1, ..., f_n : c_n \quad f_i : c_i \leftrightarrow F_i \\ F \models F_i \quad \text{for } i = 1, ..., n \end{array}}{f : Ct\ a\ b \leftrightarrow F}$$

By induction, we have $\bigcup_1^n \Gamma_i \vdash f : a \rightarrow b$. Because $\bigcup_1^n \Gamma_i \subseteq \Gamma$ derived from $F \models F_i$, then we conclude $\Gamma \vdash f : a \rightarrow b$.

□

B.3 Proof of Theorem 2 (TCET to ET Soundness)

Theorem 2 follows directly from the following more general lemma.

Lemma 11 *Let* $C, \Gamma \vdash^T e : t$, $C, \Gamma \vdash^T e : t \rightsquigarrow e'$ *and* Γ' *such that* $C \rightsquigarrow \Gamma'$. *Then* $\Gamma \cup \Gamma' \vdash^E e' : t$.

PROOF. The proof proceeds by induction on derivations.

∘ *Case* (*K*):

$$\frac{\begin{array}{c} (K : \forall \bar{a}, \bar{b}.Ct\ t_1\ t_1', Ct\ t_1'\ t_1 ..., Ct\ t_n\ t_n', Ct\ t_n'\ t_n \Rightarrow t \rightarrow T\ \bar{a}) \\ \rightsquigarrow \\ (K' : \forall \bar{a}, \bar{b}.t \rightarrow E\ t_1\ t_1' \rightarrow ... \rightarrow E\ t_n\ t_n' \rightarrow T\ \bar{a}) \\ C, \Gamma \vdash^T e : [\bar{t}/\bar{a}]t \rightsquigarrow e' \\ P_p \models C \supset \overline{(g, h)} : [\bar{t}/\bar{a}](Ct\ t_1\ t_1', Ct\ t_1'\ t_1 ..., Ct\ t_n\ t_n', Ct\ t_n'\ t_n) \end{array}}{C, \Gamma \vdash^T Ke : T\ \bar{t} \rightsquigarrow K'\ e'\ E\ (g_1, h_1) ... (g_n, h_n)}$$

By the induction hypothesis, we have

$$\Gamma \cup \Gamma' \vdash^E e' : [\bar{t}/\bar{a}]t \qquad (1)$$

Also we have

$$K' : \forall \bar{a}, \bar{b}.t \rightarrow E\ t_1\ t_1' \rightarrow ... \rightarrow E\ t_n\ t_n' \rightarrow T\ \bar{a} \qquad (2)$$

Note that here we assume an ordering among the constraints. $P_p \models C \supset \overline{(g, h)} : [\bar{t}/\bar{a}](Ct\, t_1\ t_1', Ct\, t_1'\ t_1 ..., Ct\, t_n\ t_n', Ct\, t_n'\ t_n)$ implies

$$g_i : Ct\ t_i\ t_i' \leftrightarrow C \text{ and } h_i : Ct\ t_i'\ t_i \leftrightarrow C$$

W.l.o.g we can assume $g_i, h_i \notin \Gamma$. Hence by Lemma 2, we have

$$\Gamma \cup \Gamma' \vdash^E g_i : t_i \rightarrow t_i' \text{ and } \Gamma \cup \Gamma' \vdash^E h_i : t_i' \rightarrow t_i$$
where $i = 1 \dots n$

Thus we can obtain that

$$\Gamma \cup \Gamma' \vdash^E E\ (g_i, h_i) : E\ t_i\ t_i' \text{ where } i = 1 \dots n \qquad (3)$$

From (1),(2),(3) and rule (K), we conclude

$$\Gamma \cup \Gamma' \vdash^E K'\ e'\ E\ (g_1, h_1) ... (g_n, h_n) : T\ \bar{t}$$

∘ *Case* (*Reduce*):

$$\frac{D \subseteq C \quad f : Ct\ t_1\ t_2 \leftrightarrow D}{C, \Gamma \vdash^T cast : t_1 \rightarrow t_2 \rightsquigarrow f}$$

Given $D \subseteq C \quad f : Ct\ t_1\ t_2 \leftrightarrow D$, W.l.o.g. we assume $f \notin \Gamma$. Thus we conclude by Lemma 2

$$\Gamma \cup \Gamma' \vdash^E f : t_1 \rightarrow t_2$$

∘ *Case* (*Pat*):

$$\frac{\begin{array}{c} p : t_1 \vdash \forall \bar{b}.(D \mid \Gamma_p \mid p') \quad \bar{b} \cap fv(C, \Gamma, t_2) = \emptyset \\ C \wedge D, \Gamma \cup \Gamma_p \vdash^T e : t_2 \rightsquigarrow e' \end{array}}{C, \Gamma \vdash^T p \rightarrow e : t_1 \rightarrow t_2 \rightsquigarrow p' \rightarrow e'}$$

By the induction hypothesis, we have

$$\Gamma \cup \Gamma_p \cup \Gamma_C \cup \Gamma_D \vdash^T e' : t_2$$

where $C \rightsquigarrow \Gamma_C$ and $D \rightsquigarrow \Gamma_D$.
Also by Lemma 12 (see below), we have $p' \vdash \forall \bar{b}.(\Gamma_p \cup \Gamma_D)$. Thus we conclude

$$\Gamma \cup \Gamma_C \vdash^E p' \rightarrow e' : t_1 \rightarrow t_2$$

∘ Other cases are standard. □

Lemma 12 *Given* $p : t_1 \vdash \forall \bar{b}.(D \mid \Gamma_p \mid p')$ *then* $p' \vdash \forall \bar{b}.\Gamma_p \cup \Gamma'$ *where* $D \rightsquigarrow \Gamma'$.

PROOF. Standard by induction on derivation. □

B.4 Proof of Theorem 3 (TCET to ET Completeness)

Theorem 3 follows directly from the following lemma.

Lemma 13 *Let $C, \Gamma \vdash^T e : t$ and all types appearing in assumption constraints in intermediate derivations are decomposable. The $C, \Gamma \vdash^T e : t \rightsquigarrow e'$ for some e'.*

PROOF. The proof is done by construction of e'.

∘ *Case* (*Reduce*):

Note that $cast$ is a class method of type $\forall t, t'.Ct\ t\ t' \Rightarrow t \rightarrow t'$. Since we have $C, \Gamma \vdash^T cast : t_1 \rightarrow t_2$, by rule (M), we can derive $P_p \vdash C \supset Ct\ t_1\ t_2$.

Given all the types are decomposable, by Lemma 3, we know $f : Ct\ t_1\ t_2 \leftrightarrow C$ for some f if $P_p \vdash C \supset Ct\ t_1\ t_2$. Thus the rule (Reduce) always produces a f.

∘ *Case*:

Other rules are standard. □

B.5 Proofs of Lemmas 4, 5 and 6

B.5.1 Proof of Lemma 4 (Sound CHR Construction)

Our assumptions are: Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$ and $i : CtM\ a\ b, C \rightarrowtail^* D'$ and $castm_i \rightsquigarrow^* e$ such that the CHR derivation is good. Then, $f : Ct\ a\ b \leftrightarrow C$ such that f and e are equivalent.

PROOF. The proof is done through induction on the CHR derivation. W.l.o.g we combine rule (∀ E) with rules (Id), (Var), (Arrow) and (Pair). We also combine (∃ E) with (Trans).

∘ Suppose the rule applied is (Id):

$$\begin{array}{lcl} i : CtM\ a\ b, C & \rightarrowtail & a = b, C \rightarrowtail^* D' \\ \mathtt{castm}_i & \rightsquigarrow & \lambda x.x \end{array}$$

Note that the above derivation unifies a and b. Thus we have

$$\lambda x.x : Ct\ a\ a \leftrightarrow True.$$

∘ Suppose the rule applied is (Trans1):

$$\begin{array}{lcl} i : CtM\ a\ b, C & \rightarrowtail & a_g = a, j : CtM\ b_g\ b, C \rightarrowtail^* D' \\ \mathtt{castm}_i & \rightsquigarrow & \mathtt{castm}_j \circ g \end{array}$$

Note that the above derivation unifies a and a_g. Thus we have

$$(\circ)\ \dfrac{(\text{Trans})\ \dfrac{f = castm_j \circ g}{f : Ct\ a\ b \leftrightarrow g : Ct\ a\ b_g, castm_j : Ct\ b_g\ b}}{f : Ct\ a\ b \leftrightarrow D}$$

where $g : Ct\ a\ b_g \subseteq C$. Also by induction, we know $j : Ct\ b_g\ b \leftrightarrow D'$ for some $D' \subseteq C$. Take D as D', we have $D \subseteq C$.

∘ Suppose the rule applied is (Arrow):

$$\begin{array}{lcl} i : CtM\ (a_1 \rightarrow a_2)\ (b_1 \rightarrow b_2), C & \rightarrowtail & i_1 : CtM\ b_1\ a_1, \\ & & i_2 : CtM\ a_2\ b_2, C \rightarrowtail^* D' \\ \mathtt{castm}_i & \rightsquigarrow & \lambda g.\lambda x. \\ & & \mathtt{castm}_{i_2}(g\ (\mathtt{castm}_{i_1}\ x)) \end{array}$$

Also we have

$$(\circ)\ \dfrac{(\text{Trans})\ \dfrac{f = \lambda g.\lambda x.castm_{i_2}(g\ (castm_{i_1}\ x))}{f : Ct\ (a_1 \rightarrow a_2)\ (b_1 \rightarrow b_2) \leftrightarrow castm_{i_1} : Ct\ b_1\ a_1,\ castm_{i_2} : Ct\ a_2\ b_2}}{f : Ct\ (a_1 \rightarrow a_2)\ (b_1 \rightarrow b_2) \leftrightarrow D}$$

Also by induction, we know $j : Ct\ b_1\ a_1 \leftrightarrow D'$ and $j : Ct\ a_2\ b_2 \leftrightarrow D''$for some $D' \subseteq C$ and $D'' \subseteq C$. Take D as $D' \cup D''$, we have $D \subseteq C$.

∘ (Pair) is similar to (Arrow). □

B.5.2 Proof of Lemma 5 (Complete CHR Construction)

Our assumptions are: Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$ such that $f : Ct\ a\ b \leftrightarrow C$. Then, $i : CtM\ a\ b, C \rightarrowtail^* C$ such that $castm_i \rightsquigarrow^* e$ and f and e are equivalent.

PROOF. W.l.o.g we combine rule (∀ E) with rules (Id), (Var), (Arrow) and (Pair). We also combine (∃ E) with (Trans).

∘ Case (Id).

$$\lambda x.x : Ct\ a\ a \leftrightarrow True$$

Then we have

$$\begin{array}{lcl} i : CtM\ a\ a, C & \rightarrowtail_{Id} & a = a, C \\ \mathtt{castm}_i & \rightsquigarrow & \lambda x.x \end{array}$$

∘ Case (Var).

$$f : Ct\ a\ b \leftrightarrow f : Ct\ a\ b$$

Then we have, given $f : Ct\ a\ b \in C$

$$\begin{array}{lclcl} i : CtM\ a\ b, C & \rightarrowtail_{Trans1} & j : CtM\ b\ b, C & \rightarrowtail_{Id} & C \\ \mathtt{castm}_i & \rightsquigarrow & \mathtt{castm}_i \circ f & \rightsquigarrow & \lambda x.x \circ f \end{array}$$

∘ Case (Trans).

$$(\text{Trans})\ \dfrac{f = \lambda x.f_2\ (f_1\ x)}{f : Ct\ a_1\ a_3 \leftrightarrow f_1 : Ct\ a_1\ a_2, f_2 : Ct\ a_2\ a_3}$$

We have

$$\begin{array}{ll} & i : CtM\ a_1\ a_3, f_1 : Ct\ a_1\ a_2, f_2 : Ct\ a_2\ a_3 \\ & \mathtt{castm}_i \\ \rightarrowtail_{Trans1} & j : Ct\ a_2\ a_3, f_1 : Ct\ a_1\ a_2, f_2 : Ct\ a_2\ a_3 \\ \rightsquigarrow & \mathtt{castm}_j \circ f_1 \\ \rightarrowtail_{Trans1} & k : CtM\ a_3\ a_3, f_1 : Ct\ a_1\ a_2, f_2 : Ct\ a_2\ a_3 \\ \rightsquigarrow & \mathtt{castm}_k \circ f_2 \circ f_1 \\ \rightarrowtail_{Id} & f_2 \circ f_1 : Ct\ a_1\ a_2, f_2 : Ct\ a_2\ a_3 \\ \rightsquigarrow & \lambda x.x \circ f_2 \circ f_1 \end{array}$$

∘ Case (Arrow).

$$(\text{Arrow})\ \dfrac{f = \lambda g.\lambda x.f_2\ (g\ (f_1\ x))}{\forall a_1, a_2, b_1, b_2.f : Ct\ (a_1 \rightarrow a_2)\ (b_1 \rightarrow b_2) \leftrightarrow f_1 : Ct\ b_1\ a_1, f_2 : Ct\ a_2\ b_2}$$

By induction,

$$\begin{array}{lll} C, i_1 : CtM\ b_1\ a_1 & \rightarrowtail^* & D_1 \\ castm_{i_1} & \rightsquigarrow^* & f_1 \end{array}$$

$$\begin{array}{lll} C, i_2 : CtM\ a_2\ b_3 & \rightarrowtail^* & D_2 \\ castm_{i_2} & \rightsquigarrow^* & f_2 \end{array}$$

Therefore

$$\begin{array}{ll} & i : CtM\ (a_1 \to a_2)\ (b_1 \to b_2), f_1 : Ct\ b_1\ a_1, \\ & f_2 : Ct\ a_2\ b_2 \\ & \mathtt{castm}_i \\ \rightarrowtail_{Arrow} & i_1 : CtM\ b_1\ a_1, i_2 : CtM\ a_2\ b_2, f_1 : Ct\ b_1\ a_1, \\ & f_2 : Ct\ a_2\ b_2 \\ \rightsquigarrow & \lambda g.\lambda x.\mathtt{castm}_{i_2}\ (g\ (\mathtt{castm}_{i_1}\ x)) \\ \rightarrowtail^*_{Var} & f_1 : Ct\ b_1\ a_1, f_2 : Ct\ a_2\ b_2 \\ \rightsquigarrow^* & \lambda g.\lambda x.f_2\ (g\ (f_1\ x)) \end{array}$$

∘ (Pair) is similar to (Arrow).

□

B.5.3 Proof of Lemma 6 (Sound Term Construction)

Our assumptions are: Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$, $i : CtM\ a\ b, C \rightarrowtail^* D_1$ and $castm_i \rightsquigarrow^* e_1$ and $i : CtM\ a\ b, C \rightarrowtail^* D_2$ and $castm_i \rightsquigarrow^* e_2$ such that both CHR derivations are good. Then, e_1 and e_2 are equivalent.

PROOF. Let $f : Ct\ a\ b \leftrightarrow C$, from Lemma 4, we know that e_1 is equivalent to f and e_2 is equivalent to f. Thus we conclude that e_1 is equivalent to e_2. □

B.6 Termination of CHRs

We impose a termination condition on derivations. We show that this condition does not rule out any good derivations which are vital. The basic idea is to attach each constraint with a distinct *justification*. Justifications J refer to sets of numbers. Each Ct constraints carries a distinct, singleton justifications sets. Each CtM constraints carries initially a singleton justification set referring to its location. We write j as a short-hand for the singleton set $\{j\}$. We need to maintain justifications during CHR applications.

Consider rule instance (Trans1) $g : Ct\ a\ b, i : CtM\ a'\ b' \iff g : Ct\ a\ b, a = a', j : CtM\ b\ b'$ and store C such that $(g : Ct\ a\ b)_j, (i : CtM\ a'\ b')_J \in C$ Then $C \rightarrowtail_{Trans1} C - (i : CtM\ a'\ b')_J, a = a', (j : CtM\ b\ b')_{\{j\} \cup J}$. We say that the *termination condition* is violated iff $j \in J$.

Consider rule instance (Arrow) $i : CtM\ (a_1 \to a_2)\ (b_1 \to b_2) \iff i_1 : CtM\ b_1\ a_1, i_2 : CtM\ a_2\ b_2$ such that $(i : CtM\ (a_1 \to a_2)\ (b_1 \to b_2))_J \in C$. Then, $C \rightarrowtail_{Arrow} C - (i : CtM\ (a_1 \to a_2)\ (b_1 \to b_2))_J, (i_1 : CtM\ b_1\ a_1)_J, (i_2 : CtM\ a_2\ b_2)_J$. The justified CHR semantics for rule (Pair) is similar.

Silently, we assume that all propagation rules have been exhaustively applied such that all Ct constraints are attached with a unique number. Note that we could encounter "duplicates" such as $(g_1 : Ct\ a\ b)_{j_1}$ and $(g_2 : Ct\ a\ b)_{j_2}$. However, g_1 and g_2 are equivalent. Hence, we may keep both constraints.

We impose an order among derivations. Let $C = \{f_1 : Ct\ a_1\ b_1, ..., f_n : Ct\ a_n\ b_n\}$, $i : CtM\ a\ b, C \rightarrowtail^* D_1$ and $castm_i \rightsquigarrow^* e_1$ and $i : CtM\ a\ b, C \rightarrowtail^* D_2$ and $castm_i \rightsquigarrow^* e_2$ such that both CHR derivations are good. We say that $i : CtM\ a\ b, C \rightarrowtail^* D_1$ is *shorter* than $i : CtM\ a\ b, C \rightarrowtail^* D_2$ iff the size of e_1 is shorter than the size of e_2 where the size function returns the number of nodes in the syntax tree of an expression. In case of initial stores with multiple CtMs we compare the sum of the individual sizes of resulting expressions.

Lemma 14 *Let $i : CtM\ t\ t, C \rightarrowtail^* D$ be a good derivation. Then, $castm_i \rightsquigarrow^* e$ where e is equivalent to the identity.*

Lemma 15 *Any good derivation which violates the termination condition can be shortened.*

PROOF. We assume a good derivation which violates the termination condition where we consider the "earliest" violation in the derivation.

$$\begin{array}{llll} & C & & \\ \rightarrowtail & ... & & \\ \rightarrowtail & C_1, (g : Ct\ t_1\ t_2)_{l_1}, (i : CtM\ t_1'\ t_2')_{L_1} & l_1 \notin L_1 & \\ \rightarrowtail_{Trans1} & C_1, (g : Ct\ t_1\ t_2)_{l_1}, (j : CtM\ t_2\ t_2')_{\{l_1\} \cup L_1}, & & \\ & t_1 = t_1' & (1) & \\ \rightarrowtail & ... & & \\ \rightarrowtail & C_2, (g : Ct\ t_1\ t_2)_{l_1}, (k : CtM\ t_1''\ t_2'')_{L_2} & l_1 \in L_2 & (2) \\ \rightarrowtail_{Trans1} & C_2, (g : Ct\ t_1\ t_2)_{l_1}, (n : CtM\ t_2\ t_2'')_{L_2}, & & \\ & t_1 = t_1'' & & \\ \rightarrowtail & ... & & \\ \rightarrowtail & D & & \end{array}$$

W.l.o.g., in the derivation steps between (1) and (2) we only apply CHRs on $(j : CtM\ t_2\ t_2')_{\{l_1\} \cup L_1}$ or its successors, i.e. those resulting from (Trans1) and (Arrow) rules.

First, we show that only (Trans1) or (Id) rules could have been applied on $(j : CtM\ t_2\ t_2')_{\{l_1\} \cup L_1}$ or its successors. Assume the contrary, that is some (Pair) (or a similar type-constructor) rule has been applied on $(j : CtM\ t_2\ t_2')_{\{l_1\} \cup L_1}$. Then,

$$\begin{array}{ll} & ..., (g : Ct\ t_1\ t_2)_{l_1}, t_2 = (t_3, t_4), t_2' = (t_5, t_6), \\ & (j : CtM\ t_2\ t_2')_{\{l_1\} \cup L_1} \\ \rightarrowtail_{Pair} & ..., (g : Ct\ t_1\ t_2)_{l_1}, t_2 = (t_3, t_4), t_2' = (t_5, t_6), \\ & (j_1 : CtM\ t_3\ t_5)_{\{l_1\} \cup L_1}, (j_2 : CtM\ t_4\ t_6)_{\{l_1\} \cup L_1} \end{array}$$

However, then we obtain a cycle among types. E.g., assume that $(j_1 : CtM\ t_3\ t_5)_{\{l_1\} \cup L_1}$ equals $(k : CtM\ t_1''\ t_2'')_{L_2}$. We find that $t_1 = t_1', t_2 = (t_3, t_4), t_2'' = (t_5, t_6), t_1'' = t_3, t_1 = t_1''$ which implies $(g : Ct\ t_1\ (t_1, t_4))_{l_1}$. Thus, we obtain a contradiction. Note that by assumption the type equations resulting from Ct constraints ($Ct\ a\ b$ yields $a = b$) must be satisfiable. Otherwise, the GRDT definition is invalid.

Hence, we only find (Trans1) or (Id) applications in between (1) and (2). Effectively, we generate a cast function to convert t_1 into some b which then we convert back into t_1. However, any such transformation yields a cast function which is equivalent to the identity. See Lemma 14. Hence, the steps between (1) and (2) are redundant. Hence, we obtain a shorter derivation. □

Lemma 16 *CHRs are terminating under the termination condition.*

PROOF. Follows immediately. Note that we disallow Ct assumptions of the form $g : Ct\ a\ (a, b)$. Hence, any non-terminating derivation must violate the termination condition. $\square$