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 (ET). 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 [5] 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 ∀a,b.(a=[b]) ⇒ a → 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.

\[ \text{data E a b = E (a -> b, b -> a)} \]
\[ \text{data Erk_H' a = I_H' a (E a Int)} \]
\[ \quad \mid \text{forall b. L_H' a (E a [b])} \]
\[ f_H' :: \text{Erk_H' a -> a} \]
\[ f_H' (I_H' x (E (g,h))) = h ((+) (g x) 1) \]
\[ f_H' (L_H' x (E (g,h))) = h (\text{tail} (g x)) \]

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 -> Int. Then, we apply (+) which is assumed to have type Int -> Int -> 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 contributions 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 \( \sigma \) to denote a sequence of objects \( o_1, \ldots, o_n \). We write \( f(o) \) 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., \( \text{\textit{f}}[a] \) denotes the substitution which has the effect of replacing each occurrence of \( a \) by \( b \). Often, we abbreviate \( a_1, \ldots, a_n \) by \( [a] \).

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 \( \approx \) 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 \( \exists x \) denote the formula \( \exists x_1 \ldots \exists x_n \).

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 [9]. Our formulation is closest to the system described by Simonet and Pottier [22].

First, we define the set of expressions and types.

\[ \text{Expressions} \quad e ::= K [x | \lambda x.e] \quad e \text{ case of } [p_i \rightarrow e_i]_{i \in t} \]
\[ \text{Patterns} \quad p ::= x \mid (p, p) \mid K p \]
\[ \text{Types} \quad t ::= a \mid t \rightarrow t \mid T \]

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

\[ \text{data Erk a = (a=Int) \Rightarrow I a \mid forall b. (a=[b]) \Rightarrow L a} \]

imply constructors \( I : \forall a. a = \text{Int} \Rightarrow a \rightarrow \text{Erk} a \) and \( L : \forall a, b.a = [b] \Rightarrow a \rightarrow \text{Erk} a \). We prohibit “invalid” definitions such as \( \text{data \textbf{Unsat} a = (a=(a, \text{Int})) \Rightarrow U a} \) which yields a constructor with an unsatisfiable set of equations. We assume that booleans, integers, pairs and lists are pre-defined.

The typing rules describing well-typing of GRDT expressions are in Figure 1. We introduce judgments \( C, \Gamma \vdash \text{e :: t} \) to denote that expression \( e \) has type \( t \) under constraint \( C \) and environment \( \Gamma \). 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 \( \Gamma \) we record the types of lambda-bound variables and primitive functions such as \( \text{head} : \forall a. \rightarrow a, \text{tail} : \forall a. \rightarrow [a] \), 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 \( \Gamma_{inl} \).

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 \( \phi \) of \( C \) we have
that \( \phi(t_1) = \phi(t_2) \) holds. In rule (Pat) we make use of an auxiliary judgment \( p : t \vdash \forall \delta \cdot (D, \Gamma_p) \) which expresses a relation among pattern \( p \) of type \( t \) and the binding \( \Gamma_p \) of variables in \( p \). Variables \( \delta \) refer to all “existential” variables.

Logically, these variables must be considered as universally quantified. Hence, we write \( \forall \delta \). The side condition \( \delta \cap \text{fv}(C, \Gamma, t_2) = \emptyset \) prevents existential variables from escaping. In rule (P-Pair), we assume that there are no name clashes between variables \( \delta_1 \) and \( \delta_2 \). 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 \( \mathbf{x} \cdot \mathbf{y} \) provides the additional type assumption \( a = \text{Int} \) which is used in typing of the body \( \mathbf{x} \cdot \mathbf{y} \). Note that because of this additional assumption, rule (Eq) is able to turn the type of \( \mathbf{x} \) from \( a \) to \( \text{Int} \). Thus, the expression \( \mathbf{x} \cdot \mathbf{y} \) is well typed. Similarly, rule (Eq) also turns the type of \( \mathbf{x} \cdot \mathbf{y} \) 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

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

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

As already observed by Cheney and Hnizd [4], such meaningless programs can always be replaced by “undefined”. Note that we never ever construct a value of type \( \text{Erk Bool} \). 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 \( \text{False} \) occurs in (intermediate) typing judgments. The definition of constructive entailment among type equality is as follows:

\[
\begin{align*}
\Gamma &\vdash t = t' \in C \\
C &\vdash \neg e \rightarrow t = t' \\
C &\vdash \neg e \rightarrow t_1 = t_2 \\
C &\vdash \neg e \rightarrow t_3 = t_3
\end{align*}
\]

We obtain the constructive GRDT system \( \vdash \neg e \) by replacing (Eq) with the following rule.

\[
\begin{align*}
\Gamma &\vdash e : t \\
C &\vdash \neg e \rightarrow t = t'
\end{align*}
\]

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 \( \text{Ct} \ a \ b \) 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.

```plaintext
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 perform 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 \( C \, t_1 \, t_2 \) and \( Ct \, t_1 \, t_2 \). 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) => L_H a | forall b. (Ct a [b], Ct [b] a) => L_H a
f_H ::= Erk_H a => a_L_H (L_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 \( C t \) [a] and \( Ct \) [b] a. Thus, \( x \) can be given type \( b \).

We make use of instance (Id) to show that cast tail has type \( [b] \Rightarrow [b] \). Hence, \( (\text{cast tail}) \, (\text{cast x}) \) 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 Läuffer [13]. In our own work [23], we 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 new type (multi-parametr) 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 logical formulae. We commonly denote these logical formulae by \( P \) and refer to \( P \) as the program theory. E.g., the instance declarations from above can be described by the following first-order formulae.

\[
\forall a. (Ct a \ a \leftrightarrow \text{True})
\]

\[
\forall a_1, a_2, b_1, b_2. (Ct (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2) \leftrightarrow Ct \ (a_1 \ a_2 \wedge Ct \ a_2 \ b_2))
\]

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

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

For each class declaration class \( TC \, a_1 \, ... \, a_n \) where \( m \, : \, t \) we assume a new primitive \( m : \forall \alpha. TC \, \alpha \Rightarrow t \). For simplicity, we restrict ourselves to monomorphic methods. That is, we require that \( m \, : \, t \) \( \subseteq \alpha \). 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 introduce rule (K) and introduce a new rule (M) to take care of method class methods.

Let \( K \) : \( \forall \alpha. b \, : \, \text{TC} \, \alpha \Rightarrow t \) -> T \( \alpha \). Note that we restrict to monomorphic methods.

\[
K \, : \, \forall \alpha. b \, : \, \text{TC} \, \alpha \Rightarrow t \, \Rightarrow \, T \, \alpha
\]

We provide the following example 4 where we denote \( m : \forall \alpha. TC \, \alpha \Rightarrow t \) \( m \, : \, T \, \alpha \, \Rightarrow \, \alpha \, \alpha \, \alpha \).

\[
\text{Example 4}
\]
typing derivation (including $C \vdash_{s} t_{1} = t_{2}$ 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”.\footnote{Indeed, GHC will only accept instance (Trans) once we turn on the “undecidable instances” option.} E.g., consider instance (Trans) from above. When performing context reduction\footnote{This is the process of resolving type classes w.r.t. a given set of class and instance declarations.} we need to guess the intermediate type when applying instance (Trans). Hence, context-reduction may or may not terminate. Hence, the check whether $C \supset C_{t_{1} t_{2}}$ holds where $C$ is a set of $C_{t}$ 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 $C_{t} t_{i} t'$ 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 $C_{t} t_{t}$.

We start off by describing our proof system. We assume that constraints such as $f : C _ {t} a b$ carry now a proof term $f$ representing “evidence” for $C_{t} a b$. We silently drop $f$ in case proof terms do not matter. We introduce judgments of the form $f : C_{t} a b \Rightarrow F$ to denote that $f$ is the proof term corresponding to $C_{t} a b$ 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 (s) allows for the structural composition of proof terms. Rules (VE) and (GE) deal with universal and existential quantifiers. In essence, we make the construction rules represented by $C_{t}$ instance declarations explicit.

Example 5 We give the derivation tree for $f : C_{t} a$ (Int, Bool)
$\vdash g_{1} : C_{t} a$ (Int, $c_{1}$), $g_{2} : C_{t} b$ Int $g_{3} : C_{t} c$ Bool in Figure 2. For convenience, we combine rule (VE) with rules (Id), (Var), (Arrow). We conclude that $f x = \text{let } g_{1} (x, y) = (g_{2} x, g_{3} y) \text{ in } g_{1} (g_{1} x)$

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

\textbf{Lemma 1 (Soundness)} Let $P_{e}$ be the program theory. Let $C = \{f_{1} : C_{t_{1}} a_{1} b_{1}, \ldots, f_{n} : C_{t_{n}} a_{n} b_{n}\}$ such $f : C_{t} a b \Leftrightarrow C$ is valid. Then, $P_{e} \vdash C \supset C_{t} a b$.

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

\textbf{Definition 3} Let $C = \{f_{1} : C_{t_{1}} a_{1} b_{1}, \ldots, f_{n} : C_{t_{n}} a_{n} b_{n}\}$.
We construct an environment $\Gamma$ out of $C$, written as $C \supset \Gamma$, by mapping each $t_{i}$ : $C_{t_{i}} a_{i} b_{i} \Rightarrow \Gamma : a \Rightarrow b$.

\textbf{Lemma 2 (Well-Typed)} Let $C = \{f_{1} : C_{t_{1}} a_{1} b_{1}, \ldots, f_{n} : C_{t_{n}} a_{n} b_{n}\}$ and $\Gamma$ such that $C \supset \Gamma$ and $f : C_{t} a b \Leftrightarrow C$ is valid. Then $\Gamma \vdash f : a \Rightarrow b$.

Proofs can be found in Appendix B. Note that the proof term $f$ is equivalent to the identity assuming $f_{1}, \ldots, f_{n}$ 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 $f : C_{t} a b \Leftrightarrow C$ given $C_{t} a b$ and $C$.

We are in the position to systematically translate TCET to ET. Each TCET constructor $K : \sigma$ is turned into an ET constructor $K' : \sigma'$, written $(K : \sigma) \Rightarrow (K' : \sigma')$. We have that $(K : \forall a, b. D \Rightarrow t \Rightarrow \overline{a}, \overline{b} \Rightarrow E) \Rightarrow (K' : \forall a, b. D \Rightarrow t_{1}, t_{2} \Rightarrow \overline{a}, \overline{b} \Rightarrow E_{1}, t_{1}, t_{2} \Rightarrow \overline{a}, \overline{b} \Rightarrow E_{2})$ where $D = \{C_{t_{1}} t_{1}, C_{t_{2}} t_{2}, C_{t_{1}} t_{3}, C_{t_{2}} t_{4}, C_{t_{1}} t_{5}, C_{t_{2}} t_{6}\}$. Silently, we assume a fixed order among $C_{t}$ constraints. Note that the type constructor $E$ is defined in Example 2.

For the translation of expressions we introduce judgments of the form $E_{1} t_{1} e \Rightarrow e' : e'$ where $C_{t}$ holds $C_{t}$ assumptions, $e$ is a TCET expression and $e'$ is an 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 $P_{e} \vdash C \supset (g, h) : [\overline{f / \overline{a}} / \overline{b}] D$ iff $g : C_{t_{1}} t_{1}, t_{1} \Rightarrow C$ and $h_{i} : C_{t_{1}} t_{1}, t_{i} \Rightarrow C$ for $i = 1, \ldots, n$ where $[\overline{f / \overline{a}} / \overline{b}] D = \{C_{t_{1}} t_{1}, C_{t_{1}} t_{1}, \ldots, C_{t_{n}} t_{n}, C_{t_{n}} t_{n}\}$. Note that $P_{e} \vdash C \supset (g, h) : [\overline{f / \overline{a}} / \overline{b}] D$ implies that $P_{e} \vdash C \supset [\overline{f / \overline{a}} / \overline{b}] D$ (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 $\Gamma \vdash^{T} e : t$ to denote a judgment in the ET system.

\textbf{Theorem 2 (TCET to ET Soundness)} Let $\text{True}, \Gamma \vdash^{T} e : t$ and $\text{True}, \Gamma \vdash^{T} e' : t'$. 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:

\[
\begin{align*}
(\text{Id}) & \quad \forall a. \lambda x : C t \ a \ a \leftrightarrow \text{True} \quad (\text{Var}) & \quad \forall a, b, f : C t \ a \ b \leftrightarrow f : C t \ a \ b \\
(\text{Trans}) & \quad f = \lambda x. f_2 \ (f_1 \ x) \quad \forall a, a_3, f : C t \ a \ a_3 \leftrightarrow \exists a_2. f_1 : C t \ a_1 \ a_2, f_2 : C t \ a_2 \ a_3
\end{align*}
\]

\[
(\text{Arrow}) \quad f = \lambda g. \lambda x. f_2 \ (g \ (f_1 \ x)) \quad \forall a_1, a_2, b_1, b_2, f : C t \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2) \leftrightarrow f_1 : C t \ b_1 \ a_1, f_2 : C t \ a_2 \ b_2
\]

\[
(\text{Pair}) \quad f = \lambda (x, y). \ (f_1 \ x, f_2 \ y) \quad \forall a_1, a_2, b_1, b_2, f : C t \ (a_1, a_2) \ (b_1, b_2) \leftrightarrow C
\]

\[
(\exists) \quad f : C t \ a \ b \leftrightarrow F = \{ f_i : C t \ a \ b \mid i = 1, \ldots, n \}
\]

\[
(\forall) \quad f = \lambda \phi. \ (\forall x. \phi) \quad \forall a, f : C t \ a \ b \leftrightarrow F = \{ f_i : C t \ a \ b \mid i = 1, \ldots, n \}
\]

\[
(\exists) \quad f : C t \ a \ (\text{Int, Bool}) \leftrightarrow \exists g : C t \ a \ (b, c), g : C t \ (b, c) \ (\text{Int, Bool})
\]

\[
(\forall) \quad g_1 : C t \ a \ (b, c) \leftrightarrow g_1 : C t \ a \ (b, c) \ (\text{Pair}) \quad g_1 : C t \ a \ (b, c) \leftrightarrow g_1 : C t \ a \ (b, c)
\]

\[
\begin{align*}
(\text{Var}) & \quad g : C t \ a \ (b, c) \leftrightarrow g_1 : C t \ a \ (b, c) \ (\text{Pair}) & \quad g_2 : C t \ b \ (\text{Int, Bool}) \leftrightarrow g_2 : C t \ b \ (\text{Int, Bool})
\end{align*}
\]

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 C t a b \rightarrow C t \ (Foo a) \ (Foo b) \ where \ K = K

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

The above is not surprising. Similar situations arise for simple type class programs. E.g., we cannot decompose \( \text{Eq} \ a a \) into \( \text{Eq} \ a \) 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 \( i \in \{1, \ldots, n\} \) iff a proof term construction rule \( f_i : C t \ (a_i, a_{i+1} \ldots, a_n, T b_1 b_2 \ldots, b_m) \leftrightarrow C t \ (T b_1 \ldots b_n) \) exists such that (1) \( f_i \) is well-typed under \( (g : T a_1 \ldots a_n \rightarrow T b_1 \ldots b_n, h) : C t \ (T b_1 \ldots b_n) \rightarrow T a_1 \ldots a_n \) 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 \( \bot : \forall a. a \). Consider

\[
(\text{Pair1}) \quad g_1 = \lambda x. \text{fst} \ (f \ (x, \bot)) \quad g_1 : C t \ a_1 \ b_1 \leftrightarrow f : C t \ (a_1, a_2) \ (b_1, b_2)
\]

\[
(\text{Pair2}) \quad g_2 = \lambda x. \text{snd} \ (f \ (\bot, x)) \quad g_2 : C t \ a_2 \ b_2 \leftrightarrow f : C t \ (a_1, a_2) \ (b_1, b_2)
\]

However, function types seem only to be decomposable in their co-variant position under a non-strict semantics. \( \square \)
\begin{figure}[h]
\begin{align*}
\text{(Abs)} & \quad C, \Gamma; x : t_1 \vdash t : t_2 \sim e \quad \Rightarrow \quad C, \Gamma; \lambda x. e : t_1 \rightarrow t_2 \sim \lambda x. e' \\
\text{(App)} & \quad C, \Gamma; e : t_1 \rightarrow e' \quad \Rightarrow \quad C, \Gamma; e_1 : t \sim e'_1 \\
\text{(Var-x)} & \quad \frac{\,(x : \forall \alpha.t) \in \Gamma\,}{C, \Gamma; \vdash x : \overline{t/\alpha} \sim x} \\
\text{(Reduce)} & \quad \frac{D \subseteq C \quad f : C t_1 t_2 \leftrightarrow D}{C, \Gamma; \vdash \text{cast : } t_1 \rightarrow t_2 \sim f} \\
\text{(Case)} & \quad \frac{C, \Gamma; \vdash e : t \sim e'}{C, \Gamma; \vdash \pi_i e : t_1 \sim \pi_i e'} \quad \text{for } i \in I \\
\text{(Pat)} & \quad \frac{\vdash p : t \vdash \forall \alpha. (D \Gamma p \quad p') \quad \overline{\alpha} \cap \mathcal{F}(C, \Gamma, t_2) = \emptyset}{C, \Gamma; \vdash \overline{\alpha} e : \overline{t/\alpha} \sim e'} \\
\text{(K)} & \quad \frac{\vdash \alpha \in C \quad (g, h) : \overline{f/\alpha}}{C, \Gamma; \vdash K e : T \overline{f} \sim K' e' : E (g_1, h_1) \ldots E (g_n, h_n)} \\
\text{(P-Var)} & \quad \frac{x : t \vdash (\text{True} \mid \{x : t\}) t}{P, \overline{\alpha} e : \overline{t/\alpha} \sim e'} \\
\text{(P-Pair)} & \quad \frac{p_1 : \vdash t_1 \vdash \forall \alpha_1. (D_1 \Gamma p_1 \quad p'_1) \quad p_2 : \vdash t_2 \vdash \forall \alpha_2. (D_2 \Gamma p_2 \quad p'_2)}{(p_1, p_2) : \vdash \forall \alpha. (D_1 \Gamma p_1 \quad p'_1) \quad (p_1, p_2) : \vdash \forall \alpha_2. (D_2 \Gamma p_2 \quad p'_2)} \\
\text{(P-K)} & \quad \overline{\alpha} \cap \overline{\alpha} = \emptyset \quad \overline{\alpha} \vdash p : \overline{f/\alpha} t \vdash \forall \alpha. (D' \Gamma p \quad p') \quad \overline{g_1, h_1, \ldots, g_n, h_n} \quad \text{fresh} \\
& \quad K' \vdash (D', g_1, h_1 : C t_1 t_1' \ldots g_n : C t_n t_n' : C t_n' t_n) \\
& \quad \overline{g_1, h_1, \ldots, g_n, h_n} \\
& \quad \vdash \overline{f/\alpha} (D' \Gamma p \quad p') \quad \overline{g_1, h_1, \ldots, g_n, h_n} \quad \text{fresh} \\
& \quad D'' = (D', g_1, h_1 : C t_1 t_1' \ldots g_n : C t_n t_n' : C t_n' t_n) \\
& \quad K' \vdash (D'' \Gamma p \quad p') \quad (g_1, h_1) \ldots (g_n, h_n) \\
\end{align*}
\end{figure}

**Example 8**

\[ g = \lambda x. (f (\lambda y. x)) \Downarrow \]
\[ g : C t_1 \leftrightarrow f : C (a_1 \rightarrow a_2) (b_1 \rightarrow b_2) \]

Note that \( g \) is the identity under a non-strict semantics. However, it seems that \( h : C t_1 a_1 \leftrightarrow f : Ct \ (a_1 \rightarrow a_2) (b_1 \rightarrow b_2) \) 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:

\[ g = \lambda x. \text{project}_L \ (f \ (\text{inject}_L \ x)) \quad \Downarrow \quad \text{EitherL} \]
\[ g : C t_1 \leftrightarrow f : C (\text{Either} \ a_1 \ a_2) (b_1 \leftrightarrow b_2) \]

\[ g = \lambda x. \text{project}_R \ (f \ (\text{inject}_R \ x)) \quad \Downarrow \quad \text{EitherR} \]
\[ g : C t_1 \leftrightarrow f : C (\text{Either} \ a_1 \ a_2) (b_1 \leftrightarrow b_2) \]

Note that the decomposition conditions (Definition 4) are satisfied. Consider the (EitherL) case. Expressions are well-typed. Assume \( f \) is the identity. Then, \( f \ (\text{inject}_L \ x) \) 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 $P_z$ 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 $P_z \vdash C \supset Ct_t_1 t_2$ and all types appearing in constraints are decomposable. Then, $f : Ct_t_1 t_2 \iff C$ for some proof term $f$.

The proof is straightforward and proceeds by induction over $P_z \vdash C \supset Ct_t_1 t_2$.

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 $\Gamma \vdash^\tau e : t$ and all types appearing in assumption constraints in intermediate derivations are decomposable. Then $\Gamma \vdash^\tau e : t \sim e'$ for some $e'$.

6. DECIDABLE TERM CONSTRUCTION METHOD

We introduce a method to decide $f : Ct_t_1 t_2 \iff C$ (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 constraint assumptions whereas $Ct_t_1 t_2$ refers to a use site (see rule (Reduce) in Figure 3). In order to distinguish between $Ct_t_1 t_2$ uses and assumptions we write $Ct_M t_1 t_2$ to refer to a use of $Ct_t_1 t_2$. Our task is to construct $Ct_M$ out of a given set of $Ct$ assumptions. Note that $Ct_t_1 t_2$ can be viewed as directed edges. Hence, the successful construction of a $Ct_M$ use is equivalent to finding a path in the graph of $Ct$ edges. However, we do not rely on our method on graph algorithms. We would like our method to work even under some additional side conditions such as $Ct_M t_1 t_2$. $Ct_M$ is $a \rightarrow a$. That is, construct $Ct_M t_1 t_2$ and $Ct_M t_0 t_1$ out of some assumption set $C$ under the side condition that $t_2 = t_1 \rightarrow a$ for some $a$. Therefore, we view proof term construction as constraint solving where we rewrite constraint stores until all $Ct_M$s have been resolved.

The formal development is as follows. We assume that $Ct_t_1 t_2$ uses are attached to "locations". The idea is that $i : Ct_t_1 t_2$ refers to some program text cast, 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 $f : Ct_t_1 t_2$ to refer to the proof term $f$ associated to a $Ct_t_1 t_2$ assumption.

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

A CHR propagation rule (R) states that if we find a constraint matching the lhs of a rule then we add the rhs to the store. Formally, we write $C \rightarrow_R C', \phi[\bar{d}]$ where $\bar{d} \in C$ such that $\phi(\bar{e}) = \bar{e}$. 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 $e \sim e'$. We write $C \rightarrow_R D'$ to denote an $n$ number of application of CHRs starting with the initial store $C$ yielding store $D'$. We write $e \sim e'$ 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 optimal uses of $Ct$ assumptions. A naive CHR-translation of transitivity such as

$$\text{(Trans)} : i : Ct_M a' b' \iff \text{cast}_i \rightarrow a' \quad \text{cast}_i \rightarrow a' \rightarrow \text{cast}_j \rightarrow a'$$

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 cast transformations.

In the above derivation, $\rightarrow^*_R$ represents n step derivation.

There is also another set of rules which exclusively manipulates $Ct$ assumptions. In rule (Trans1) 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 $Ct_M$ uses whereas propagation rules build the closure of all available $Ct$ assumptions. The following example stresses the importance of propagation rules.
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 \( C = \{ f_1 : C t a_1 \cdot b_1, \ldots, f_n : C t a_n \cdot b_n \} \) and \( i : \cdot C t a, b, C \Rightarrow^* D' \) and \( \text{cast} \Rightarrow^* e \) such that the \( C t \) derivation is good. Then, \( f : C t a \Leftrightarrow C \) such that \( f \) and \( e \) are equivalent.

Lemma 5 (Complete CHR Construction) Let \( C = \{ f_1 : C t a_1 \cdot b_1, \ldots, f_n : C t a_n \cdot b_n \} \) such that \( f : C t a \Leftrightarrow C \). Then, \( i : \cdot C t a, b, C \Rightarrow^* D' \) such that \( \text{cast} \Rightarrow^* e \) and \( f \) and \( e \) are equivalent.

Lemma 6 (Sound Term Construction) Let \( C = \{ f_1 : C t a_1 \cdot b_1, \ldots, f_n : C t a_n \cdot b_n \} \), \( i : \cdot C t a, b, C \Rightarrow^* D_1 \) and \( \text{cast} \Rightarrow^* e_1 \) and \( i : \cdot C t a, b, C \Rightarrow^* D_2 \), and \( \text{cast} \Rightarrow^* e_2 \), such that both \( C t \) 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 backtrack. See Examples 12 and 10. To obtain a decidable proof method we yet need to rule out certain \( C t \) derivations. E.g., consider

\[
\begin{array}{l}
g : C t a, b, h : C t b \Leftrightarrow C t a_i, b_i : C t M \Leftrightarrow D' \\
\Rightarrow^*_{\text{Trans1}} g : C t a, b, h : C t b \Leftrightarrow C t a_i, b_i : C t M \Leftrightarrow D'
\end{array}
\]

Unfortunately, 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” \( C t \) assumptions of the form \( g : C t a \cdot (a, b) \). Such assumptions must result from invalid \( \text{GRDT} \) definitions which we generally rule out.

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

\[
\text{data } \text{ErkJH} \ a \ = \ \text{forall } b. (\text{Ct} \ [b], \ \text{Ct} \ [b] \ a) \rightarrow \ \text{LJH} \ a
\]
\[
f_{\text{LJH}} : \ \text{ErkJH} \ a \rightarrow a
\]
\[
f_{\text{LJH}} \ (\text{LJH} \ x) = \ \text{cast} \ ((\text{cast} \ \text{tail}) \ (\text{cast} \ x))
\]

In a first step, we translate data types and patterns according to Figure 3 and replace all occurrences of \( \text{cast} \) in the program text by \( \text{castm} \) where each \( \text{castm} \) occurrence is attached to distinct locations.

\[
\text{data } \text{ErkJH'} \ a \ = \ \text{forall } b. \ \text{LJH'} \ a \ (\text{E} \ a \ [b])
\]
\[
f_{\text{LJH'}} : \ \text{ErkJH'} \ a \rightarrow a
\]
\[
f_{\text{LJH'}} \ (\text{LJH'} \ x) = \ \text{castm} \ ((\text{cast} \ \text{tail}) \ (\text{cast} \ x))
\]

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

\[
t = \text{Erank} \ a \rightarrow a, a = \text{Sk} \ b, b = \text{Sk} \ a,
\]
\[
\begin{align*}
&g : \text{Ct} \ [b], h : \text{Ct} \ [b] \ a \supset (1 : \text{CtM} \ a, b, b) = a, \\
&2 : \text{CtM} \ a, b, a = \alpha_1 \rightarrow \alpha_2, \\
&3 : \text{CtM} \ a, b, a = \alpha_3, \\
&b_1 = b_3 \rightarrow a_1)
\end{align*}
\]

(1)

Annotation \( f_{JH} : \text{ErkJH} \ a \rightarrow a \) implies \( f_{JH} : \forall a. \text{ErkJH} \ a \rightarrow a \). Hence, we substitute \( a \) by the skolem constructor \( \text{Sk} \). Similarly, we substitute \( b \) by \( \text{Sk} \). Each \( \text{castm} \) expression gives rise to \( i : \text{CtM} \ a \ b \) where \( \text{castm} \ a \ b \rightarrow b \). To each \( \text{Ct} \) 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 \( \rightarrow \) to state that under the \( \text{Ct} \) assumptions we can derive the \( \text{CtM} \) uses.

The constraint in (1) represents all possible typing derivations. We simply solve this constraint by applying CHRs defined in Figure 4 until all \( \text{CtM} \) uses have been resolved. Thus, all locations in the function body referring to proof terms are defined in terms of proof terms attached to \( \text{Ct} \) assumptions. In general, we solve \( \text{CtM} \ a \ b \rightarrow b \) by running \( \text{CtM} \ a \ b \rightarrow b \) and \( \text{CtM} \ b \ a \ b \rightarrow b \) and check that \( b' \) and \( b' \) 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 \( t = \text{Erank} \ a \rightarrow a, a = \text{Sk} \ b, b = \text{Sk} \ a, g : \text{Ct} \ [b], h : \text{Ct} \ [b] \ a \) (2) is immediately final. Consider,

\[
t = \text{Erank} \ a \rightarrow a, a = \text{Sk} \ b, b = \text{Sk} \ a, g : \text{Ct} \ [b], h : \text{Ct} \ [b] \ a, \text{CtM} \ a, b, a = \alpha_1 \rightarrow \alpha_2, \\
2 : \text{CtM} \ a, b, a = \alpha_1 \rightarrow \alpha_2, \\
3 : \text{CtM} \ a, b, a = \alpha_3, \\
b_1 = b_3 \rightarrow a_1, 1 : \text{CtM} \ a, a,
\]

(2)

\[
f_{\text{LJH}} \ (\text{LJH} \ x) = \ \text{castm} \ ((\text{cast} \ \text{tail}) \ (\text{cast} \ x))
\]

(3)

The constraint in (1) represents all possible typing derivations. We simply solve this constraint by applying CHR rules defined in Figure 4 until all \( \text{CtM} \) uses have been resolved. Thus, all locations in the function body referring to proof terms are defined in terms of proof terms attached to \( \text{Ct} \) assumptions. In general, we solve \( \text{CtM} \ a \ b \rightarrow b \) by running \( \text{CtM} \ a \ b \rightarrow b \) and \( \text{CtM} \ b \ a \ b \rightarrow b \) and check that \( b' \) and \( b' \) are logically equivalent (modulo variables in the initial store). We refer the interested reader to [23] for more details.

Note that we simultaneously transform constraints and program text. Constraints involved in rule applications are underlined. Silently, we extend \( e' \) \( \rightarrow e'' \) to \( e' \) \( \rightarrow e'' \) where \( e' \) 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 (H) instead of (Trans1).

\[ t = \text{Erk} \ a \rightarrow a, a = \text{Sk1} \ b, b = \text{Sk2} \ a, g : \text{Ct} \ a \ [b], \]
\[ h : \text{Ct} \ [b] \ a, 1 : \text{CtM} \ a_1 \ b_1 \ b_1 = a, \]
\[ 1 : \text{CtM} \ a_2 \ (a_2, a_2 = \text{Sk1} \ b, b = \text{Sk2} \ a, g : \text{Ct} \ a \ [b], \]
\[ h : \text{Ct} \ [b] \ a, 1 : \text{CtM} \ a_3 \ b_3, a_3 = a, b_3 = b_3 \rightarrow a_1, 1 : \text{CtM} \ a_1, \]
\[ t = \text{Erk} \ a \rightarrow a, a = \text{Sk1} \ b, b = \text{Sk2} \ a, g : \text{Ct} \ a \ [b], \]
\[ h : \text{Ct} \ [b] \ a, 1 : \text{CtM} \ a_2 \ (a_2, a_2 = \text{Sk1} \ b, b = \text{Sk2} \ a, g : \text{Ct} \ a \ [b], \]
\[ h : \text{Ct} \ [b] \ a, a_3 = a_2 \rightarrow \text{Sk1} \ b, b_3 = b_3 \rightarrow a_1, 1 : \text{CtM} \ a_1, \]
\[ \equiv False \]

Note that skolem variable Sk1 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.

\[
\begin{align*}
&\exists \lambda' \ (LJ'_x : \text{E} \ (g, h)) = \\
&\text{let castm}_0 \ g \ x = \text{castm}_5 \ (g \ \text{castm}_4 \ x) \\
&\text{castm}_2 = h \\
&\text{castm}_3 = x \\
&\text{castm}_3 = x \\
&\text{in castm}_5 \ ((\text{castm}_2 \ \text{tail}) \ (\text{castm}_3 \ x))
\end{align*}
\]

8. RELATED WORK

Our systematic translation method is inspired by the work by Baars and Swierstra [1], Chen, Zhu and Xi [2], Hinze and Chemy [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 \(\text{EQ} \ a \ b\) 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 \(\text{E} \ t \ t\) represent the identity to ensure preservation of the semantics of programs (see Definition 1). On the other hand, the \(\text{EQ}\) representation faces problems when trying to manipulate proof terms. E.g., there are situations where we need to “decompose” a value of type \(\text{EQ} \ a \ b\) \((c, d)\) into a value of type \(\text{EQ} \ a \ c\) 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.

Kislev [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 definition- alization 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 (TCT). 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 TCT (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

3Examples 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 has been recently added to Haskell. Implementations are available in the latest release of GHC [6] and Chameleon [25] (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 may 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

http://www.haskell.org/ghc/.
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

$$\mathcal{V} = \mathbf{W} \bot + \mathcal{V} \rightarrow \mathcal{V}.$$  

The meaning function on terms is as follows:

$$[x][\eta] = \eta(x)$$

$$[\lambda x.e][\eta] = \lambda v. [e][\eta[v := v]]$$

$$[e \prime e'][\eta] = \begin{cases} [e][\eta] \land [e'][\eta] \neq \mathbf{W} & \text{if } [e][\eta] \land [e'][\eta] \neq \mathbf{W} \\ \mathbf{W} & \text{else} \end{cases}$$

$$[\text{let } x = e \text{ in } e'][\eta] = \begin{cases} [e][\eta] & \text{if } [e][\eta] \neq \mathbf{W} \\ [e'][\eta] & \text{else} \end{cases}$$

Note that the above semantics is call-by-value.

B. PROOFS

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

First, we introduce an auxiliary 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 $(C t t' \in C' \land C t t' \in C')$. We call $C'$ the “$\sim C$” equivalent of $C$; and $C$ the “$\sim Eq$” equivalent of $C$.

**Lemma 8** Let $P_x$ be a full and faithful type class theory. Let $C$ be a set of equality constraints and $C'$ its “$\sim C$” equivalent. We have $C \vdash t \sim t$ iff $P_x \vdash C \supset (C t t', C t t')$.

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

- **Case:**
  $$t = t' \in C$$
  $$\frac{C \vdash t \sim t}{C \vdash C t t'}$$

  Because we have $t = t' \in C$, we know $C t t' \in C'$ and $C t t' \in C'$. Thus $P_x \vdash C \supset (C t t', C t t')$.

- **Case:**
  $$C \vdash t \sim t_1 \quad C \vdash t \sim t_2 \quad C \vdash t \sim t_3$$
  $$\frac{C \vdash C t t'}{C \vdash C t t'}$$

  By induction, we have

  $$P_x \vdash C \supset (C t t_1, C t t_2, C t t_3)$$

By the type class instance

$$\forall a_1, a_3, (C t a_1 a_3 \leftrightarrow \exists a_2. (C t a_1 a_2 \land C t a_2 a_3))$$

We conclude

$$P_x \vdash C \supset (C t t_1, C t t_2, C t t_3, C t t_3)$$

Other cases are similar. (Direction \(\Leftarrow\)).

**Case:** Suppose the type class instance

$$\forall a_1, a_3, (C t a_1 a_3 \leftrightarrow \exists a_2. (C t a_1 a_2 \land C t a_2 a_3))$$

is applied. Then we have

$$P_x \vdash C \supset C t t$$

We also have

$$C \vdash t \sim t$$

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

- **Case (Eq):**
  $$\frac{C, \Gamma \vdash C t t' \quad C \vdash t \sim t}{C, \Gamma \vdash C t t'}$$

  By the induction hypothesis, we have

  $$C', \Gamma \vdash C t t' \sim t \quad (1)$$

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

**Lemma 9** $C, \Gamma \vdash \text{cast } t \rightarrow t'$ iff $P_x \vdash C \supset C t t'$

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 $P_x$ 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 $C, \Gamma \vdash^{C e} t \iff C, \Gamma \vdash^{C e'} t$ where $C'$ is the “$\sim C$” equivalent of $C$.

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

- **Case (E):**
  $$\frac{C, \Gamma \vdash C t t' \quad C \vdash t \sim t}{C, \Gamma \vdash C t t'}$$

By the induction hypothesis, we have

$$C', \Gamma \vdash t \sim t \quad (1)$$
Also by Lemma 8 and $C \vdash^e t = t'$ we have
$$P_e \models C \supset (C t t', C t' t) \;
(2)$$
From (1) and (2), we conclude that
$$\begin{array}{l}
C', \Gamma \vdash^T (cast e') : t' \\
\end{array}$$
W.l.o.g. We can assume $e' \equiv (cast e'')$. Thus we obtain
$$\begin{array}{l}
C', \Gamma \vdash^T ((cast \circ cast) e'') : t' \\
\end{array}$$
We assume $C', \Gamma \vdash^T e'' : t''$. In the above case, the first cast
is of type $t \rightarrow t'$ and the second $t' \rightarrow t$. Thus by Lemma 9,
we know that $C t t'$ and $C t' t'$ can be derived from the context.
By the (Trans) type class instance, we can derive $C t t'$. Then by Lemma 9, we know there exists a cast of
type $t'' \rightarrow t'$. After replacing the cast composition $cast \circ cast$
in the above judgement by the new cast, we obtain
$$\begin{array}{l}
C', \Gamma \vdash^T (cast e') : t' \\
\end{array}$$
This is equivalent to
$$\begin{array}{l}
C', \Gamma \vdash^T e' : t' \\
\end{array}$$
\text{o Case (App)}:
$$\begin{array}{l}
C, \Gamma \vdash^{G_c} e_1 : t_2 \rightarrow t \\
C, \Gamma \vdash^{G_c} e_2 : t_2 \\
\vdash C, \Gamma \vdash^{G_c} (e_1 e_2) : t \\
\end{array}$$
By the induction hypothesis, we have
$$\begin{array}{l}
C', \Gamma \vdash^T e_1 : t_2 \rightarrow t \\
C', \Gamma \vdash^T e_2 : t_2 \\
\vdash C', \Gamma \vdash^T (e_1 e_2) : t \\
\end{array}$$
By application of rule (App), we obtain
$$\begin{array}{l}
C', \Gamma \vdash^T (cast (e_1 e_2)) : t \\
\end{array}$$
Note that we always have $C \vdash^e t = t$. Thus we conclude
$$\begin{array}{l}
C', \Gamma \vdash^T (cast (e_1 e_2)) : t \\
\end{array}$$

Other cases are similar.

\text{(Direction $\Rightarrow$): 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 $e' \equiv (cast e'').$}

\text{o $e'' = x$}

$$\begin{array}{l}
C', \Gamma \vdash^T cast : t \rightarrow t' \\
C', \Gamma \vdash^T e'' : t \\
\vdash C', \Gamma \vdash^T (cast e'') : t' \\
\end{array}$$

Because $e'' = x$, then $[[e'']] = e''$. Therefore, we have
$$\begin{array}{l}
C, \Gamma \vdash^{G_c} [e''] : t \\
\end{array}$$
By $C', \Gamma \vdash^T cast : t \rightarrow t'$ and Lemma 9, we obtain
$$\begin{array}{l}
P_e \models C' \supset (C t t', C t' t) \\
\end{array}$$
Together with Lemma 8, we have
$$\begin{array}{l}
C \vdash^e t = t' \\
\end{array}$$
By (1), (2) and rule (Eq), we conclude
$$\begin{array}{l}
C, \Gamma \vdash^{G_c} [e''] : t' \\
\end{array}$$
Because $[[cast e'']] = [e'']$, then we have
$$\begin{array}{l}
C, \Gamma \vdash^{G_c} [cast e''] : t' \\
\end{array}$$
This is equivalent to
$$\begin{array}{l}
C, \Gamma \vdash^{G_c} [e'] : t' \\
\end{array}$$
\text{o $e'' = \lambda x. e'''$}

$$\begin{array}{l}
C', \Gamma \vdash^T cast : t \rightarrow t' \\
\vdash C', \Gamma \vdash^T e'' = : t_2 \\
C', \Gamma \vdash^T e''' : t_2 \\
\vdash C', \Gamma \vdash^T (cast e'') : t' \\
\end{array}$$
In the above derivation $t = t_1 \rightarrow t_2$. By the induction hypothesis, we have
$$\begin{array}{l}
C, \Gamma \vdash^{G_c} [e'''] : t_2 \\
\end{array}$$
By applying the (Abs) rule, we obtain
$$\begin{array}{l}
C, \Gamma \vdash^{G_c} [e'''] : t \\
\end{array}$$
By $C', \Gamma \vdash^T cast : t \rightarrow t'$ and Lemma 9, we obtain
$$\begin{array}{l}
P_e \models C' \supset (C t t', C t' t) \\
\end{array}$$
Together with Lemma 8, we have
$$\begin{array}{l}
C \vdash^e t = t' \\
\end{array}$$
By (1), (2) and rule (Eq), we conclude
$$\begin{array}{l}
C, \Gamma \vdash^{G_c} [e'''] : t' \\
\end{array}$$
Because $[[cast e'']] = [e''']$, then we have
$$\begin{array}{l}
C, \Gamma \vdash^{G_c} [cast e''] : t' \\
\end{array}$$
This is equivalent to
$$\begin{array}{l}
C, \Gamma \vdash^{G_c} [e'] : t' \\
\end{array}$$
Other cases are similar. □
B.2 Proof of Lemma 2 (Well-Typed)

Our assumptions are $L = \{ f_1 : C t a_1 b_1, ..., f_n : C t a_n b_n \}$ and $\Gamma$ such that $C \to \Gamma$ and $f : C t a b \leftrightarrow C$ is valid. Then $\Gamma \vdash f : a \to b$.

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

- **Case (Id):**
  
  $\lambda x.x : C t a a \leftrightarrow \text{True}$
  We know that $\Gamma = \emptyset$. Thus we conclude $\Gamma \vdash \lambda x.x : a \to a$.

- **Case (Var):**
  
  $f : C t a b \leftrightarrow f : C t a b$
  We know that $\Gamma = \{ f : a \to b \}$. Thus we conclude $\Gamma \vdash f : a \to b$.

- **Case (Trans):**
  
  $f = \lambda g, \lambda x. f_2 \circ (g (f_1 x))$
  $f : C t a_1 a_3 \leftrightarrow f_1 : C t a_1 a_2, f_2 : C t a_2 a_3$
  We know that $\Gamma = \{ f_1 : a_1 \to a_2, f_2 : a_2 \to a_3 \}$. Thus by typing derivation we can easily conclude $\Gamma \vdash f : a_1 \to a_3$.

- **Case (Arrow):** Similar to (Trans).

- **Case ($\circ$):**
  
  $f : C t a b \leftrightarrow f_1 : C t c_1 ..., f_n : C t c_n, f_i : c_i \leftrightarrow F_i$
  $F \supseteq F_i$ for $i = 1, ..., n$
  $f : C t a b \leftrightarrow F$
  By induction, we have $\bigcup_i \Gamma_i \vdash f : a \to b$. Because $\bigcup_i \Gamma_i \subseteq \Gamma$ derived from $F = F_i$, then we conclude $\Gamma \vdash f : a \to b$.

\[\square\]

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

Theorem 2 follows directly from the following more general lemma.

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

**PROOF.** The proof proceeds by induction on derivations.

- **Case ($\exists \mathbf{K}$):**
  
  $(K : \forall \alpha, \beta t \to E t_1 t'_1 \to \ldots \to E t_n t'_n \to T \alpha)$
  $C, \Gamma \vdash^T e : [\tilde{\alpha}/\tilde{\beta}] t \to e'$
  $P_x \vdash C \supset (g, h) : [\tilde{\alpha}/\tilde{\beta}](C t t_1 t'_1, C t t_1 t'_1, ..., C t t_n t'_n, C t t_n t'_n)$$
  \rightarrow C, \Gamma \vdash^E E (g, h_1)(g, h_n)$

  By the induction hypothesis, we have
  
  $\Gamma \cup \Gamma' \vdash^E e' : [\tilde{\alpha}/\tilde{\beta}] t$ (1)

  We also have
  
  $K' : \forall \alpha, \beta t \to E t_1 t'_1 \to \ldots \to E t_n t'_n \to T \alpha$ (2)

  Note that here we assume $g, h_i \notin \Gamma$. Hence by Lemma 2, we have
  
  $\Gamma \cup \Gamma' \vdash^E g_i : t_i \to t'_i$ and $\Gamma \cup \Gamma' \vdash^E h_i : t_i' \to t_i$

  where $i = 1, ..., n$

  Thus we can obtain that
  
  $\Gamma \cup \Gamma' \vdash^E E (g_1, h_1)(g_1, h_n) : T \tilde{t}$ (3)

  From (1),(2),(3) and rule (K), we conclude
  
  $\Gamma \cup \Gamma' \vdash^E K' e' : E (g_1, h_1)(g_1, h_n) : T \tilde{t}$

  - **Case (Reduce):**
    
    $D \subseteq C \vdash C, \Gamma \vdash^T f : C t t_1 t_2 \vdash D$
    
    Given $D \subseteq C \vdash C, \Gamma \vdash^T f : C t t_1 t_2 \vdash D$, W.l.o.g we assume $f \notin \Gamma$.
    
    Thus we conclude by Lemma 2
    
    $\Gamma \cup \Gamma' \vdash^E f : t_1 \to t_2$

  - **Case ($\exists \mathbf{P}$):**
    
    $p : t_1 \vdash \forall \beta. (D \vdash \exists \beta. C t t_1 t_2) \cap f_0 (C, \Gamma, t_2) = 0$
    
    $C \land D, \Gamma \vdash^T e : t_2 \to e'$
    
    By the induction hypothesis, we have
    
    $C \land \Gamma \cup \Gamma_C \cup \Gamma_D \vdash^T e' : t_2$

    where $C \to \Gamma_C$ and $D \to \Gamma_D$.

    Also by Lemma 12 (see below), we have $p' \vdash \forall \beta. (\Gamma_p \cup \Gamma_D)$. Thus we conclude
    
    $\Gamma \cup \Gamma_C \vdash^E p' \to e' : t_1 \to t_2$

    - Other cases are standard. \[\square\]

**Lemma 12** Given $p : t_1 \vdash \forall \beta. (D \vdash \exists \beta. C t t_1 t_2)$ then $p' \vdash \forall \beta. \Gamma_p \cup \Gamma_D \to \Gamma$ where $D \to \Gamma$.

**PROOF.** Standard by induction on derivation. \[\square\]
B.4 Proof of Theorem 3 (TCET to ET Completess)

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 $C, \Gamma \vdash^T e : t \Rightarrow e'$ for some $e'$.

Proof. The proof is done by construction of $e'$.

Case (Reduce):

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

Given all the types are decomposable, by Lemma 3, we know $f : Ct \ t_1 \ t_2 \leftrightarrow C$ for some $f$ if $P_i \vdash C \supset Ct \ t_1 \ t_2$. 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 $C = \{f_i : Ct \ a_i b_i, \ldots, f_n : Ct \ a_n b_n\}$ and $i : Ct M a b, C \Rightarrow^* D'$ and castm$_i \sim^e$ 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 ($\forall E$) with rules (Id), (Var), (Arrow) and (Pair). We also combine ($\exists E$) with (Trans).

Case the rule applied is (Id):

$$i : Ct M a b, C \Rightarrow a = b, C \Rightarrow^* D'$$

$$\text{castm}_i \sim^e \lambda x. x$$

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

$$\lambda x. x : Ct \ a a \leftrightarrow \text{True}.$$}

Case the rule applied is (Trans1):

$$i : Ct M a b, C \Rightarrow a = a, j : Ct M b g, C \Rightarrow^* D'$$

$$\text{castm}_i \sim^e \text{castm}_j \circ g$$

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

$$f = \text{castm}_j \circ g$$

$$f : Ct a b \leftrightarrow g : Ct a b, \text{castm}_j : Ct b g$$

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

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

Case the rule applied is (Arrow):

$$i : Ct M (a_1 \rightarrow a_2) (b_1 \rightarrow b_2), C \Rightarrow$$

$$\text{castm}_i \sim^e \lambda g, \lambda x. (\text{castm}_i, x)$$

Also we have

$$f = \lambda g, \lambda x. \text{castm}_i (g (\text{castm}_i, x))$$

$$f : Ct (a_1 \rightarrow a_2) (b_1 \rightarrow b_2) \leftrightarrow \text{castm}_i : Ct b_1 a_1, \text{castm}_i : Ct a_2 b_2$$

Case (Arrow).

We have

$$f = \lambda g, \lambda x. f_1 (g (f_1 x))$$

$$f : Ct a_1 a_3 \leftrightarrow f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3$$

We further have

$$i : Ct M a_1 a_3, f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3$$

$$\Rightarrow \text{trans}_1 j : Ct a_2 a_3, f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3$$

$$\overset{\rightarrow}{\text{trans}_1} k : Ct M a_3 a_2, f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3$$

Then we have,

$$f_2 \circ f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3$$

$$\sim \lambda x. x \circ f_2 \circ f_1$$

Case (Arrow).

We have

$$f = \lambda g, \lambda x. f_1 (g (f_1 x))$$

$$f : Ct a_1 a_3 \leftrightarrow f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3$$
By induction,
\[
C, i_1 : \text{CtM} \ b_1 \ a_1 \Rightarrow^* D_1 \\
\text{cast}_{m_1} \overset{\sim}{\Rightarrow} f_1
\]
\[
C, i_2 : \text{CtM} \ a_2 \ b_2 \Rightarrow^* D_2 \\
\text{cast}_{m_2} \overset{\sim}{\Rightarrow} f_2
\]
Therefore
\[
i : \text{CtM} \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2), f_1 : \text{Ct} \ b_1 \ a_1, f_2 : \text{Ct} \ a_2 \ b_2 \\
\text{cast}_i \\
\Rightarrow^\text{Arrow}
\]
\[
i_1 : \text{CtM} \ b_1 \ a_1, i_2 : \text{CtM} \ a_2 \ b_2, f_1 : \text{Ct} \ b_1 \ a_1, f_2 : \text{Ct} \ a_2 \ b_2 \\
\lambda g. \lambda x. \text{cast}_{m_2} (g (\text{cast}_{m_1} \ x)) \\
\Rightarrow^\text{\sim}_e
\]
\[
f_1 : \text{Ct} \ b_1 \ a_1, f_2 : \text{Ct} \ a_2 \ b_2 \\
\lambda g. \lambda x. f_2 (g (f_1 \ x))
\]
\(\circ \ (\text{Pair})\) is similar to \(\text{Arrow}\).

\[\square\]

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

Our assumptions are: Let \(C = \{f_1 : \text{Ct} \ a_1 \ b_1, \ldots, f_n : \text{Ct} \ a_n \ b_n\}, i : \text{CtM} \ a \ b, C \Rightarrow^* D_1\) and \(\text{cast}_{m_i} \overset{\sim}{\Rightarrow} e_1\) and \(i : \text{CtM} \ a \ b, C \Rightarrow^* D_2\) and \(\text{cast}_{m_i} \overset{\sim}{\Rightarrow} e_2\) such that both CHR derivations are good. Then, \(e_1\) and \(e_2\) are equivalent.

**Proof.** Let \(f : \text{Ct} \ a \ b \Leftrightarrow C\), 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 \(e_2\). \(\square\)

**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 \(CtM\) 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 \(CtM\) applications.

Consider rule instance \((\text{Trans}1)\) \(g : \text{Ct} \ a \ b, i : \text{CtM} \ a' b' \Leftrightarrow g : \text{Ct} \ a \ b, a' = a', j : \text{CtM} \ b' b'\) and \(C\) such that \(g : \text{Ct} \ a \ b, i : \text{CtM} \ a' b'\) \(\in C\). Then \(C \Rightarrow^\text{\text{Trans}1} C - (i : \text{CtM} \ a' b') \Leftrightarrow a = a', (j : \text{CtM} \ b' b') \Leftrightarrow j\). We say that the **termination condition** is violated iff \(j \in J\).

Consider rule instance \((\text{Arrow})\) \(i : \text{CtM} \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2) \Leftrightarrow i_1 : \text{CtM} \ b_1 \ a_1, i_2 : \text{CtM} \ a_2 \ b_2\) such that \(i : \text{CtM} \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2)\) \(\in C\). Then, \(C \Rightarrow^\text{\text{Arrow}} C - (i : \text{CtM} \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2)) (i_1 : \text{CtM} \ b_1 \ a_1), (i_2 : \text{CtM} \ a_2 \ b_2)\). The justiﬁed CHR semantics for rule \((\text{Pair})\) is similar.

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

We impose an order among derivations. Let \(C = \{f_1 : \text{Ct} a_1 \ b_1, \ldots, f_n : \text{Ct} a_n \ b_n\}, i : \text{CtM} \ a \ b, C \Rightarrow^* D_1\) and \(\text{cast}_{m_i} \overset{\sim}{\Rightarrow} e_1\) and \(i : \text{CtM} \ a \ b, C \Rightarrow^* D_2\) and \(\text{cast}_{m_i} \overset{\sim}{\Rightarrow} e_2\) such that both CHR derivations are good. We say that \(i : \text{CtM} \ a \ b, C \Rightarrow^* D_1\) is shorter than \(i : \text{CtM} \ a \ b, C \Rightarrow^* D_2\) if 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 \(CtM\)s we compare the sum of the individual sizes of resulting expressions.

**Lemma 14** Let \(i : \text{CtM} \ a \ b, C \Rightarrow^* D\) be a good derivation. Then, \(\text{cast}_{m_i} \overset{\sim}{\Rightarrow} e\) 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.

\[
\begin{align*}
C & \Rightarrow^* D \\
\Rightarrow^\text{Trans1} & C_1, (g : \text{Ct} t_1 t_2)_{t_1}, (i : \text{CtM} t_1' t_2')_{t_1}, t_1 \notin L_1 \\
\Rightarrow^\text{Pair} & C_2, (g : \text{Ct} t_1 t_2)_{t_1}, (k : \text{CtM} t_1' t_2')_{t_2}, t_1 = t_2'
\end{align*}
\]

W.l.o.g., in the derivation steps between (1) and (2) we only apply CHR (translating (1) and (2)) rules which results from \((\text{Trans}1)\) and \((\text{Arrow})\) rules.

First, we show that only \((\text{Trans}1)\) or \((\text{H})\) rules could have been applied on \(j : \text{CtM} t_2 t_2'\) \(\in L_2\), or its successors. Assume the contrary, that is some \((\text{Pair})\) (or a similar type-constructor) rule has been applied on \(j : \text{CtM} t_2 t_2'\) \(\in L_1\).

Then,
\[
\begin{align*}
\cdots (g : \text{Ct} t_1 t_2)_{t_1} t_2 & = (t_3, t_4), t_4 = (t_5, t_6), \\
(j : \text{CtM} t_2 t_2')_{t_2} & = (t_7, t_8), (j : \text{CtM} t_2 t_2')_{t_2} = (t_9, t_{10}),
\end{align*}
\]

However, then we obtain a cycle among types. E.g., assume that \((j : \text{CtM} t_2 t_3)_{t_3} = \cdots\) equals \((k : \text{CtM} t_1' t_2')_{t_2}\). We find that \(t_7 = t_1', t_2 = (t_3, t_4), t_4' = (t_5, t_6), t_6' = t_7\) which implies \((g : \text{Ct} t_1 (t_7, t_8))_{t_1}\). Thus, we obtain a contradiction. Note that by assumption the type equations resulting from \(CtM\) constraints \((\text{Ct} a b) \text{ yields } a = b\) must be satisﬁable. Otherwise, the GRDT deﬁnition is invalid.

Hence, we only ﬁnd \((\text{Trans}1)\) or \((\text{H})\) 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. \(\square\)

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

17
PROOF. Follows immediately. Note that we disallow \( C_t \) assumptions of the form \( g : C_t a \ (a,b) \). Hence, any non-terminating derivation must violate the termination condition. □