A SYSTEMATIC TRANSLATION OF
GUARDED RECURSIVE DATA TYPES TO
EXISTENTIAL TYPES

WANG MENG
(B.Comp.(Hons.), NUS)

A THESIS SUBMITTED
FOR THE DEGREE OF MASTER OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE
2004
Acknowledgements

Many people have provided help during the course of this research. First and foremost, I would like to thank my supervisor Martin Sulzmann whom I value not only as an advisor but also a friend. I benefited enormously from his knowledge and expertise. It is really a challenge and privilege to be in his research group. I would also like to thank my co-supervisor Associate Professor Chin Wei Ngan who provided constant support and advise. Without him, the completion of this project would not have been possible. I also have to thank Associate Professor Khoo Siau Cheng for his hospitality and kindness.

Thanks to all my colleagues sitting in PLS (Programming Language and System) laboratory II of SoC for the memorable time I spent with them.
## Contents

Acknowledgements

Summary

1 Introduction

2 Background

   2.1 Existential Types

   2.2 Type Classes

      2.2.1 Multi-parameter Type Classes

      2.2.2 Type Classes with Existential Types

   2.3 Guarded Recursive Data Types

   2.4 Formal System

3 Translating GRDTs to Type Classes with Existential Types

4 Translating GRDTs to Existential Types

   4.1 Decidable Proof Construction Method
## Contents

4.2 Combing Proof Term Construction and Building Typing Derivations 31

5 Heuristics 38

6 Further Examples 45

6.1 Transitivity Example 45
6.2 Sheard and Pasalic Append Example 49
6.3 Hinze and Cheney Trie Example 58

7 Conclusion 71

A Semantics of Expressions 75

B Termination of CHRs 77

C Proofs 81

C.1 Proof of Theorem 1 81
C.2 Proof of Lemma 1 88
C.3 Proof of Lemma 2 89
C.4 Proof of Theorem 2 90
C.5 Proof of Theorem 3 92
C.6 Proof of Lemma 3 92
C.7 Proof of Lemma 4 93
C.8 Proof of Lemma 5 95
C.9 Proof of Lemma 6 96
C.10 Proof of Theorem 4 96
C.11 Proof of Theorem 5 98
Guarded recursive data types (GRDTs) are a new language feature which allows to type check the different branches of case expressions under different type assumptions. We observe that GRDTs are pretty close in their typing behavior to type classes with existential types (TCET). We give a translation scheme from GRDTs to TCET. The translation to TCET might be ambiguous in the sense that common implementations such as the Glasgow Haskell Compiler (GHC) fail to accept the translated program. Hence, we provide for another translation from TCET to existential types (ET) which is accepted by GHC. To achieve this goal we combine an existing constraint solving procedure with a novel proof term construction method.
List of Figures

2.1 General Typing Rules ........................................... 15
4.1 Proof Term Construction Rules .............................. 21
4.2 Type-Directed Translation (Part I) ...................... 25
4.3 Type-Directed Translation (Part II) .................... 26
4.4 CHR-based Proof Term Construction .................... 27
4.5 Pre-term and Formula generation (Part I) ............. 32
4.6 Pre-term and Formula generation (Part II) ............ 33
5.1 CHR-based Proof Term Construction (Part I) .......... 43
5.2 CHR-based Proof Term Construction (Part II) .......... 44
Introduction

Guarded recursive data types (GRDTs) [22] introduced by Xi, Chen and Chen are a new language feature which allows for type checking of more programs. The basic idea is to use different type assumptions for each branch of a case expression. There exist several variations of GRDTs 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 [15]. In a recent work [18], the authors proposed another variation combining GRDTs and type classes. Here, we consider GRDTs as introduced by Xi, Chen and Chen.

Example 1 Consider a evaluator for a simple arithmetic language

data Term a = (a=Int) => Lit Int
  | (a=Int) => Inc (Term Int)
  | (a=Bool) => IsZ (Term Int)
  | If (Term Bool) (Term a) (Term a)
  | forall b c. (a=(b,c)) => Pair (Term b) (Term c)
  | forall b c. (a=b) => Fst (Term (b,c))
  | forall b c. (a=c) => Snd (Term (b,c))
eval :: Term a -> a
eval (Lit i) = i
eval (Inc t) = eval t + 1
eval (IsZ t) = eval t == 0
eval (If b t e) = if eval b then eval t else eval e
eval (Pair x y) = (eval x, eval y)
eval (Fst t) = fst (eval t)
eval (Snd t) = snd (eval t)

The data type definition introduces constructors belonging to data type \textit{Term} \textit{a}. The novelty is that the type is refined for each constructor. For example, in case of constructor \texttt{Inc} we refine the type to \textit{Term Int} whereas in case of \texttt{Pair} we refine the type to \textit{Term} \((b, c)\) for some \(b, c\). We present type refinement in terms of equations such as \(a = (b, c)\). Note that some presentations [4] write \texttt{Inc (Term Int) with (a=Int) instead of (a=Int) => Inc (Term Int).} We chose the latter to stay closer to Haskell syntax [8]. More importantly, we make use of these additional type assumptions in case of pattern matching. Consider the function definition where in the second clause we temporarily add \(a = \text{Int}\) to our assumptions (assuming that \(t\) has type \textit{Int}). Thus, we can verify that the \texttt{eval t + 1} has type \(a\). A similar observation applies to other clauses. Hence, function \texttt{eval} is type correct.

A maybe surprising observation is that GRDTs can almost trivially be encoded in terms of multi-parameter type classes with existential types (TCETs). We introduce a type class \texttt{Ct a b} to convert a term of type \(a\) into a term of type \(b\). Operationally, the conversion performs the identify operation for all monomorphic instances derivable w.r.t. the following rules.
We translate GRDT programs by replacing each equation $t_1 = t_2$ in a data type definition by $\text{Ct } t_1 \rightarrow t_2$ and $\text{Ct } t_2 \rightarrow t_1$ Additionally, we apply $\text{cast}$ to all sub-expressions.

**Example 2** Here is the translation of Example 1. (For simplicity, we only show 2 clauses ($\text{Inc}$ and $\text{Pair}$) here. The rest are similar.)

```haskell
data Term_H a = (Ct a Int, Ct Int a) => Inc_H (Term_H a) |
forall b c.(Ct a (b,c), Ct (b,c) a) =>
Pair_H (Term_H b,Term_H c)

eval_H :: Term_H a -> a
eval_H (Inc_H t) =
cast ((cast ((cast (+)) (cast ((cast eval_H) (cast t)))))) (cast 1)
eval_H (Pair_H x y) =
cast (cast ((cast eval_H) (cast x)),cast ((cast eval_H) (cast y)))
```

Note that we use function notation for addition. When typing the first clause we temporarily make use of $\text{Ct } a \text{ Int}$ and $\text{Ct } \text{Int} a$. By using the instance (Id), we can give type $a$ to $(\text{cast ((cast eval_H) (cast t))})$. We make use of instance (Arrow) to show that $\text{cast (+)}$ has type $a \rightarrow \text{Int} \rightarrow a$. Hence, $\text{cast ((cast ((cast (+)) (cast ((cast eval_H) (cast t)))))) (cast 1)}$ has type $a$. A similar reasoning applies to the second clause.
It is well-known how to translate TCET programs by means of the type-directed evidence-translation scheme [7]. The subtle point is that to apply this scheme we first need to provide a TCET type derivation. This task is by no means obvious considering the above instances and program. E.g., instance (Trans) is “non-terminating” unless we are able to guess the proper intermediate type. The program text \texttt{cast \ x} gives rise to the constraint \texttt{Ct a c} for some \texttt{c}. Hence, we need to guess for which \texttt{c} we can satisfy \texttt{Ct a c}. Note that the type inference for GRDTs is a hard problem [10, 17, 18]. Hence, it is not that surprising why type inference for the TCET program remains difficult. Our goal is to find a translation which is accepted by common Haskell implementations such as GHC [6].

**Example 3** Here is a translation of Example 1 which is accepted by GHC. We introduce a special data type \texttt{E} to represent equality assumption among types. E.g., we represent \texttt{a = Int} by \texttt{E a Int} where the associated value \texttt{E (g,h)} implies functions \texttt{g} and \texttt{h} to convert \texttt{a}'s to and from \texttt{Int}'s.

\begin{verbatim}
  data E a b = E (a->b,b->a)
  data Term_H' a = Inc_H' (Term_H' Int) (E a Int)
                  | forall b c. Pair_H' (Term_H' b,Term_H' c) (E a (b,c))

  eval_H' :: Term_H' a -> a
  eval_H' (Inc_H' t (E (g,h))) = let cast g' y z = h (g' (g y) z)
                                in (cast (+)) x 1
  eval_H' (Pair_H' (x,y) (E (g,h))) = h (eval_H' x,eval_H' y)
\end{verbatim}

Note that we explicitly construct the necessary casting functions. E.g., \texttt{cast} turns a function of type \texttt{Int \to Int \to Int} into a function of type \texttt{a \to Int \to a}. Operationally, \texttt{cast} represents the identity function. The above program makes use of existential types and is accepted by GHC.
Baars and Swierstra [1], Chen, Zhu and Xi [2], Hinze and Cheney [3] and Weirich [21] gave similar examples which show how to express GRDT-style behavior in terms of existing language features available in Haskell. Note that in [1, 3] equality is represented in terms of the following definition.

\[
\text{newtype EQ a b = EQ (forall f. f a -> f b)}
\]

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). However, we face 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. In contrast, our encoding of equality in terms of \(E\ a\ b\) allows for proof term manipulation. To ensure preservation of the semantics of programs we need to postulate that all values attached to monomorphic instances of \(E\ t\ t\) represent the identity.

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, 14, 21]. 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 Chapter 2 where we review related background. Chapter 3 provides for an intermediate translation from GRDTs and TCETs. Chapter 4 provides for a translation scheme from GRDTs to ETs. In Chapter 5, we describe a strategy to improve efficiency of our translation. After that, more realistic examples are given in Chapter 6. We conclude in Chapter 7. We refer to the Appendix for complete proofs of all theorems and lemmas stated.
Background

Throughout the paper we work with the following set of expressions and types.

Expressions \( e ::= K \mid x \mid \lambda x.e \mid e \ e \mid \text{case} \ e \ \text{of} \ [p_i \rightarrow e_i]_{i \in I} \)

Annotation \( an ::= e :: \sigma \)

Patterns \( p ::= x \mid (p, p) \mid K \ p \)

Types \( t ::= a \mid t \rightarrow t \mid T \ t \)

Type Schemes \( \sigma ::= t \mid \forall \bar{\alpha}.C \Rightarrow t \)

In this language, we have data constructors, variables, \( \lambda \) abstractions, applications and pattern matchings as expressions. For simplicity, we leave out let-definitions but may make use of them in examples. We write \( \bar{o} \) to denote a sequence of objects \( o_1, \ldots, o_n \) and \( \bar{o} : \bar{t} \) to denote \( o_1 : t_1, \ldots, o_n : t_n \). Constraints \( C \) consist of conjunctions of equality constraints \( t_1 = t_2 \). We often treat constraints as sets. E.g., we use “,” as a short-hand for Boolean conjunction. We assume that the reader is familiar with the concepts of substitution, unifiers, most general unifiers (m.g.u.) etc [11].

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 \( \exists_W F \) denote the formula \( \exists \alpha_1 \ldots \exists \alpha_n F \) where \( \{\alpha_1, \ldots, \alpha_n\} = fv(F) - W \). We refer to [16] for details. GRDT definitions such as
data Term a = (a=Int) => Inc (Term Int)
            | forall b c. (a=(b,c)) => Pair (Term b) (Term c)

imply constructors $\text{Inc} : \forall a. a = \text{Int} \Rightarrow \text{Term Int} \rightarrow \text{Term a}$ and $\text{Pair} : \forall a, b, c. a = (b, c) \Rightarrow \text{Term b} \rightarrow \text{Term c} \rightarrow \text{Term a}$. 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.

In the following sections, we introduce three related typing machineries namely Existential Types (ET), Type Classes with Existential Types (TCET) and Guarded Recursive Data Types (GRDT). They will be illustrated informally by examples before we compare their underlying type systems with GRDT’s.

## 2.1 Existential Types

Existential quantified types can be used in data type declarations in Haskell. Again, we will illustrate with examples (the material in this section is borrowed from GHC’s documentation [6]). Consider the following declaration:

data Foo = forall a. MkFoo a (a -> Bool)
            | Nil

The data type Foo has two constructors with types:

MkFoo :: forall a. a -> (a -> Bool) -> Foo
Nil  :: Foo

Notice that the type variable a in the type of MkFoo does not appear in the data type itself, which is plain Foo. For example, the following expression is fine:

[MkFoo 3 even, MkFoo ’c’ isUpper] :: [Foo]
Here, \((\text{MkFoo} \ 3 \ \text{even})\) packages an integer with a function even that maps an \text{Integer} to \text{Bool}; and \text{MkFoo} \ 'c' \ \text{isUpper} packages a character with a compatible function. These two things are each of type \text{Foo} and can be put in a list.

What can we do with a value of type \text{Foo}? In particular, what happens when we pattern-match on \text{MkFoo}?

\[
f (\text{MkFoo} \ \text{val} \ \text{fn}) = ???
\]

Since all we know about \text{val} and \text{fn} is that they are compatible, the only (useful) thing we can do with them is to apply \text{fn} to \text{val} to get a boolean. For example:

\[
f :: \text{Foo} \rightarrow \text{Bool}
f (\text{MkFoo} \ \text{val} \ \text{fn}) = \text{fn} \ \text{val}
\]

What this allows us to do is to package heterogenous values together with a bunch of functions that manipulate them, and then treat that collection of packages in a uniform manner.

### 2.2 Type Classes

\textit{Typeclasses} is the overloading mechanism in Haskell. It groups types into different classes which allow the programmer to define relations over types. For single-parameter type classes, the relation simply states set membership. The types from a class share overloaded behaviors which are different for each type (in fact the behavior is sometimes undefined, or error). We call them \textit{class methods}. Let’s consider the \textit{Eq} class, the declaration

\[
\text{class Eq a where}
\]

\[
(==) :: a \rightarrow a \rightarrow \text{Bool}
\]
states that every type \( a \) in type class \( Eq \) has an overloaded function \((==)\) compares two values of the same type for equality. Members of the class and the specific overloaded behavior is declared by instances. For example, \( \text{Integer} \) is in \( Eq \):

```haskell
instance Eq Integer where
    x == y = x 'integerEq' y
```

This instance states that \( \text{==} \) for \( \text{Int} \) is \text{integerEq} which is a built-in primitive. Let's look at another instance:

```haskell
instance (Eq a) => Eq (List a) where
    Nil == Nil = True
    (Cons x lx) == (Cons y ly) = (x==x) && (lx==ly)
    _ == _ = False
```

This instance says that we can compare lists of \( a \)'s for equality as long as we know how to compare \( a \)'s for equality. Note that the two usages of \((==)\) in the body of the second clause are different. The former compares two elements whereas the latter compares two lists.

Note that the \( \text{==} \) function has a constrained type:

\[
(==) :: \text{Eq a} \Rightarrow a \rightarrow a \rightarrow \text{Bool}
\]

which has a constraint component \( \text{Eq a} \) and a type component \( a \rightarrow a \rightarrow \text{Bool} \). As a result, \( \text{==} \) can only be used on values with types that are in \( Eq \).

### 2.2.1 Multi-parameter Type Classes

One addition type classes feature is \textit{multi-parameter type classes} which allows multiple class parameters. One example will be the \( Ct a b \) class first mentioned in Chapter 1 and used throughout the thesis.

```haskell
class Ct a b where cast :: a -> b
```
It defines a relation which says a value of type $a$ can be coerced into a value of type $b$. For example, a type $a$ can be casted into itself:

```haskell
instance Ct a a where cast x = x
```

Also the relation is transitive:

```haskell
instance (Ct a1 a2, Ct a2 a3) => Ct a1 a3 where
  cast a1 = cast (cast a1)
```

### 2.2.2 Type Classes with Existential Types

Type classes can be used as context to constrain data type constructors [12] and [18]. Consider an example:

```haskell
data Baz = forall a. Eq a => Baz1 a a 
          | forall b. Show b => Baz2 b (b -> b)
```

Similar to the constrained type for type class functions, the two constructors have the following types:

- $\text{Baz1} :: \forall a. \text{Eq } a \Rightarrow a \to a \to \text{Baz}$
- $\text{Baz2} :: \forall b. \text{Show } b \Rightarrow b \to (b \to b) \to \text{Baz}$

When pattern matching on $\text{Baz1}$ the matched values can be compared for equality, and when pattern matching on $\text{Baz2}$ the first matched value can be converted to a string (as well as applying the function to it). So this program is legal:

```haskell
f :: Baz -> String
f (Baz1 p q)  | p == q = "Yes"
              | otherwise = "No"
f (Baz2 v fn) = show (fn v)
```

Consider $\text{Term}_\text{H}$ in Example 2:
2.3 Guarded Recursive Data Types

The constructors have the types we’d expect:

\[\text{Inc}_H :: \forall a. (\text{Ct} \ a \ \text{Int}, \ \text{Ct} \ \text{Int} \ a) \Rightarrow \text{Term}_H a \rightarrow \text{Term}_H a\]

\[\text{Pair}_H :: \forall a \ b \ c. (\text{Ct} \ a \ (b,c), \ \text{Ct} \ (b,c) \ a) \Rightarrow (\text{Term}_H b, \text{Term}_H c) \rightarrow \text{Term}_H a\]

When pattern matching on \(\text{Inc}_H\) we can cast a term of type \(a\) to a term of type \(\text{Int}\) and vice versa, and similarly when pattern matching on \(\text{Pair}_H\) we can cast between \(a\) and \((b,c)\).

2.3 Guarded Recursive Data Types

In Chapter 1, we have seen an example of a type safe evaluator. In this section, we give another example which shows how GRDT can be used to simulate dependent types. The example is sequences of elements with the semantic property that the length of the sequence is encoded in its type. For instance, the \textit{append} function which append a sequence after the other will have type \(\text{Seq} \ a \ n \rightarrow \text{Seq} \ a \ m \rightarrow \text{Seq} \ a \ (n+m)\). In order to type such functions it is necessary to do arithmetic at the type level. The following program shows how to capture this specification.

\textbf{Example 4}

\begin{verbatim}
data \text{Z} = \text{Z}
data \text{S} \ n = \text{S} \ n

data \text{Sum} \ w \ x \ y = (\text{w} = \text{Z}, \text{x} = \text{y}) \Rightarrow \text{Base}
  | \forall m \ n. (\text{w} = \text{S} \ m, \text{y} = \text{S} \ n) \Rightarrow \text{Step} \ (\text{Sum} \ m \ x \ n)
\end{verbatim}
data Seq a n = n=Z => Nil
    | forall m. n=S m => Cons a (Seq a m)

append :: Sum n m p -> Seq a n -> Seq a m -> Seq a p
append Base Nil ys = ys
append (Step s) (Cons x xs) ys = Cons x (app s xs ys)

In this example, we encode arithmetic using data types. \( Z \) represents zero and \( S \ n \) represents the successor of \( n \). As you would expect \( S \ (S \ Z) \) represents number three. Now we can defined submission as a GRDT. \( Sum \ w \ x \ y \) carries the meaning that \( w + x = y \). When \( w \) is zero, we know \( x = y \), this equation is encoded by the constructor \( Base \). When \( w \) is the successor of \( m \) for some \( m \), we know that \( y \) is the successor of some \( n \) where \( n \) is the sum of \( m \) and \( x \). This is reflected in constructor \( Step \). We also can define a sequence with its length information included. When the sequence is empty, the length is zero. When we “Cons” an element to a sequence, the length increases by one.

Now we are ready to define a variant of append function which have type \( Sum \ n \ m \ p \rightarrow \text{Seq} \ a \ n \rightarrow \text{Seq} \ a \ m \rightarrow \text{Seq} \ a \ p \). This type carries the information that the length of the output list \( p \) is the sum of the two input lists’ \( (n \) and \( m) \). This property is enforced by the type of the first parameter of append. In the first clause, \( n \) is zero. Thus \( p \) is equal to \( m \). This agrees with the constraints which the first argument \( Base \) carries. For the second clause, suppose \( s \) has type \( Sum \ n' \ m' \ p' \), then we know from the recursive call that the length of \( xs \), \( ys \) and \( (\text{app} \ s \ xs \ ys) \) are \( n' \), \( m' \) and \( p' \) respectively. Because \( \text{Cons} \ a \ (\text{app} \ s \ xs \ ys) \) has length \( p \), we know that \( p=S \ p' \) from the constraint attached on \( \text{Cons} \). We also know the length of, \( (\text{Cons} \ a \ xs) \), \( n \) is equal to \( S \ n' \) and \( m = m' \). With all this, we can derive \( \text{Step} \ a \ s \) has type \( Sum \ n \ m \ p \). Thus the function is well typed.
### 2.4 Formal System

The rules in Figure 2.1 describing a general type system. We introduce judgments \( C, \Gamma \vdash e : t \) to denote that expression \( e \) has type \( t \) under constraint \( C \) and environment \( \Gamma \). 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 \((+): \text{Int} \to \text{Int} \to \text{Int}\), \(\text{fst}: \forall ab.(a, b) \to a\) etc. Note that \( C \supset t_1 = t_2 \) 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 \( p : t \vdash \forall \bar{b}.(D \mid \Gamma_p) \) which establishes the binding \( \Gamma_p \) of variables and accumulates constraints \( D \) attached to constructors in \( p \). In rule (P-K), we assume that there are no name clashes between variables \( \bar{b}_1 \) and \( \bar{b}_2 \). Constraint \( D \) arises from constructor uses in \( p \). Variables \( \bar{b} \) refer to all “existential” variables. Note that these variables become universally quantified when moving out the quantifier.

The ET system is a special case of the general type system found in Figure 2.1 where we take all the constraints \( C \) and \( D \) to be \textit{True}. The TCET and GRDT systems extend the general type system slightly. For TCET, we need to add in a rule to take care of class methods.

\[
\begin{array}{c}
\text{(M)} \\
(m : \forall \bar{a}.TC \bar{a} \Rightarrow t \quad \text{fv}(t) \subseteq \bar{a} \quad C \supset TC \bar{t}) \\
\hline
C, \Gamma \vdash^T m : [\bar{t}/\bar{a}]t
\end{array}
\]

where \( m \) is assumed to be a class method of type \( m : \forall \bar{a}.TC \bar{a} \Rightarrow t \).

GRDT system can also be seen as an extension of the general type system. It has a special rule (Eq).

\[
\begin{array}{c}
\text{(Eq)} \\
C, \Gamma \vdash^G e : t \quad C \supset t = t' \\
\hline
C, \Gamma \vdash^G e : t'
\end{array}
\]

This rule allow us to change the type of an expression.
### 2.4 Formal System

**Figure 2.1: General Typing Rules**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| (App) | $C, \Gamma \vdash e_2 : t_2$  
  $C, \Gamma \vdash e_1 : t_2 \rightarrow t$  
  $\Rightarrow C, \Gamma \vdash e_1 \ e_2 : t$ |
| (Abs) | $C, \Gamma \vdash x : t_1 \vdash e : t_2$  
  $\Rightarrow C, \Gamma \vdash \lambda x.e : t_1 \rightarrow t_2$ |
| (Var-x) | $(x : \forall \bar{a}.D \Rightarrow t) \in \Gamma$  
  $\Rightarrow C, \Gamma \vdash x : [t/a]D$  
  $\Rightarrow C, \Gamma \vdash x : [t/a]t$  
  $C, \Gamma \vdash e_1 : t_1$  
  $\Rightarrow C, \Gamma \vdash e : t_1$  
  $C, \Gamma \vdash e_1 \rightarrow e_2 : t_1 \rightarrow t_2$  
  $\Rightarrow C, \Gamma \vdash case \ e \ of \ [e_1 \rightarrow e_2]_{i \in I} : t_2$ |
| (Ann) | $C, \Gamma \vdash e : t$  
  $\Rightarrow C, \Gamma \vdash (e :: (D \Rightarrow t)) : t$ |
| (K) | $K : \forall \bar{a}, \bar{b}.D \Rightarrow t \rightarrow T \bar{a}$  
  $\Rightarrow p : t_1 \vdash \forall \bar{b}.(D \mid \Gamma_p)$  
  $C, \Gamma \vdash e : [\bar{t}]/\bar{a}, \bar{t}]/\bar{b}]t$  
  $\Rightarrow \bar{b} \cap \mathit{fv}(C, \Gamma, t_2) = \emptyset$  
  $\Rightarrow C, \Gamma \vdash e_1 : t_2$  
  $\Rightarrow C, \Gamma \vdash e : t_1 \rightarrow t_2$  
  $\Rightarrow C, \Gamma \vdash K e : T \bar{t}$  
  $\Rightarrow C, \Gamma \vdash p \rightarrow e : t_1 \rightarrow t_2$ |
| (Pat) | $C, \Gamma \vdash [\bar{t}]/\bar{a}, \bar{t}]/\bar{b}]D$  
  $\Rightarrow C \mid [\bar{t}]/\bar{a}, \bar{t}]/\bar{b}]D$  
  $\Rightarrow C \land D, \Gamma \cup \Gamma_p \vdash e : t_2$ |
| (P-Var) | $x : t \vdash^G (\mathbf{True} \mid \{x : t\})$ |
| (P-Pair) | $p_1 : t_1 \vdash \forall \bar{a}_{1}.(D_1 \mid \Gamma_{p_1})$  
  $p_2 : t_2 \vdash \forall \bar{b}_{2}.(D_2 \mid \Gamma_{p_2})$  
  $(p_1, p_2) : (t_1, t_2) \vdash^G \forall \bar{a}_{1}, \bar{b}_{2}.(D_1 \land D_2 \mid \Gamma_{p_1} \cup \Gamma_{p_2})$ |
| (P-K) | $K : \forall \bar{a}, \bar{b}.D \Rightarrow t \rightarrow T \bar{a}$  
  $\Rightarrow \bar{b} \cap \bar{a} = \emptyset$  
  $p : [\bar{t}/\bar{a}]t \vdash \forall \bar{b}'.(D' \mid \Gamma_p)$  
  $\Rightarrow K p : T \bar{t} \vdash \forall \bar{b}', \bar{b}.(D' \land [\bar{t}/\bar{a}]D \mid \Gamma_p)$ |
2.4 Formal System

Let’s consider the first clause of \( f \) in Example 1 again. According to rule (Pat), the pattern \((\text{Inc } t)\) provides additional type assumption \(a = \text{Int}\) which is used in typing of the body \(\text{eval } t + 1\). Note that because of this additional assumption, rule (Eq) is able to turn the type of \(\text{eval } t\) from \(a\) to \(\text{Int}\). Thus, the expression \(\text{eval } t + 1\) is well typed. Similarly, rule (Eq) also turns the type of \(\text{eval } t + 1\) to \(a\) which obeys the annotation.

**Example 5** Consider the following variation of Example 1

data Term a = (a=\text{Int}) => Inc (Term \text{Int})
g :: Term \text{Bool} \to b
g (\text{Inc } t) = \text{eval } 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 \(\text{Term} \text{Bool} \to b\) for any \(b\).

We rule out such programs by introducing a constructive entailment relation among equations.

\[
\begin{align*}
& t = t' \in C \quad C \vdash^e t = t' \\
& C \vdash^e t_1 = t_2 \quad C \vdash^e t_2 = t_3 \\
& C \vdash^e t_1 = t_3 \\
& C \vdash^e t_1 \rightarrow t_3 = t_2 \rightarrow t_4 \\
& C \vdash^e t_1 = t_3 \quad C \vdash^e t_4 \\
& C \vdash^e t_i = t'_i \quad \text{for } i = 1, \ldots, n \\
& C \vdash^e T t_1 \ldots t_n = T t'_1 \ldots t'_n
\end{align*}
\]

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

\[
\begin{array}{c}
(C, \Gamma \vdash^{G_e} e : t) \quad C \vdash^e t = t' \\
\hline
C, \Gamma \vdash^{G_e} e : t'
\end{array}
\]
Chapter 3

Translating GRDTs to Type Classes with Existential Types

In this chapter we assume that the reader is familiar with type classes with existential types (TCET) as found in Chapter 2 and [6]. E.g., Läufer [12] gives a formal description for the single-parameter case. In a recent work [18], the authors formalized the general case including multi-parameter type classes which we will make use of in the following.

We can derive the TCET system straightforwardly from the GRDT system. Instead of equality constraints $t_1 = t_2$ we find now type class constraints $TC t_1...t_n$. For simplicity, we assume that instance declarations are preprocessed and the relations they describe are translated to logic formula. We commonly denote those logic formulas by $P_p$. Commonly, we refer to $P_p$ as the program theory. E.g., the instance declarations from Chapter 1 can be described by the following first-order
formulas.

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

$$\forall a_1, a_2, b_1, b_2. (Ct\ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2) \leftrightarrow Ct\ b_1\ a_1 \land Ct\ a_2\ b_2)$$

$$\forall a_1, a_2, b_1, b_2. (Ct\ a_1\ a_2 \leftrightarrow \exists a_2. (Ct\ a_1\ a_2 \land Ct\ a_2\ a_3))$$

where $\leftrightarrow$ denotes Boolean equivalence. We refer the interested reader to [19] for more details on the translation of instances to logic formula.

For each class declaration `class TC a1...an where m::t` we assume a new primitive $m : \forall a.TC\ a \Rightarrow t$. For simplicity, we restrict ourselves to monomorphic class methods. That is, we require that $fv(t) \subseteq a$ which is sufficient for the purpose of the paper.

The typing rules for TCET are almost the same as those for GRDTs in Figure 2.1. 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 $\Gamma$ in the TCET system. In case of $True, \Gamma \vdash^T e : t$ we sometimes write $\Gamma \vdash^T e : t$ for short. We also adjust rule (K) and introduce a new rule (M) to take care of class methods.

$$K : \forall \vec{a}, \vec{b}. D \Rightarrow t \rightarrow T\ \vec{a}$$

$$C, \Gamma \vdash^T e : [\vec{t}/\vec{a}]t\ \ P_p \models C \supset [\vec{t}/\vec{a}, \vec{t}/\vec{b}]D$$

$$M : \forall \vec{a}. TC\ \vec{a} \Rightarrow t\ \ fv(t) \subseteq \vec{a}$$

$$P_p \models C \supset TC\ \vec{t}$$

$$C, \Gamma \vdash^T m : [\vec{t}/\vec{a}]t$$

Note that entailment is now defined w.r.t. the program theory. We write $P_p \models C \supset [\vec{t}/\vec{a}, \vec{t}/\vec{b}]D$ to denote that any model satisfying $P_p$ and $C$ also satisfies $[\vec{t}/\vec{a}, \vec{t}/\vec{b}]D$.

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

\[ P_p \models (\text{Ct } (T \ a_1...a_n) \ (T \ b_1...b_n) \land \text{Ct } (T \ b_1...b_n) \ (T \ a_1...a_n)) \supset \]

\[ (\text{Ct } a_1 \ b_1 \land \text{Ct } b_1 \ a_1 \land ... \text{Ct } a_n \ b_n \land \text{Ct } b_n \ a_n) \]

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 [13].

The above condition (1) can always be met by introducing an instance for each constructor $T$. The second condition is an assumption in our approach. Note that Baars and Swierstra [1] and Hinze and Cheney [3] employ a different encoding which satisfies the above condition (2) by construction.

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 $e$. The transformation is defined as $\forall e_1. e[e_1] \rightsquigarrow e[\text{cast } e_1]$ where $e_1$ is syntactically different from $\text{cast } e_2$ for some expression $e_2$.

We are in the position to establish the following connection between GRDTs and TCETs.

Theorem 1 (GRDT to TCET) Let $e$ be a GRDT expression and $e'$ be its fully casted version. Let $P_p$ a full and faithful program theory representing all GRDTs type constructors mentioned in $e$. Silently, we transform the GRDT constructors mentioned in $e$ to TCET constructors. We have that $\text{True, } \Gamma \vdash^{G_c} e : t$ iff $\text{True, } \Gamma \vdash^T e' : t$.

Note that in order to translate Example 5 the program theory would need to be strengthened by including additional “improvement” rules such as $P_p \models \text{Ct Bool Int } \supset \text{False}$, $P_p \models \text{Ct Int Bool } \supset \text{False}$ etc.
Translating GRDTs to Existential Types

We give a type-directed translation scheme from TCET to ET. First, we describe a constructive method on how to derive the necessary casting functions. 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 the proof term does 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 4.1. 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 functions and pairs. We assume that the proof rules will be extended accordingly for user-defined types. Rule ($\circ$) allows for the structural composition of proof terms. We assume that $f$ has been appropriately defined in terms of $f_i$ such that the conditions stated in Definition 1 are satisfied. Rules ($\forall E$) and ($\exists E$) deal with universal and existential quantifiers.

**Example 6** We give the derivation tree for $f : Ct a (Int, Bool) \leftrightarrow g_1 : Ct a (b, c), g_2 : Ct b Int, g_3 : Ct c Bool$. For convenience, we combine rule ($\forall E$) with rules (Id),
(Id) \( \forall a. \lambda x. x : Ct a a \leftrightarrow True \)

(Var) \( \forall a, b, f : Ct a b \leftrightarrow f : Ct a b \)

(Trans) \[ f = \lambda x. f_2 (f_1 x) \]
\[ \forall a_1, a_3. f : Ct a_1 a_3 \leftrightarrow \exists a_2. f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3 \]

(Arrow) \[ f = \lambda g. \lambda x. f_2 (g (f_1 x)) \]
\[ \forall a_1, a_2, b_1, b_2. f : Ct (a_1 \rightarrow a_2) (b_1 \rightarrow b_2) \leftrightarrow f_1 : Ct b_1 a_1, f_2 : Ct a_2 b_2 \]

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

(\( \circ \)) \( f : Ct a b \leftrightarrow f_1 : c_1, ..., f_n : c_n \]
\( f_i : c_i \leftrightarrow F_i \)
\( F = F_i \) for \( i = 1, ..., n \)

\( f : Ct a b \leftrightarrow F \)

(\( \forall E \)) \( \forall a. f : Ct t_1 t_2 \leftrightarrow F \)
\( \phi = [t/a] \)

(\( \exists E \)) \( f : c \leftrightarrow \exists a. F \)

Figure 4.1: Proof Term Construction Rules

(Var), (Arrow) and (T).

(Trans) \[ f = \lambda x. g_4 (g_1 x) \]
\( f : Ct a (\text{Int, Bool}) \leftrightarrow g_1 : Ct a (b, c), g_4 : Ct (b, c) (\text{Int, Bool}) \)

(\( \circ \)) \( g_1 : Ct a (b, c) \leftrightarrow \)
\( g_1 : Ct a (b, c) \)
\( g_4(x, y) = (g_2 x, g_3 y) \)

(Pair) \[ g_4 : Ct (b, c) (\text{Int, Bool}) \leftrightarrow \]
\( g_2 : Ct b \text{ Int}, g_3 : Ct c \text{ Bool} \)

\( f : Ct a (\text{Int, Bool}) \leftrightarrow g_1 : Ct a (b, c), g_2 : Ct b \text{ Int}, g_3 : Ct c \text{ Bool} \)
Note that rule (Pair) is an instance of (T). We conclude that

\[
f \ x = \text{let } g_4 \ (x, y) = (g_2 \ x, g_3 \ y) \\
in \ g_4 \ (g_1 \ x)
\]

We can state that proof terms are well-typed.

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

**Lemma 1 (Well-Typed)** Let \( C = \{f_1 : Ct \ a_1 \ b_1, \ldots, f_n : Ct \ a_n \ b_n\} \) and \( \Gamma \) such that \( C \sim \Gamma \) and \( f : Ct \ a \ b \leftrightarrow C \) is valid. Then \( \Gamma \vdash f : a \rightarrow b \).

Note that the proof term \( f \) is equivalent to the identity assuming \( f_1, \ldots, f_n \) are equivalent to the identity as well. This preserves the condition stated in Definition 1.

In our next transformation step, we turn a TCET constructor \( K : \sigma \) into an ET constructor \( K' : \sigma' \). We write \( (K : \sigma) \sim (K' : \sigma') \) to denote this step. We have that \( (K : \forall \bar{a}, \bar{b}.D \Rightarrow t \rightarrow T \bar{a}) \sim (K' : \forall \bar{a}, \bar{b}.t \rightarrow E t_1 t'_1 \rightarrow \ldots \rightarrow E t_n t'_n \rightarrow T \bar{a}) \) where \( D = \{Ct \ t_1 \ t'_1, Ct \ t_1' \ t_1, \ldots, Ct \ t_n \ t'_n, Ct \ t'_n \ t_n\} \). Silently, we assume a fixed order among \( Ct \) constraints. Note that the type constructor \( E \) is defined in Example 3.

We define \( P_p \models C \supset (g, h) : [\bar{t}/\bar{a}]D \) iff \( g_i : Ct \ t_i \ t'_i \leftrightarrow C \) and \( h_i : Ct \ t'_i \ t_i \leftrightarrow C \) for \( i = 1, \ldots, n \) where \( [\bar{t}/\bar{a}]D = \{Ct \ t_1 \ t'_1, Ct \ t'_1 \ t_1, \ldots, Ct \ t_n \ t'_n, Ct \ t'_n \ t_n\} \). Note that \( P_p \models C \supset (g, h) : [\bar{t}/\bar{a}]D \) implies that \( P_p \models C \supset [\bar{t}/\bar{a}]D \) but the other direction does not hold necessarily. That is, proof terms are not “decomposable” in general. This has already been observed by Chen, Zhu and Xi [2].
Example 7  Consider

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

We have that \( P_p \vdash g : Ct (Foo a) (Foo b) \supset h : Ct a b \) but \( h : Ct a b \leftrightarrow g : Ct (Foo a) (Foo b) \) does not exist. 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 \( \vdash =^e \). Clearly, we can build \( g \) on type \( Foo a \rightarrow Foo b \) given \( h \) on type \( a \rightarrow b \) whereas for the other direction we would need to decompose proof terms which is not possible here.

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 \( a \). Hence, we identify some sufficient conditions which allow us to extend the rules in Figure 4.1 faithfully.

Definition 4 (Decomposable Types)  Let \( T \) be an \( n \)-ary type constructor. We say that \( T \) is decomposable at position \( i \) where \( i \in \{1, ..., n\} \) iff \( f_i : Ct a_i b_i \leftrightarrow g : Ct (T a_1...a_n) (T b_1...b_n), h : Ct (T b_1...b_n) (T a_1...a_n) \) exists such that (1) \( f_i \) is well-typed under \( \{g : T a_1...a_n \rightarrow T b_1...b_n, h : T b_1...b_n \rightarrow T a_1...a_n\} \) 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.

Example 8  We show that function types are decomposable in their co-variant position. We make use of \( \bot : \forall a.a \).

\[
\begin{align*}
(\text{Arrow}_\bot) & \quad g = \lambda x.(f \ (\lambda y.x)) \ \bot \\
& \quad g : Ct a_2 b_2 \leftrightarrow f : Ct (a_1 \rightarrow a_2) (b_1 \rightarrow b_2)
\end{align*}
\]

Note that \( g \) is the identity under a lazy semantics. However, it seems that \( h : Ct b_1 a_1 \leftrightarrow f : Ct (a_1 \rightarrow a_2) (b_1 \rightarrow b_2) \) does not exist.
Lemma 2 (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 $f$.

We introduce judgments of the form $C, \Gamma \vdash Te : t \Rightarrow e'$ to translate a TCET expression $e$ into a ET expression $e'$. The translation rules can be found in Figure 4.2 and 4.3. Our main tasks are to resolve cast functions (see rule (Reduce)) and to explicitly insert proof terms in constructors (see rule (P-K)). Note that rule (P-K) implicitly suggests that $D = \{ Ct t_1 t_1', Ct t_1' t_1, ..., Ct t_n t_n', Ct t_n' t_n \}$.

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 $True, \Gamma \vdash^T e : t \Rightarrow e'$. Then $\Gamma \vdash^E e' : t$ where $e$ and $e'$ are equivalent after removal of casts and proof terms.

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

Our proof term construction rules in Figure 4.1 are problematic. E.g., rule (Trans) is potentially non-terminating. In the following Section 4.1, we devise
(Abs) \[ C, \Gamma \vdash^T x : t_1 \vdash^T e : t_2 \leadsto e' \quad \Rightarrow \quad C, \Gamma \vdash^T \lambda x.e : t_1 \rightarrow t_2 \leadsto \lambda x.e' \]

(App) \[ C, \Gamma \vdash^T e_1 : t_2 \rightarrow t \leadsto e'_1 \quad \Rightarrow \quad C, \Gamma \vdash^T e_1 e_2 : t \leadsto e'_2 e'_1 \]

(Var-x) \[ (x : \forall \bar{a}.t) \in \Gamma \quad \Rightarrow \quad C, \Gamma \vdash^T x : [\overline{t[a]}]t \leadsto x \]

(Reduce) \[ D \subseteq C \quad f : Ct \vdash^T t_1 t_2 \leftrightarrow D \quad \Rightarrow \quad C, \Gamma \vdash^T \text{cast} : t_1 \rightarrow t_2 \leadsto f \]

(Case) \[ C, \Gamma \vdash^T \text{case}\ e : [p_i \rightarrow e_i]_{i \in I} : t_2 \leftrightarrow \text{case}\ e' \quad \text{of}\ [p'_i \rightarrow e'_i]_{i \in I} \quad \text{for}\ i \in I \]

(Pat) \[ p : t_1 \vdash^T \forall \bar{b}.(D \cup \Gamma p \vdash^T p') \quad \bar{b} \cap \text{fv}(C, \Gamma, t_2) = \emptyset \quad \Rightarrow \quad C, \Gamma \vdash^T p : t_1 \rightarrow t_2 \leadsto p' \rightarrow e' \]

(K) \[ (K : \forall \bar{a}, \bar{b}.D \Rightarrow t \rightarrow T \bar{a}) \leadsto (K' : \forall \bar{a}, \bar{b}.t \rightarrow E t_1 t'_1 \rightarrow \ldots \rightarrow E t_n t'_n \rightarrow T \bar{a}) \quad \Rightarrow \quad C, \Gamma \vdash^T e : [\overline{t[\bar{a}]}]t \leadsto e' \quad P_p \vdash C \supset (g, h) : [\overline{t[\bar{a}]}]D \quad \Rightarrow \quad C, \Gamma \vdash^T Ke : T \bar{t} \leadsto K' e' \quad E(g, h) \]

Figure 4.2: Type-Directed Translation (Part I)

...
4.1 Decidable Proof Construction Method

In order to distinguish between “Ct” uses and assumptions we write $i : Ct M a b$ to refer 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). We write $f : Ct a b$ to refer to the proof term $f$ associated to a $Ct a b$ assumption. Our task is to construct $Ct M$ uses out of a given set of $Ct$ assumptions. Note that the $Ct$ constraints 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 our method on graph algorithms because $Ct M$ uses must obey some side conditions. E.g., consider $i : Ct M a_1 b_1, j : Ct M a_2 b_2, b_1 = a_2 \rightarrow a$. Hence, we employ Constraint Handling Rules (CHR) [5]. CHRs are rule-based language for specifying transformations among constraints. In Figure 4.4, we provide CHRs to construct $Ct M$s out of $Ct$s. Each CHR simplification rule $(R) \bar{c} \iff \bar{d}$ states that if we find a constraint matching the lhs of a rule we replace this constraint by...
4.1 Decidable Proof Construction Method

the rhs. We assume that $c_i$s refer to type class constraints and $d_i$s refer to either type class constraints or equations. We write $C \rightarrow_R C' - \vec{c}, \phi(\vec{d})$ where $\vec{c} \in C$ such that $\phi(\vec{c}) = \vec{c}'$. Logically, rule (R) reads as $\forall \vec{a}. \vec{c} \leftrightarrow \exists \vec{b}. \vec{d}$ where $\vec{a} = f\nu(\vec{c})$ and $\vec{b} = f\nu(\vec{d}) - \vec{a}$. Each CHR also introduces a transformation rule among expressions written $e \rightsquigarrow e'$. We write $C \rightarrow^* C'$ to denote an $n$ number of application of CHRs starting with the initial store $C$ yielding store $D'$. We write $e \rightsquigarrow^* e'$ to denote a reduction among expressions according to the rules in Figure 4.4.

<table>
<thead>
<tr>
<th>(Id)</th>
<th>$i : CtM a b \iff a = b$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{castm}_i \rightsquigarrow \lambda x.x$</td>
</tr>
<tr>
<td>(Trans1)</td>
<td>$g : Ct a b, i : CtM a' b' \iff g : Ct a b, a = a', j : CtM b b'$</td>
</tr>
<tr>
<td></td>
<td>$\text{castm}_i \rightsquigarrow \text{castm}_j \circ g$</td>
</tr>
<tr>
<td>(Arrow)</td>
<td>$i : CtM (a_1 \rightarrow a_2) (b_1 \rightarrow b_2) \iff i_1 : CtM b_1 a_1, i_2 : CtM a_2 b_2$</td>
</tr>
<tr>
<td></td>
<td>$\text{castm}<em>i \rightsquigarrow \lambda g. \lambda x. \text{castm}</em>{i_2} (g (\text{castm}_{i_1} x))$</td>
</tr>
<tr>
<td>(Pair)</td>
<td>$i : CtM (a_1, a_2) (b_1, b_2) \iff i_1 : CtM a_1 b_1, i_2 : CtM a_2 b_2$</td>
</tr>
<tr>
<td></td>
<td>$\text{castm}<em>i \rightsquigarrow \lambda (x, y). ((\text{castm}</em>{i_1} x), (\text{castm}_{i_2} y))$</td>
</tr>
<tr>
<td>(Trans↓)</td>
<td>$g : Ct a b, h : Ct b c \implies h \circ g : Ct a c$</td>
</tr>
</tbody>
</table>

Figure 4.4: CHR-based Proof Term Construction

Note that rule (Trans) from Figure 4.1 has been split into rules (Trans1) and (Id). A naive CHR-translation of transitivity such as

$i : CtM a' b' \iff j : CtM a' b, k : CtM b b'$

$\text{castm}_i \rightsquigarrow \text{castm}_k \circ \text{castm}_j$

leads to problems because we need to guess $b$. Our idea is to incrementally build $CtM$ uses out of $Ct$ assumptions. Note that there is no rule (Var). The same effect can be achieved by rule (Trans1) in combination with rule (Id).
Example 9  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.

\[
g_1 : \text{Ct} (a, (b, c)), g_2 : \text{Ct} b \text{ Int}, g_3 : \text{Ct} c \text{ Bool}, i : \text{CtM} a (\text{Int}, \text{Bool})
\]

\[\Rightarrow_{\text{Trans}}\]

\[
g_1 : \text{Ct} (a, (b, c)), g_2 : \text{Ct} b \text{ Int}, g_3 : \text{Ct} c \text{ Bool}, j : \text{CtM} (b, c) (\text{Int}, \text{Bool})
\]

\[\Rightarrow_{\text{Pair}}\]

\[
g_1 : \text{Ct} a (b, c), g_2 : \text{Ct} b \text{ Int}, g_3 : \text{Ct} c \text{ Bool}, k : \text{CtM} b \text{ Int},
\]

\[
l : \text{CtM} c \text{ Bool}
\]

\[\Rightarrow_{\text{Trans}}\]

\[
g_1 : \text{Ct} a (b, c), g_2 : \text{Ct} b \text{ Int}, g_3 : \text{Ct} c \text{ Bool}, m : \text{CtM} \text{ Int} \text{ Int},
\]

\[
l : \text{CtM} c \text{ Bool}
\]

\[\Rightarrow_{\text{Trans}}\]

\[
g_1 : \text{Ct} a (b, c), g_2 : \text{Ct} b \text{ Int}, g_3 : \text{Ct} c \text{ Bool}, m : \text{CtM} \text{ Int} \text{ Int},
\]

\[
n : \text{CtM} \text{ Bool} \text{ Bool}
\]

\[\Rightarrow_{\text{Id}}\]

\[
g_1 : \text{Ct} a (b, c), g_2 : \text{Ct} b \text{ Int}, g_3 : \text{Ct} c \text{ Bool}
\]

Note that cast_m is equivalent to f which is defined in Example 6. Rule (Pair) is a special instance of rule (T). The side conditions for rule (T) are the same as those stated in Figure 4.1.

In rule (Trans↓) we make use of a CHR propagation rule where we add the rhs if we find a constraint in the store which matches the lhs. Note that each “decomposition” rule such as (Arrow↓) in Example 8 implies a propagation rule

\[(\text{Arrow↓}) \quad f : \text{Ct} (a_1 \rightarrow a_2) (b_1 \rightarrow b_2) \quad \Rightarrow (\lambda x. (f (\lambda y. x)) \bot) : \text{Ct} a_2 b_2\]

It should be clear now that simplification rules incrementally resolve CtM uses whereas propagation rules build the closure of all available Ct assumptions. Silently, we avoid to apply propagation rules twice on the same constraints (to avoid infinite propagation).
Example 10 The following derivation shows that building the transitive closure of \( Ct \) is vital. However, we can only apply \( \text{Arrow} \downarrow \) after we have applied \( \text{Trans} \downarrow \).

\[
g : \text{Ct} (b \to c) \ a, h : \text{Ct} a (b \to d), i : \text{CtM} c d
\]

\[
\rightarrow_{\text{Trans} \downarrow} g : \text{Ct} (b \to c) \ a, h : \text{Ct} a (b \to d), (h \circ g) : \text{Ct} (b \to c) (b \to d), i : \text{CtM} c d
\]

\[
\rightarrow_{\text{Arrow} \downarrow} g : \text{Ct} (b \to c) \ a, h : \text{Ct} a (b \to d), (h \circ g) : \text{Ct} (b \to c) (b \to d),
\]

\[
(\lambda x.((h \circ g) (\lambda y.x)) \bot) : \text{Ct c d}, i : \text{CtM} c d
\]

We note that CHRs are “indeterministic”. E.g., Example 9 gives rise to the following alternative derivation.

\[
g_1 : \text{Ct} a (b, c), g_2 : \text{Ct} b \text{ Int}, g_3 : \text{Ct} c \text{ Bool}, i : \text{CtM} a (\text{Int, Bool})
\]

\[
\rightarrow^* g_1 : \text{Ct} a (b, c), g_2 : \text{Ct} b \text{ Int}, g_3 : \text{Ct} c \text{ Bool}, b = \text{Int}, c = \text{Bool}
\]

Note that the final stores differ. Indeed, CHRs are non-confluent. E.g., rules \((\text{Id})\) and \((\text{Trans1})\) 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 \( C = \{f_1 : \text{Ct} \ a_1 b_1, ..., f_n : \text{Ct} \ a_n b_n\} \) and \( i : \text{CtM} a b, C \rightarrow^* D' \). We say that the CHR derivation is \textit{good} iff \( C \) and \( D' \) are \textit{logically} equivalent, i.e., \( \models C \leftrightarrow \exists f_0(D') - f_0(C).D' \). That is, we rule out derivations yielding stores with unresolved \( \text{CtM} \) uses and \text{False}. We can state that our CHR-based method in Figure 4.4 is sound w.r.t. the system described in Figure 4.1. 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 3 (Sound CHR Construction) Let \( C = \{f_1 : \text{Ct} \ a_1 b_1, ..., f_n : \text{Ct} \ a_n b_n\} \) and \( i : \text{CtM} a b, C \rightarrow^* D' \) and \( \text{castm}_i \rightarrow^* e \) such that the CHR derivation is \textit{good}.

Then, \( f : \text{Ct} a b \rightarrow C \) such that \( f \) and \( e \) are equivalent.
Lemma 4 (Complete CHR Construction) Let \( C = \{f_1 : \text{Ct}\ a_1\ b_1, \ldots, f_n : \text{Ct}\ a_n\ b_n\} \) such that \( f : \text{Ct}\ a\ b \leftrightarrow C \). Then, \( i : \text{CTM}\ a\ b, C \rightarrow^* C \) such that \( \text{castm}_i \rightarrow^* e \) and \( f \) and \( e \) are equivalent.

Lemma 5 (Sound Term Construction) 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 \( \text{castm}_i \rightarrow^* e_1 \) and \( i : \text{CTM}\ a\ b, C \rightarrow^* D_2 \) and \( \text{castm}_i \rightarrow^* e_2 \) such that both CHR derivations are good. Then, \( e_1 \) and \( e_2 \) are equivalent.

Note that in order to find a good derivation we might need to back track. Recall that our rules are non-confluent. Even worse, CHR are non-terminating. E.g., consider

\[
\begin{align*}
g : \text{Ct}\ a\ b, h : \text{Ct}\ b\ a, i : \text{CTM}\ a\ b \\
&\rightarrow_{\text{Trans}1} g : \text{Ct}\ a\ b, h : \text{Ct}\ b\ a, j : \text{CTM}\ b\ b \\
&\rightarrow_{\text{Trans}1} g : \text{Ct}\ a\ b, h : \text{Ct}\ b\ a, k : \text{CTM}\ a\ b \\
&\ldots
\end{align*}
\]

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" \( \text{Ct} \) assumptions of the form \( g : \text{Ct}\ a\ (a, b) \). Such assumptions must result from invalid GRDT definitions which we generally rule out. Due to space limitations, we refer to the technical report version [20] for details. 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. In the following, we show how to integrate our method with a general solving method for constructing typing derivations. Thus, we obtain a decidable method for translating GRDTs to ETs.
4.2 Combing Proof Term Construction and Building Typing Derivations

In [18], 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. Here, we combine the solving approach introduced in [18] with our CHR-based proof term construction method. We introduce a judgement of the form \( \Gamma, e \vdash (e' \mid F_e \mid t_e) \) to denote GRDT expression \( e \) under type environment \( \Gamma \) produce an ET expression \( e' \) of type \( t_e \) and a formula \( F_e \) which describes all possible typing derivations of \( e' \). We call expression \( e' \) as a pre-term which has translated data types and patterns according to Figure 4.2 and 4.3 and is fully casted.

Before we state the soundness of the pre-term and formula generation in Figure 4.5 and 4.6, we define formula solving.

**Definition 5** Assume there is no nested case expression. Let \( \Gamma, e \vdash (e' \mid F_e \mid t_e) \) where \( F_e \) is a formula of shape \( C_o \land (D_1 \supset C_1) \land \ldots \land (D_n \supset C_n) \). Let \( C \) be a constraint. We run

\[
C \rightarrow^* C'
C_o \rightarrow^* C'_o
C, D_i \rightarrow^* D'_i
C, D_i, C_i \rightarrow^* C'_i
\]

for \( i = 1, \ldots, n \). Then \( C \) solves \( F_e \) iff \( |(\exists \bar{a} . C') \leftrightarrow (\exists \bar{a} . C'_o) \) where \( \bar{a} = fv(C, \Gamma) \). False \( \notin C' \) and \( |(\exists \bar{a}_i . D'_i) \leftrightarrow (\exists \bar{a}_i . C'_i) \) where \( \bar{a}_i = fv(C, D_i, \Gamma) \) for \( i = 1, \ldots, n \).

We say \( F_e \) is solvable iff \( C \) solves \( F_e \) for some \( C \).

**Lemma 6** Let \( \Gamma, e \vdash (e' \mid F_e \mid t_e) \). Given \( C \) solves \( F_e \), then we have \( e' \rightsquigarrow e'' \) by rules in Figure 4.4 where there is no castm left in \( e'' \).
4.2 Combing Proof Term Construction and Building Typing Derivations

\[(\text{Abs})\]
\[\Gamma, x : a, e \vdash (e' \downarrow F \uparrow t)\]
\[\Gamma, \lambda x.e \vdash ((\text{cast}_i (\lambda x.e')) \downarrow F \land i : \text{CtM} (a \rightarrow t) \downarrow b)\]

\[(\text{App})\]
\[\Gamma, e_1 \vdash (e'_1 \downarrow F_1 \downarrow t_1)\]
\[\Gamma, e_2 \vdash (e'_2 \downarrow F_2 \downarrow t_2)\]
\[\Gamma, (e_1 e_2) \vdash ((\text{cast}_i (e'_1 e'_2)) \downarrow F_1 \land F_2 \land t_1 = t_2 \vdash a \land i : \text{CtM} (a \downarrow b \downarrow b)\]

\[(\text{Var-x})\]
\[\Gamma, x \vdash ((\text{cast}_i x) \downarrow i : \text{CtM} \lfloor t/a \rfloor \downarrow b \downarrow b)\]

\[(\text{Case})\]
\[F' = \bigwedge_{i \in I} (F_i \land t_i = t_e \rightarrow a) \land F_e \land j : \text{CtM} (a \downarrow b)\]
\[\Gamma, \text{case e of } [p_i \rightarrow e_i]_{i \in I} \vdash ((\text{cast}_j (\text{case e' of } [p'_i \rightarrow e'_i]_{i \in I})) \downarrow F' \downarrow b)\]

\[(\text{Pat})\]
\[p \vdash \overline{b} (D \downarrow F_p \downarrow F' \downarrow t)\]
\[\Gamma, p \rightarrow e \vdash (p' \rightarrow e' \downarrow a \downarrow t \rightarrow t_e \land (\overline{b} = \overline{SK} (a, \text{fv}(\Gamma))) \land (D \supset F_e) \downarrow a)\]

\[(K)\]
\[K : \forall \overline{a}. C t t_1 t_2 \land C t t_2 t_1 \Rightarrow t \rightarrow T \overline{a}\]
\[\Gamma, e \vdash (e' \downarrow F \uparrow t')\]
\[K' : \forall \overline{a}, \overline{b}. t \rightarrow E t_1 t_2 \rightarrow T \overline{a}\]

\[F' = F \land i : \text{CtM} (T \overline{t}) \downarrow t_3 \land k : \text{CtM} t_1 t_2 \land l : \text{CtM} t_2 t_1 \land j : \text{CtM} (\lfloor t/\overline{a} \rfloor t \downarrow (E t_1 t_2) \downarrow (T \overline{t}) \downarrow b \downarrow b = t' \downarrow (E t_1 t_2) \downarrow t_3)\]
\[\Gamma, (K e) \vdash (\text{cast}_i ((\text{cast}_j K') e' (E (\text{cast}_k, \text{cast}_l))) \downarrow F' \downarrow t_3)\]

Figure 4.5: Pre-term and Formula generation (Part I)

Theorem 4 (Pre-term and Formula Generation Soundness) Let \(\Gamma, e \vdash (e' \downarrow F_e \downarrow t_e)\).

Let \(\phi\) be the mgu of \(C\) and \(\phi(C) \sim \Gamma'\) (See Definition 3). Given \(C\) solves \(F_e\) and \(e' \sim e''\) (See Figure 4.4). Then \(\phi(\Gamma) \cup \Gamma' \vdash E e'' : \phi(t_e)\).
Consider the following TCET program

```haskell
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 4.2 and 4.3 and replace all occurrences of `cast` in the program text by `castm` where each `castm` occurrences are attached to distinct locations.

```haskell
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))) = castm_1 ((castm_2 tail) (castm_3 x))
```
We generate the following “implication” constraint out of the above program text.

\[ t = Erk\ a \rightarrow a, a = Sk_1, b = Sk_2 \ a, \]
\[(g : Ct\ a [b], h : Ct\ [b] a) \supset (1 : CtM\ a_1 b_1, b_1 \equiv a,\]
\[2 : CtM\ a_2 b_2, a_2 = [a'_2] \rightarrow [a'_2],\]
\[3 : CtM\ a_3 b_3, b_3 = a, b_2 = b_3 \rightarrow a_1))\]

(4.1)

Annotation \( f_H : Erk_H a \rightarrow a \) 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 \( \text{castm}_i \) expression gives rise to \( i : CtM\ a \ b \) where \( \text{castm}_i :: a \rightarrow b \). To each \( Ct \) assumption we attach proof terms (see rule (P-K)). We make use of the TCET representation of GRDTs 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 (4.1) represents all possible typing derivations. We simply solve this constraint by applying CHRs defined in Figure 4.4 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 \( C_0, D \rightarrow^* D' \) and \( C_0, D, C \rightarrow^* C' \) and check that \( D' \) and \( C' \) are logically equivalent (modulo variables in the initial store). We refer the interested reader to [18] for more details.

For the above constraint (4.1) 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,
4.2 Combing Proof Term Construction and Building Typing Derivations

\[ t = \text{Erk} \ a \rightarrow a, a = Sk_1, b = Sk_2 \ a, g : Ct \ a \ [b], h : Ct \ [b] \ a, \]
\[ 1 : CtM \ a_1 \ b_1, b_1 = a, 2 : CtM \ a_2 \ b_2, a_2 = [a'_2] \rightarrow [a'_2], \]
\[ 3 : CtM \ a_3 \ b_3, a_3 = a, b_2 = b_3 \rightarrow a_1 \]
\[ f_H' \ (L_H' \times (E \ (g,h))) = \text{castm}_1 \ ((\text{castm}_2 \ \text{tail}) \ (\text{castm}_3 \ x)) \]

\[ \text{Trans} \]
\[ t = \text{Erk} \ a \rightarrow a, a = Sk_1, b = Sk_2 \ a, g : Ct \ a \ [b], h : Ct \ [b] \ a, \]
\[ b_1 = a, a_2 = [a'_2] \rightarrow [a'_2], a_3 = a, b_2 = b_3 \rightarrow a_1, \]
\[ 1 : CtM \ a_1 \ a, 2 : CtM \ ([a'_2] \rightarrow [a'_2]) \ (b_3 \rightarrow a_1), 3 : CtM \ a \ b_3 \]
\[ f_H (L_H \times (E \ (g,h))) = \text{let castm}_3 = \text{castm}_1 \circ g \]
\[ \text{in } \text{castm}_1 ((\text{castm}_2 \ \text{tail}) \ (\text{castm}_3 \ x)) \]

\[ \text{Trans} \]
\[ t = \text{Erk} \ a \rightarrow a, a = Sk_1, b = Sk_2 \ a, g : Ct \ a \ [b], h : Ct \ [b] \ a, \]
\[ b_1 = a, a_2 = [a'_2] \rightarrow [a'_2], a_3 = a, b_2 = b_3 \rightarrow a_1, \]
\[ 5 : CtM \ a_1 \ a, 2 : CtM \ ([a'_2] \rightarrow [a'_2]) \ (b_3 \rightarrow [b]), 4 : CtM \ [b] \ b_3 \]
\[ f_H' \ (L_H' \times (E \ (g,h))) = \text{let castm}_3 = \text{castm}_1 \circ g \]
\[ \text{castm}_1 = \text{castm}_5 \circ h \]
\[ \text{in } \text{castm}_1 ((\text{castm}_2 \ \text{tail}) \ (\text{castm}_3 \ x)) \]

\[ t_{\text{id}} \]
\[ t = \text{Erk} \ a \rightarrow a, a = Sk_1, b = Sk_2 \ a, g : Ct \ a \ [b], h : Ct \ [b] \ a, \]
\[ b_1 = a, a_2 = [a'_2] \rightarrow [a'_2], a_3 = a, b_2 = b_3 \rightarrow a_1, a_1 = [b], \]
\[ ([a'_2] \rightarrow [a'_2]) = (b_3 \rightarrow [b]), [b] = b_3 \]
\[ f_H' \ (L_H' \times (E \ (g,h))) = \text{let castm}_3 = \text{castm}_1 \circ g \]
\[ \text{castm}_1 = \text{castm}_5 \circ h \]
\[ \text{castm}_2 \ x = x \]
\[ \text{castm}_4 \ x = x \]
\[ \text{castm}_5 \ x = x \]
\[ \text{in } \text{castm}_1 ((\text{castm}_2 \ \text{tail}) \ (\text{castm}_3 \ x)) \]
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 3. We note that several other derivations are possible. E.g., consider the following where we apply rule (Id) instead of (Trans1).

$$t = \text{Erk } a \rightarrow a, a = Sk_1, b = Sk_2 \ a, g : Ct \ a \ [b], h : Ct \ [b] \ a,$$

$$1 : CtM \ a_1 \ b_1, b_1 = a, 2 : CtM \ a_2 \ b_2, a_2 = [a'_2] \rightarrow [a'_2],$$

$$3 : CtM \ a_3 \ b_3, a_3 = a, b_2 = b_3 \rightarrow a_1$$

$$\leftrightarrow t = \text{Erk } a \rightarrow a, a = Sk_1, b = Sk_2 \ a, g : Ct \ a \ [b], h : Ct \ [b] \ a,$$

$$b_1 = a, a_2 = [a'_2] \rightarrow [a'_2], a_3 = a, b_2 = b_3 \rightarrow a_1,$$

$$1 : CtM \ a_1 \ a, 2 : CtM \ ([a'_2] \rightarrow [a'_2]) \ (b_3 \rightarrow a_1), 3 : CtM \ a \ b_3$$

$$\rightarrow_{Id} t = \text{Erk } a \rightarrow a, a = Sk_1, b = Sk_2 \ a, g : Ct \ a \ [b], h : Ct \ [b] \ a,$$

$$b_1 = a, a_2 = [a'_2] \rightarrow [a'_2], a_3 = a, b_2 = b_3 \rightarrow a_1, a = b_3,$$

$$1 : CtM \ a_1 \ a, 2 : CtM \ ([a'_2] \rightarrow [a'_2]) \ (a \rightarrow a_1)$$

$$\rightarrow_{Id} t = \text{Erk } a \rightarrow a, a = Sk_1, b = Sk_2 \ a, g : Ct \ a \ [b], h : Ct \ [b] \ a,$$

$$b_1 = a, a_2 = [a'_2] \rightarrow [a'_2], a_3 = a, b_2 = b_3 \rightarrow a_1, a = b_3, ([a'_2] \rightarrow [a'_2]) = (a \rightarrow a_1)$$

$$1 : CtM \ a_1 \ a$$

$$\leftrightarrow \text{False}$$

Note that skolem variable $Sk_1$ is unified with $[a'_2]$ which immediately yields failure. That is, we obtain a “bad” final store. However, there might be other derivations which yield “good” final stores. Each of them corresponds to a valid solution. The following is another possible translation of Example 3.
4.2 Combing Proof Term Construction and Building Typing Derivations

\[
 f_{H'} (L_{H'} x (E (g,h))) = \text{let } \text{castm}_2 \ g \ x = \text{castm}_5 \ (g \ (\text{castm}_4 \ x)) \\
\quad \text{castm}_4 = g \\
\quad \text{castm}_5 = h \\
\quad \text{castm}_1 \ x = x \\
\quad \text{castm}_3 \ x = x \\
\quad \text{in } \text{castm}_1 \ ((\text{castm}_2 \ \text{tail}) \ (\text{castm}_3 \ x))
\]
Chapter 5

Heuristics

Our goal is to minimize the amount of cast functions. This serves two purposes. First, we can reduce the operational overhead of performing the identify operation on large parts of the program. Second, we can perform the translation from GRDTs to ETs more efficiently. The fewer cast functions are required, the fewer proof term construction steps are required.

The basic idea of our method to minimize the amount of cast function is to perform type inference on the original GRDT program without any of the additional GRDT type assumptions. Note that this is nothing else than simple Hindley/Milner inference. Each genuine use of a GRDT assumption will raise a type error. Therefore, we identify those locations which contribute to the type error. Clearly, fixing those locations, i.e. inserting some appropriate cast functions, will fix the type error. As a heuristics we identify all locations which contribute to a minimal unsatisfiable constraint. We only need to fully cast all these locations. We illustrate our method with a couple of examples.
Example 11 Let’s consider the second clause of the append function in Example 15. There are 8 distinct program locations in total which can be casted.

\[
\text{app} :: \text{Sum } n \ m \ p \rightarrow \text{Seq } a \ n \rightarrow \text{Seq } a \ m \rightarrow \text{Seq } a \ p
\]

\[
\text{app} (\text{Step } p) \ (\text{Cons } x \ xs) \ ys = \\
(\text{Cons}_1 \ x_2 \ (\text{app}_3 \ p_4 \ xs_5 \ ys_6)_7)_8
\]

The constraint generated by Hindley-Milner type inference from this clause is \( t_i \) represents the type of expression labelled by \( i \) :

\[
t = \text{Sum } n \ m \ p \rightarrow \text{Seq } a \ n \rightarrow \text{Seq } a \ m \rightarrow \text{Seq } a \ p, n = Sk_1, m = Sk_2, p = Sk_3, a = Sk_4,
\]

\[
t_1 = a_1 \rightarrow (\text{Seq } a_1 \ m_1) \rightarrow (E \ n_1 \ (S \ m_1)), t_2 = x, t_3 = t, t_4 = (\text{Sum } n_4 \ m \ p_4),
\]

\[
t_5 = (\text{Seq } a \ m_5), t_6 = (\text{Seq } a \ m), t_1 = t_2 \rightarrow t_7 \rightarrow t_8, t_3 = t_4 \rightarrow t_5 \rightarrow t_6 \rightarrow t_7
\]

(5.1)

In the above constraint store, there is exactly one set of locations \( \{3, 4, 5\} \) which corresponding to the only minimum unsatisfiable constraint \( t_3 = \text{Sum } n \ m \ p \rightarrow \text{Seq } a \ n \rightarrow \text{Seq } a \ m \rightarrow \text{Seq } a \ p, t_4 = (\text{Sum } n_4 \ m \ p_4), t_5 = (\text{Seq } a \ m_5), t_3 = t_4 \rightarrow t_5 \rightarrow t_6 \rightarrow t_7 \). Thus we know these are the only locations involve genuine uses of GRDT assumptions. It is sufficient to fix the types of locations \( \{3, 4, 5\} \) by inserting appropriate casts to make the program typable.

The definition of min unsat constraint naturally suggests that fixing any one location inside the set corresponding to a minimum unsatisfiable constraint to an appropriate type will make the constraint satisfiable. For instance, in Example 11 , casting at location 5 gives the ET program in Example 15.

However, the choice of location is not arbitrary without the decomposition assumption. Consider the following example:
Example 12  Let’s consider another GRDT program.

\[
\text{data Erk } a = (a \rightarrow \text{Int} = \text{Int} \rightarrow \text{Int}) \Rightarrow I a
\]

\[
g :: \text{Int} \rightarrow \text{Int}
g = \text{undefined}
\]

\[
f :: \text{Erk } a \rightarrow \text{Int}
f x = (g_1 \ x_2)_3
\]

In this small program, there is one min unsat constraint corresponding to the location set \{1, 2\}. Note that there is no way for us to cast the expression \(x\) because function types are not decomposable in their contra-variant position. Thus we cannot construct a cast from \(a\) to \(\text{Int}\) from the context \(a \rightarrow \text{Int} = \text{Int} \rightarrow \text{Int}\). However, we are able to find a cast function for \(g\) because no decomposition is involved in the proof term construction.

The difficulty now is that computing all min unsat constraints is exponential. The heuristic might not be effective in practice if the time of finding all min unsat constraints offsets the saving from the reducing of casts. However, we can effectively compute one min unsat constraint or the intersection of all of them in quadratic time. Thus, in case there is only one min unsat constraint, or the intersection of all of the min unsat constraints is non-empty we can compute the intersection and pick one location to be casted.

There are also cases where the constraint store is unsatisfiable but the intersection of all the min unsat constraints is empty.
Example 13 Let’s consider another GRDT program.

data T a = T a

data Erk a = (a=Int) => I a

f :: T Int -> Int
f = undefined

g :: Erk a -> (a,Int)
g (I x) = ((f 1 (T 2 x 3) 4) 5, (f 6 (T 7 x 8) 9) 10) 11

The constraints generated by Hindley-Milner type inference from this clause are:

\[ t = (Erk a) \to (a \text{ Int}), a = Sk_1, t_1 = (T \text{ Int}) \to \text{Int}, t_2 = a_2 \to (T a_2), t_3 = a, \]
\[ t_6 = (T \text{ Int}) \to \text{Int}, t_7 = a_2 \to (T a_2), t_8 = a, t_1 = t_4 \to t_5, t_2 = t_3 \to t_5, \]
\[ t_6 = t_9 \to t_{10}, t_7 = t_8 \to t_9, t_{11} = (a, \text{Int}), t_{11} = (t_5, t_{10}) \]

Note that there are 4 min unsat constraints corresponding to the location sets
\{1, 2, 3, 4\}, \{1, 11\}, \{6, 7, 8, 9\} and \{6, 11\}. The intersection of the sets is empty.

In this case, we use an incremental approach:

1. Find one min unsat constraint from the current constraint store

2. Pick one location from the set and insert cast

3. Remove constraints on the picked locations from the constraint store

4. Try to find the intersection of the rest min unsat constraint.

5. (a) Repeat step 1 if the intersection is empty and the constraint store is not satisfiable.
(b) Otherwise similar to the case where the intersection is non-empty.

Let’s consider the program in Example 13 again. Suppose during the incremental process, we firstly found the set \{1, 2, 3, 4\} and pick location 3 to be casted. Then constraint \(t_3 = a\) is removed from the constraint store. Next, we found another set \{1, 11\} and pick location 11. Again, constraints \(t_{11} = (a, \text{Int}), t_{11} = (t_5, t_{10})\) are removed. Then we continue with \{6, 7, 8, 9\} and pick location 8. Constraint \(t_8 = a\) is removed. By far, the constraint store is satisfiable. Thus we can conclude that 3 casts are needed at location 3, 11 and 8. Note that we didn’t find the minimum number of cast which is 2 in this case. However, if we luckily picked location 1 when \{1, 2, 3, 4\} is found and then picked location 6 from \{6, 7, 8, 9\}, we would find the optimal solution.

After finding all the casting locations in the original GRDT program \(e\). We will mark the expressions in \(e\) which do not require a cast. This process is modelled by a function as \(\text{heuristic}(e) = e_h\) where \(e_h\) is the marked program. Then we generate formula which represents type derivations and pre-term out of \(e_h\). Note that the marked expressions receive a different treatment from the rules in Figure 4.5 and 4.5. No casts are inserted in this case and no constraint \(CtM\) is created. The rules are listed in Figure 5.1 and 5.2. There are multiple instance of Rule (K) to take care all the possible combinations of marked and unmarked expressions. Note that Theorem 4 can be straightforwardly extended to marked programs.

**Theorem 5 (Soundness of Heuristic)** Let \(\Gamma, e \vdash (e' \mid F \mid t)\), \(\text{heuristic}(e) = e_h\) and \(\Gamma, e_h \vdash (e'' \mid F' \mid t')\). Assume and all types appearing in assumption constraints in intermediate derivations are decomposable. Then \(S\) solves \(F'\) if \(S\) solves \(F\).
(Var-x) \hspace{1cm} (x : ∀\bar{a}.t) ∈ \Gamma \quad \hspace{1cm} (Abs) \quad \hspace{1cm} (x : a, e \vdash (e' \mid F \mid t)) \hspace{1cm} (\bar{a}) \quad \hspace{1cm} (x : a, e \vdash (\lambda x.e') \mid F \mid t) \hspace{1cm} (\bar{a}) \\
\Gamma, x \vdash (x \mid True \mid \overline{t/a} t) \hspace{1cm} \Gamma, \lambda x.e \vdash ((\lambda x.e') \mid F \mid t) \hspace{1cm} (\bar{a}) \\
\begin{align*}
    (App) & \hspace{1cm} \Gamma, e_1 \vdash (e'_1 \mid F_1 \mid t_1) \quad \Gamma, e_2 \vdash (e'_2 \mid F_2 \mid t_2) \\
    & \hspace{1cm} \Gamma, (e_1, e_2) \vdash ((e'_1, e'_2) \mid F_1 \land F_2 \land t_1 = t_2 \to a \mid t) \\
    (Case) & \hspace{1cm} \Gamma, e \vdash (e' \mid F_1 \mid t_e) \quad \Gamma, p_i \to e_i \vdash (p'_i \to e'_i \mid F_i \mid t_i) \\
    & \hspace{1cm} F' = \bigwedge_{i \in I}(F_i \land t_i = t_e \to a) \land F_e \\
    & \hspace{1cm} \Gamma, \text{case } e \mid [p_i \to e_i]_{i \in I} \vdash (\text{case } e' \mid [p'_i \to e'_i]_{i \in I} \mid F' \mid a) \\
    (Pat) & \hspace{1cm} \phi = [SK/\bar{b}] \quad \phi(C) = C \quad \phi(\Gamma) = \Gamma \\
    & \hspace{1cm} p \vdash p \forall \bar{b}.(D \mid p \mid p' \mid t) \quad \Gamma \cup \Gamma_p, e \vdash (e' \mid F \mid t) \\
    & \hspace{1cm} \Gamma, p \to e \vdash (p' \to e' \mid a = t \to t_e \land (SK \mid a, \text{fv}(\Gamma))) = \bar{b} \land (D \supset F_e \mid a) \\
    (K-1) & \hspace{1cm} K : \forall \bar{a}, \bar{b}.Ct \ t_1 \ t_2 \land Ct \ t_2 \ t_1 \Rightarrow t \to T \ \bar{a} \quad \Gamma, e \vdash (e' \mid F \mid t') \\
    & \hspace{1cm} K' : \forall \bar{a}, \bar{b}.t \to E \ t_1 \ t_2 \to T \ \bar{a} \\
    & \hspace{1cm} F' = F \land [\overline{t/a}] t = t' \land i : CtM \mid (T \ \overline{t}) \ t_3 \land k : CtM \ t_1 \ t_2 \land l : CtM \ t_2 \ t_1 \\
    & \hspace{1cm} \Gamma, (K e) \vdash (\text{cast}_i (K' \ e' \ (E \mid \text{cast}_k, \text{cast}_l)) \mid F' \mid t_3) \\
\end{align*}

Figure 5.1: CHR-based Proof Term Construction (Part I)
\begin{align*}
K : \forall \bar{a}, \bar{b}. C t \; t_1 \; t_2 \land C t \; t_2 \; t_1 \Rightarrow t \rightarrow T \; \bar{a} \quad \Gamma, e \vdash (e' \mid F \mid t') \\
K' : \forall \bar{a}, \bar{b}. t \rightarrow E \; t_1 \; t_2 \rightarrow T \; \bar{a}
\end{align*}

(K-2)

\[F' = F \land k : C t \; M \; t_1 \; t_2 \land l : C t \; M \; t_2 \; t_1 \land\]
\[j : C t \; M \; ([\bar{t}/\bar{a}]t \rightarrow (E \; t_1 \; t_2) \rightarrow (T \; \bar{t})) \; b \land b = t' \rightarrow (E \; t_1 \; t_2) \rightarrow t_3\]

\[\Gamma, (K \; e) \vdash ((\text{cast}_j \; K') \; e' \; (E \; (\text{cast}_k, \text{cast}_l)) \; F' \mid T \; \bar{t})\]

\begin{align*}
K' : \forall \bar{a}, \bar{b}. t \rightarrow E \; t_1 \; t_2 \rightarrow T \; \bar{a} \quad \Gamma, e \vdash (e' \mid F \mid t')
\end{align*}

(K-3)

\[F' = F \land [\bar{t}/\bar{a}]t = t' \land k : C t \; M \; t_1 \; t_2 \land l : C t \; M \; t_2 \; t_1\]
\[\Gamma, (K' \; e') \vdash (E \; (\text{cast}_k, \text{cast}_l)) \; F' \mid T \; \bar{t})\]

(P-Var)

\[x \vdash_p (\text{True} \mid \{x : t\} \mid x \mid t)\]

(P-Pair)

\[
(p_1 \mid-p \; \forall \overline{b}_1. (D_1 \mid \Gamma_{p_1} \mid p'_1 \mid t_1)) \quad (p_2 \mid-p \; \forall \overline{b}_2. (D_2 \mid \Gamma_{p_2} \mid p'_2 \mid t_2))
\]
\[
(p_1, p_2) \mid-p \; \forall \overline{b}_1, \overline{b}_2. (D_1 \land D_2 \mid \Gamma_{p_1} \cup \Gamma_{p_2} \mid (p'_1, p'_2) \mid (t_1, t_2))
\]

\[K : \forall \bar{a}, \bar{b}. C t \; t_1 \; t'_1 \land C t \; t'_1 \; t_1 \Rightarrow t \rightarrow T \; \bar{a}\]

\[K' : \forall \bar{a}, \bar{b}. t \rightarrow E \; t_1 \; t'_1 \rightarrow T \; \bar{a}\]

(P-K)

\[\overline{b} \cap \bar{a} = \emptyset \quad p \mid-p \; \forall \overline{b}'. (D' \mid \Gamma_p \mid p' \mid [\bar{t}/\bar{a}]t) \quad g_1, h_1, ..., g_n, h_n \text{ fresh}\]
\[D'' = D' \land g_1 : C t \; t_1 \; t'_1 \land h_1 : C t \; t'_1 \; t_1\]
\[\Gamma'_p = \Gamma_p \cup \{g_1 : t_1 \rightarrow t'_1, h_1 : t'_1 \rightarrow t_1\}\]

\[K \; p \mid-p \; \forall \overline{b'}, \overline{b}. (D'' \mid \Gamma'_p \mid K' \; p' \mid (E \; (g_1, h_1)) \mid T \; \bar{t})\]

Figure 5.2: CHR-based Proof Term Construction (Part II)
Further Examples

6.1 Transitivity Example

Example 14 Consider

data Erk a b = forall c. (a=(b,c), b=Int) => I (Erk a c)
  | (b=Bool) => B a b

g1 :: (Int,Bool)->Int
  g1 (x, True) = x
  g1 (_, False) = 0

f :: Erk a b -> b
  f (I (B (x,y))))= g1 x

For typing of the above program it is crucial to apply transitivity. E.g., we have that $a = (b, c), b = Int, c = Bool$ which implies that $a = (Int, Bool)$.

Function $f$ is generates the following pre-term:
data Erk_H a b = forall c. I_H (Erk_H a c) (E a (b,c)) (E b Int)  
| B_H (a,b) (E b Bool)

f :: Erk_H a b -> b
f (I (B (x,y) (E (g_c, h_c))) (E (g_a, h_a)) (E (g_b, h_b)))
  = castm_1 ((castm_2 g1) (castm_3 x))

The program text gives rise to the following constrains.

\[ t = Erk a b \rightarrow b, a = Sk_1, b = Sk_2, c = Sk_3 a, \]
\[ ((g_a : Ct a (b,c), h_a : Ct (b,c) a, g_b : Ct b Int, h_b : Ct Int b, \]
\[ g_c : Ct c Bool, h_c : Ct Bool c) \supset \]
\[ (1 : CtM a_1 b_1, b = b_1, 2 : CtM a_2 b_2, a_2 = (Int, Bool) \rightarrow Int, \]
\[ 3 : CtM a_3 b_3, a_3 = a, b_2 = b_3 \rightarrow a_1)) \]

Constraint solving proceeds as follows.

\[ t = Erk a b \rightarrow b, a = Sk_1, b = Sk_2, c = Sk_3 a b, Ct a (b,c), g_a : Ct a (b,c), \]
\[ h_a : Ct (b,c) a, g_b : Ct b Int, h_b : Ct Int b, g_c : Ct c Bool, h_c : Ct Bool c \]

is final. In the other case, we have that

\[ t = Erk a b \rightarrow b, a = Sk_1, b = Sk_2, c = Sk_3 a b, g_a : Ct a (b,c), \]
\[ h_a : Ct (b,c) a, g_b : Ct b Int, h_b : Ct Int b, g_c : Ct c Bool, \]
\[ h_c : Ct Bool c, b = b_1, a_2 = (Int, Bool) \rightarrow Int, a_3 = a, b_2 = b_3 \rightarrow a_1, \]
\[ 1 : CtM a_1 b_1, 2 : CtM a_2 b_2, 3 : CtM a_3 b_3 \]

\[ \leftrightarrow t = Erk a b \rightarrow b, a = Sk_1, b = Sk_2, c = Sk_3 a b, b = b_1, a_2 = (Int, Bool) \rightarrow Int, \]
\[ a_3 = a, b_2 = b_3 \rightarrow a_1, g_a : Ct a (b,c), h_a : Ct (b,c) a, g_b : Ct b Int, \]
\[ h_b : Ct Int b, g_c : Ct c Bool, h_c : Ct Bool c, 1 : CtM a_1 b, \]
\[ 2 : CtM ((Int, Bool) \rightarrow Int) (b_3 \rightarrow a_1, 3 : CtM a b_3 \]
\[ \sim f \ldots = castm_1 ((castm_2 g1) (castm_3 x)) \]
6.1 Transitivity Example

\[ \rightarrow_{Id} \quad t = \text{Erk } a \rightarrow b, a = \text{Sk}_1, b = \text{Sk}_2, c = \text{Sk}_3 \quad a, b = b_1, a_2 = (\text{Int}, \text{Bool}) \rightarrow \text{Int}, \]
\[ a_3 = a, b_2 = b_3 \rightarrow a_1, ((\text{Int}, \text{Bool}) \rightarrow \text{Int}) = (b_3 \rightarrow a_1), g_a : \text{Ct } a \rightarrow (b, c), \]
\[ h_a : \text{Ct } (b, c) \rightarrow \text{Ct } b \text{ Int}, h_b : \text{Ct } \text{Int } b, g_c : \text{Ct } c \text{ Bool}, h_c : \text{Ct } \text{Bool } c, \]
\[ \]
\[ 1 : \text{CtM } \text{Int } b, 3 : \text{CtM } a \ (\text{Int}, \text{Bool}) \]
\[ \]
\[ \rightarrow \quad f \ldots = \text{Let } \text{castm}_2 x = x \]
\[ \text{in } \text{castm}_1 ((\text{castm}_2 \ g_1) \ (\text{castm}_3 \ x)) \]

\[ \rightarrow_{\text{Trans}} \quad t = \text{Erk } a \rightarrow b, a = \text{Sk}_1, b = \text{Sk}_2, c = \text{Sk}_3 \quad a, b = b_1, a_2 = (\text{Int}, \text{Bool}) \rightarrow \text{Int}, \]
\[ a_3 = a, b_2 = b_3 \rightarrow a_1, ((\text{Int}, \text{Bool}) \rightarrow \text{Int}) = (b_3 \rightarrow a_1), g_a : \text{Ct } a \rightarrow (b, c), \]
\[ h_a : \text{Ct } (b, c) \rightarrow \text{Ct } b \text{ Int}, h_b : \text{Ct } \text{Int } b, g_c : \text{Ct } c \text{ Bool}, h_c : \text{Ct } \text{Bool } c, \]
\[ 4 : \text{CtM } b, 3 : \text{CtM } a \ (\text{Int}, \text{Bool}) \]
\[ \]
\[ \rightarrow \quad f \ldots = \text{Let } \text{castm}_2 x = x \]
\[ \text{castm}_1 = \text{castm}_4 \circ h_b \]
\[ \text{in } \text{castm}_1 ((\text{castm}_2 \ g_1) \ (\text{castm}_3 \ x)) \]

\[ \rightarrow_{\text{Trans}} \quad t = \text{Erk } a \rightarrow b, a = \text{Sk}_1, b = \text{Sk}_2, c = \text{Sk}_3 \quad a, b = b_1, a_2 = (\text{Int}, \text{Bool}) \rightarrow \text{Int}, \]
\[ a_3 = a, b_2 = b_3 \rightarrow a_1, ((\text{Int}, \text{Bool}) \rightarrow \text{Int}) = (b_3 \rightarrow a_1), g_a : \text{Ct } a \rightarrow (b, c), \]
\[ h_a : \text{Ct } (b, c) \rightarrow \text{Ct } b \text{ Int}, h_b : \text{Ct } \text{Int } b, g_c : \text{Ct } c \text{ Bool}, h_c : \text{Ct } \text{Bool } c, \]
\[ 4 : \text{CtM } b, 5 : \text{CtM } (b, c) \ (\text{Int}, \text{Bool}) \]
\[ \]
\[ \rightarrow \quad f \ldots = \text{Let } \text{castm}_2 x = x \]
\[ \text{castm}_1 = \text{castm}_4 \circ h_b \]
\[ \text{castm}_3 = \text{castm}_5 \circ g_a \]
\[ \text{in } \text{castm}_1 ((\text{castm}_2 \ g_1) \ (\text{castm}_3 \ x)) \]

\[ \rightarrow_{p_{\text{air}}} \quad t = \text{Erk } a \rightarrow b, a = \text{Sk}_1, b = \text{Sk}_2, c = \text{Sk}_3 \quad a, b = b_1, a_2 = (\text{Int}, \text{Bool}) \rightarrow \text{Int}, \]
\[ a_3 = a, b_2 = b_3 \rightarrow a_1, ((\text{Int}, \text{Bool}) \rightarrow \text{Int}) = (b_3 \rightarrow a_1), g_a : \text{Ct } a \rightarrow (b, c), \]
\[ h_a : \text{Ct } (b, c) \rightarrow \text{Ct } b \text{ Int}, h_b : \text{Ct } \text{Int } b, g_c : \text{Ct } c \text{ Bool}, h_c : \text{Ct } \text{Bool } c, \]
\[ 4 : \text{CtM } b, 6 : \text{CtM } b \text{ Int}, 7 : \text{CtM } c \text{ Bool} \]
\[ \]
\[ \rightarrow \quad f \ldots = \text{Let } \text{castm}_2 x = x \]
\[ \ldots \]
\[ \text{castm}_3 = \text{castm}_5 \circ g_a \]
\[ \text{castm}_5 (m, n) = ((\text{castm}_6 \ m), (\text{castm}_7 \ n)) \]
\[ \text{in } \text{castm}_1 ((\text{castm}_2 \ g_1) \ (\text{castm}_3 \ x)) \]
6.1 Transitivity Example

\[ t = \text{Erk} \ a \ b \to b, a = Sk_1, b = Sk_2, c = Sk_3 \]
\[ a, b = b_1, a_2 = (\text{Int}, \text{Bool}) \to \text{Int}, \]
\[ a_3 = a, b_2 = b_3 \to a_1, ((\text{Int}, \text{Bool}) \to \text{Int}) = (b_3 \to a_1), g_a : \text{Ct} \ a \ (b, c), \]
\[ h_a : \text{Ct} \ (b, c) \ a, g_b : \text{Ct} \ b \ \text{Int}, h_b : \text{Ct} \ (\text{Int} \ b, g_c : \text{Ct} \ c \ \text{Bool}, h_c : \text{Ct} \ \text{Bool} \ c, \]
\[ 4 : \text{CtM} \ b b, 8 : \text{CtM} \ \text{Int} \ \text{Int}, 9 : \text{CtM} \ \text{Bool} \ \text{Bool} \]

\[ f \ldots = \text{Let castm}_2 \ x = x \]
\[ \ldots \]
\[ \text{castm}_5 \ (m, n) = ((\text{castm}_6 \ m), (\text{castm}_7 \ n)) \]
\[ \text{castm}_6 = \text{castm}_8 \circ g_b \]
\[ \text{castm}_7 = \text{castm}_9 \circ g_c \]
\[ \text{in} \ \text{castm}_1 ((\text{castm}_2 \ g_1) (\text{castm}_3 \ x)) \]

\[ t = \text{Erk} \ a \ b \to b, a = Sk_1, b = Sk_2, c = Sk_3 \]
\[ a, b = b_1, a_2 = (\text{Int}, \text{Bool}) \to \text{Int}, \]
\[ a_3 = a, b_2 = b_3 \to a_1, ((\text{Int}, \text{Bool}) \to \text{Int}) = (b_3 \to a_1), g_a : \text{Ct} \ a \ (b, c), \]
\[ h_a : \text{Ct} \ (b, c) \ a, g_b : \text{Ct} \ b \ \text{Int}, h_b : \text{Ct} \ (\text{Int} \ b, g_c : \text{Ct} \ c \ \text{Bool}, h_c : \text{Ct} \ \text{Bool} \ c \]

\[ f \ldots = \text{Let castm}_2 \ x = x \]
\[ \text{castm}_1 = \text{castm}_4 \circ h_b \]
\[ \text{castm}_3 = \text{castm}_5 \circ g_a \]
\[ \text{castm}_5 \ (m, n) = ((\text{castm}_6 \ m), (\text{castm}_7 \ n)) \]
\[ \text{castm}_6 = \text{castm}_8 \circ g_b \]
\[ \text{castm}_7 = \text{castm}_9 \circ g_c \]
\[ \text{castm}_1 \ x = x \]
\[ \text{castm}_3 \ x = x \]
\[ \text{castm}_5 \ x = x \]
\[ \text{in} \ \text{castm}_1 ((\text{castm}_2 \ g_1) (\text{castm}_3 \ x)) \]

The final result of the translation is

\[ \text{data Erk}_H \ a \ b = \forall c. \ \text{I}_H \ (\text{Erk}_H \ a \ c) \ (\text{E} \ b \ (b, c)) \ (\text{E} \ b \ \text{Int}) \]
\[ \mid \text{B}_H \ (a, b) \ (\text{E} \ b \ \text{Bool}) \]

\[ f :: \text{Erk}_H \ a \ b \to b \]
f (I (B (x, y) (E (g_c, h_c))) (E (g_a, h_a)) (E (g_b, h_b))) =

\[
\text{let } \text{castm}_2 \ x = x \\
\text{castm}_1 = \text{castm}_4 \circ h_b \\
\text{castm}_3 = \text{castm}_5 \circ g_a \\
\text{castm}_5 \ (m, n) = ((\text{castm}_6 \ m), (\text{castm}_7 \ n)) \\
\text{castm}_6 = \text{castm}_8 \circ g_b \\
\text{castm}_7 = \text{castm}_9 \circ g_c \\
\text{castm}_4 \ x = x \\
\text{castm}_8 \ x = x \\
\text{castm}_9 \ x = x \\
\text{in } \text{castm}_1 ((\text{castm}_2 \ g_1) (\text{castm}_3 \ x))
\]

### 6.2 Sheard and Pasalic Append Example

The following example is due to Sheard and Pasalic [15].

**Example 15**  
Consider

\[
\text{data } Z = Z \\
\text{data } S \ n = S \ n \\
\text{data } \text{Sum } w \ x \ y = \\
\quad (w=Z, x=y) \Rightarrow \text{Base} \\
\quad | \ \forall m \ n. \ (w=S \ m, n=S \ n) \Rightarrow \text{Step } (\text{Sum } m \ x \ n)
\]

\[
\text{data } \text{Seq } a \ n = \\
\quad n=Z \Rightarrow \text{Nil} \\
\quad | \ \forall m. \ n=S \ m \Rightarrow \text{Cons } a \ (\text{Seq } a \ m)
\]
app :: Sum n m p -> Seq a n -> Seq a m -> Seq a p
app Base Nil ys = ys
app (Step p) (Cons x xs) ys =
    Cons x (app p xs ys)

The program is translated to a pre-term.

data E a b = E (a->b, b->a)

data Z = Z

data S n = S n

data Sum w x y =
    Base (E w Z) (E x y)
    | forall m n. Step (Sum m x n) (E w (S m)) (E y (S n))

data Seq a n =
    Nil (E n Z)
    | forall m. Cons a (Seq a m) (E n (S m))

app :: Sum n m p -> Seq a n -> Seq a m -> Seq a p
app (Base (E (g1,h1)) (E (g2,h2))) (Nil (E (g3,h3))) ys = castm₁ ys
app (Step p (E (g1,h1)) (E (g2,h2))) (Cons x xs (E (g3,h3))) ys =
    castm₁ (Cons (castm₂ x)
        (castm₃ ((castm₄ app) (castm₅ p) (castm₆ xs) (castm₇ ys)))
        (E (castm₈,castm₀)))
The specific composition rules for the user defined data types are listed below.

\[(\text{Seq})\quad i : \text{CtM} (\text{Seq} \ a \ n) (\text{Seq} \ a \ n') \iff i_1 : \text{CtM} (E \ n \ Z) (E \ n' \ Z),
\quad i_2 : \text{CtM} \ a \ a,
\quad i_3 : \text{CtM} (\text{Seq} \ a \ Sk) (\text{Seq} \ a \ Sk),
\quad i_4 : \text{CtM} (E \ n \ (S \ Sk)) (E \ n' \ (S \ Sk))\]

\[
\text{castm}_i \rightsquigarrow \lambda s. \text{case s of}
\quad (\text{Nil} \ x) \to \text{Nil} (\text{castm}_{i_1} x)
\quad (\text{Cons} \ a \ s \ e) \to \text{Cons} (\text{castm}_{i_2} a)
\quad (\text{castm}_{i_3} s)
\quad (\text{castm}_{i_4} e)
\]

\[(E)\quad i : \text{CtM} (E \ (g,h)) (E \ (g',h')) \iff i' : \text{CtM} (g,h) (g',h')\]

\[
\text{castm}_i \rightsquigarrow \lambda (E \ (g,h)). E (\text{castm}_{i'} (g,h))
\]

We consider the constraints on a clause by clause bases. The first clause generates the following pre-term.

\[
\text{app} :: \text{Sum} \ n \ m \ p \rightarrow \text{Seq} \ a \ n \rightarrow \text{Seq} \ a \ m \rightarrow \text{Seq} \ a \ p
\]

\[
\text{app} \ (\text{Base} \ (E \ (g_1,h_1)) \ (E \ (g_2,h_2))) \ (\text{Nil} \ (E \ (g_3,h_3))) \ ys = \text{castm}_1 \ ys
\]

It gives us the following constraints:

\[
t = \text{Sum} \ n \ m \ p \rightarrow \text{Seq} \ a \ n \rightarrow \text{Seq} \ a \ m \rightarrow \text{Seq} \ a \ p, n = Sk_1, m = Sk_2, p = Sk_3, a = Sk_4,
\quad (g_n : \text{Ct} \ n \ Z, h_n : \text{Ct} \ Z \ n, \ g_p : \text{Ct} \ m \ p, h_p : \text{Ct} \ m \ p, m \supset (1 : \text{CtM} \ a_1 b_1, b_1 = p)
\]

We solve the constraint 6.1 as follows. We find that \(t = \text{Sum} \ n \ m \ p \rightarrow \text{Seq} \ a \ n \rightarrow \text{Seq} \ a \ m \rightarrow \text{Seq} \ a \ p, n = Sk_1, m = Sk_2, p = Sk_3, a = Sk_4, g_n :\)
\( Ct \ n \ Z, h_n : Ct \ Z \ n, g_p : Ct \ m \ p, h_p : Ct \ p \ m \) is immediately final. Consider

\[
\begin{align*}
t &= \text{Sum} \ n \ m \ p \to \text{Seq} \ a \ n \to \text{Seq} \ a \ m \to \text{Seq} \ a \ p, n = Sk_1, m = Sk_2, \\
p &= Sk_3, a = Sk_4, g_n : Ct \ n \ Z, h_n : Ct \ Z \ n, g_p : Ct \ m \ p, h_p : Ct \ p \ m, \\
n : CtM \ a_1 b_1, a_1 = \text{Seq} \ a \ m, b_1 = \text{Seq} \ a \ p \\
\text{app} \ldots &= \text{cast}_{m_1} \ ys
\end{align*}
\]

\[
\begin{align*}
\leftharpoonup \quad t &= \text{Sum} \ n \ m \ p \to \text{Seq} \ a \ n \to \text{Seq} \ a \ m \to \text{Seq} \ a \ p, n = Sk_1, m = Sk_2, \\
p &= Sk_3, a = Sk_4, a_1 = \text{Seq} \ a \ m, b_1 = \text{Seq} \ a \ p, a_2 = E \ m \ Z, b_2 = E \ p \ Z, \\
a_3 = a, b_3 = a, a_4 &= \text{Seq} \ a \ Sk, b_4 = \text{Seq} \ a \ Sk, a_5 = E \ m \ (S \ Sk), b_5 = E \ p \ (S \ Sk), \\
g_n : Ct \ n \ Z, h_n : Ct \ Z \ n, g_p : Ct \ m \ p, h_p : Ct \ p \ m, 2 : CtM \ (E \ m \ Z) \ (E \ p \ Z), \\
3 : CtM \ a_1, a_4 &= \text{Seq} \ a \ Sk \ a_2, 5 : CtM \ (E \ m \ (S \ Sk)) \ (E \ p \ (S \ Sk)) \\
\rightharpoonup \quad \text{app} \ldots &= \text{let} \ \text{cast}_{m_1} \ (\text{Nil} \ e) = \text{Nil} \ (\text{cast}_{m_2} \ e) \\
& \quad \quad \quad \quad \quad \text{cast}_{m_1} \ (\text{Cons} \ x \ y \ e) = \text{Cons} \ (\text{cast}_{m_3} \ x) \ (\text{cast}_{m_4} \ y) \ (\text{cast}_{m_5} \ e) \\
& \quad \quad \quad \quad \quad \text{in} \ \text{cast}_{m_1} \ ys
\end{align*}
\]

\[
\begin{align*}
\leftharpoonup \quad t &= \text{Sum} \ n \ m \ p \to \text{Seq} \ a \ n \to \text{Seq} \ a \ m \to \text{Seq} \ a \ p, n = Sk_1, m = Sk_2, \\
p &= Sk_3, a = Sk_4, a_1 = \text{Seq} \ a \ m, b_1 = \text{Seq} \ a \ p, g_n : Ct \ n \ Z, \\
h_n : Ct \ Z \ n, g_p : Ct \ m \ p, h_p : Ct \ p \ m, 2 : CtM \ (E \ m \ Z) \ (E \ p \ Z), \\
5 : CtM \ (E \ m \ (S \ Sk)) \ (E \ p \ (S \ Sk)) \\
\rightharpoonup \quad \text{app} \ldots &= \text{let} \ \text{cast}_{m_1} \ (\text{Nil} \ e) = \text{Nil} \ (\text{cast}_{m_2} \ e) \\
& \quad \quad \quad \quad \quad \text{cast}_{m_1} \ (\text{Cons} \ x \ y \ e) = \text{Cons} \ (\text{cast}_{m_3} \ x) \ (\text{cast}_{m_4} \ y) \ (\text{cast}_{m_5} \ e) \\
& \quad \quad \quad \quad \quad \text{cast}_{m_3} \ x = x \\
& \quad \quad \quad \quad \quad \text{cast}_{m_4} \ x = x \\
& \quad \quad \quad \quad \quad \text{in} \ \text{cast}_{m_1} \ ys
\end{align*}
\]
6.2 Sheard and Pasalic Append Example

\[ \rightarrow_E t = \text{Sum } n \ m \ p \rightarrow \text{Seq } a \ n \rightarrow \text{Seq } a \ m \rightarrow \text{Seq } a \ p, n = Sk_1, m = Sk_2, p = Sk_3, \]
\[ a = Sk_4, a_1 = \text{Seq } a \ m, b_1 = \text{Seq } a \ p, g_n : \text{Ct } n \ Z, h_n : \text{Ct } Z \ n, g_p : \text{Ct } m \ p, \]
\[ h_p : \text{Ct } m \ p, 6 : \text{CtM } (m \rightarrow Z, Z \rightarrow m) \ (p \rightarrow Z, Z \rightarrow p), \]
\[ 5 : \text{CtM } (E \ m \ (S \ Sk)) \ (E \ p \ (S \ Sk)) \]
\[ \rightarrow \text{app ... = let castm}_1 (\text{Nil } e) = \text{Nil } (\text{castm}_2 e) \]
\[ \ldots \]
\[ \text{castm}_4 x = x \]
\[ \text{castm}_2 (E \ (g, h)) = E \ (\text{castm}_6 (g, h)) \]
\[ \text{in castm}_1 \ ys \]

\[ \rightarrow_{p_{air}} t = \text{Sum } n \ m \ p \rightarrow \text{Seq } a \ n \rightarrow \text{Seq } a \ m \rightarrow \text{Seq } a \ p, n = Sk_1, m = Sk_2, p = Sk_3, \]
\[ p = Sk_3, a = Sk_4, a_1 = \text{Seq } a \ m, b_1 = \text{Seq } a \ p, g_n : \text{Ct } n \ Z, h_n : \text{Ct } Z \ n, \]
\[ g_p : \text{Ct } m \ p, h_p : \text{Ct } m \ p, 7 : \text{CtM } (m \rightarrow Z) \ (p \rightarrow Z), 8 : \text{CtM } (Z \rightarrow m) \ (Z \rightarrow p), \]
\[ 5 : \text{CtM } (E \ m \ (S \ Sk)) \ (E \ p \ (S \ Sk)) \]
\[ \rightarrow \text{app ... = let castm}_1 (\text{Nil } e) = \text{Nil } (\text{castm}_2 e) \]
\[ \ldots \]
\[ \text{castm}_2 (E \ (g, h)) = E \ (\text{castm}_6 (g, h)) \]
\[ \text{castm}_6 (g, h) = ((\text{castm}_7 g), (\text{castm}_8 h)) \]
\[ \text{in castm}_1 \ ys \]

\[ \rightarrow_{\text{Arrows}} t = \text{Sum } n \ m \ p \rightarrow \text{Seq } a \ n \rightarrow \text{Seq } a \ m \rightarrow \text{Seq } a \ p, n = Sk_1, m = Sk_2, p = Sk_3, \]
\[ a = Sk_4, a_1 = \text{Seq } a \ m, b_1 = \text{Seq } a \ p, g_n : \text{Ct } n \ Z, h_n : \text{Ct } Z \ n, g_p : \text{Ct } m \ p, \]
\[ h_p : \text{Ct } m \ p, 9 : \text{CtM } p \ m, 10 : \text{CtM } Z \ Z, 11 : \text{CtM } Z \ Z, 12 : \text{CtM } m \ p, \]
\[ 5 : \text{CtM } (E \ m \ (S \ Sk)) \ (E \ p \ (S \ Sk)) \]
\[ \rightarrow \text{app ... = let castm}_1 (\text{Nil } e) = \text{Nil } (\text{castm}_2 e) \]
\[ \ldots \]
\[ \text{castm}_6 (g, h) = ((\text{castm}_7 g), (\text{castm}_8 h)) \]
\[ \text{castm}_7 g x = \text{castm}_{10} (g \ (\text{castm}_9 x)) \]
\[ \text{castm}_8 h x = \text{castm}_{12} (h \ (\text{castm}_{11} x)) \]
\[ \text{in castm}_1 \ ys \]
6.2 Sheard and Pasalic Append Example

\( \mapsto_E \) $$t = \text{Sum } n \ m \ p \rightarrow \text{Seq } a \ n \rightarrow \text{Seq } a \ m \rightarrow \text{Seq } a \ p, n = Sk_1, m = Sk_2, p = Sk_3,$$
\( a = Sk_4, a_1 = \text{Seq } a \ m, b_1 = \text{Seq } a \ p, g_n : \text{Ct } n \ Z, h_n : \text{Ct } Z \ n, g_p : \text{Ct } m \ p, \)
\( h_p : \text{Ct } m \ p, 9 : \text{CtM } m \ p, 10 : \text{CtM } Z \ Z, 11 : \text{CtM } Z \ Z, 12 : \text{CtM } m \ p, \)
\( 13 : \text{CtM } (m \rightarrow (S \ Sk)), (S \ Sk) \rightarrow m) \ (p \rightarrow (S \ Sk), (S \ Sk) \rightarrow p) $$
\( \mapsto \) $$\text{app ... = let } \text{castm}_1 (\text{Nil } e) = \text{Nil} (\text{castm}_2 e)$$

$$\ldots$$
\( \text{castm}_8 h \ x = \text{castm}_{12} (h (\text{castm}_{11} x)) \)
\( \text{castm}_5 (E (g,h)) = E (\text{castm}_{13} (g,h)) \)

in \( \text{castm}_1 \) \( \text{ys} \)

\( \mapsto_{\text{pair}} \) $$t = \text{Sum } n \ m \ p \rightarrow \text{Seq } a \ n \rightarrow \text{Seq } a \ m \rightarrow \text{Seq } a \ p, n = Sk_1, m = Sk_2, p = Sk_3,$$
\( a = Sk_4, a_1 = \text{Seq } a \ m, b_1 = \text{Seq } a \ p, g_n : \text{Ct } n \ Z, h_n : \text{Ct } Z \ n, g_p : \text{Ct } m \ p, \)
\( h_p : \text{Ct } m \ p, 9 : \text{CtM } m \ p, 10 : \text{CtM } Z \ Z, 11 : \text{CtM } Z \ Z, 12 : \text{CtM } m \ p, \)
\( 14 : \text{CtM } (m \rightarrow (S \ Sk)) \ (p \rightarrow (S \ Sk)), 15 : \text{CtM } ((S \ Sk) \rightarrow m) \ ((S \ Sk) \rightarrow p) $$
\( \mapsto \) $$\text{app ... = let } \text{castm}_1 (\text{Nil } e) = \text{Nil} (\text{castm}_2 e)$$

$$\ldots$$
\( \text{castm}_5 (E (g,h)) = E (\text{castm}_{13} (g,h)) \)
\( \text{castm}_{13} (g,h) = (\text{castm}_{14} g), (\text{castm}_{15} h)) \)

in \( \text{castm}_1 \) \( \text{ys} \)

\( \mapsto_{\text{Arrows}} \) $$t = \text{Sum } n \ m \ p \rightarrow \text{Seq } a \ n \rightarrow \text{Seq } a \ m \rightarrow \text{Seq } a \ p, n = Sk_1, m = Sk_2, p = Sk_3,$$
\( a = Sk_4, a_1 = \text{Seq } a \ m, b_1 = \text{Seq } a \ p, g_n : \text{Ct } n \ Z, h_n : \text{Ct } Z \ n, g_p : \text{Ct } m \ p, \)
\( h_p : \text{Ct } m \ p, 9 : \text{CtM } m \ p, 10 : \text{CtM } Z \ Z, 11 : \text{CtM } Z \ Z, 12 : \text{CtM } m \ p, \)
\( 16 : \text{CtM } p \ m, 17 : \text{CtM } (S \ Sk) (S \ Sk), 18 : \text{CtM } (S \ Sk) (S \ Sk), 19 : \text{CtM } m \ p $$
\( \mapsto \) $$\text{app ... = let } \text{castm}_1 (\text{Nil } e) = \text{Nil} (\text{castm}_2 e)$$

$$\ldots$$
\( \text{castm}_{13} (g,h) = (\text{castm}_{14} g), (\text{castm}_{15} h)) \)
\( \text{castm}_{14} g \ x = \text{castm}_{16} (g (\text{castm}_{17} x)) \)
\( \text{castm}_{15} h \ x = \text{castm}_{18} (h (\text{castm}_{19} x)) \)

in \( \text{castm}_1 \) \( \text{ys} \)
\[ \text{Trans}^{*} \quad t = \text{Seq} a n \rightarrow \text{Seq} a m \rightarrow \text{Seq} a p, n = Sk_1, m = Sk_2, p = Sk_3, \]
\[ a = Sk_4, a_1 = \text{Seq} a m, b_1 = \text{Seq} a p, g_n : \text{Ct} n Z, h_n : \text{Ct} Z n, g_p : \text{Ct} m p, \]
\[ h_p : \text{Ct} p m, 20 : \text{CtM} m m, 10 : \text{CtM} Z Z, 11 : \text{CtM} Z Z, 12 : \text{CtM} m p, \]
\[ 21 : \text{CtM} m m, 17 : \text{CtM} (S Sk) (S Sk), 18 : \text{CtM} (S Sk) (S Sk), 19 : \text{CtM} m p \]
\[ \leadsto \quad \text{app} \ldots = \text{let castm}_1 (\text{Nil } e) = \text{nil} (\text{castm}_2 e) \]
\[ \quad \ldots \]
\[ \quad \text{castm}_{15} h x = \text{castm}_{18} (h (\text{castm}_{19} x)) \]
\[ \quad \text{castm}_9 = \text{castm}_{20} \circ h_p \]
\[ \quad \text{castm}_{16} = \text{castm}_{21} \circ h_p \]
\[ \quad \text{in castm}_1 \text{ ys} \]

\[ \text{Trans}^{*} \quad t = \text{Seq} a n \rightarrow \text{Seq} a m \rightarrow \text{Seq} a p, n = Sk_1, m = Sk_2, p = Sk_3, \]
\[ a = Sk_4, a_1 = \text{Seq} a m, b_1 = \text{Seq} a p, g_n : \text{Ct} n Z, h_n : \text{Ct} Z n, g_p : \text{Ct} m p, \]
\[ h_p : \text{Ct} p m, 20 : \text{CtM} m m, 10 : \text{CtM} Z Z, 11 : \text{CtM} Z Z, 22 : \text{CtM} p p, \]
\[ 21 : \text{CtM} m m, 17 : \text{CtM} (S Sk) (S Sk), 18 : \text{CtM} (S Sk) (S Sk), 23 : \text{CtM} p p \]
\[ \leadsto \quad \text{app} \ldots = \text{let castm}_1 (\text{Nil } e) = \text{nil} (\text{castm}_2 e) \]
\[ \quad \ldots \]
\[ \quad \text{castm}_{16} = \text{castm}_{21} \circ h_p \]
\[ \quad \text{castm}_{12} = \text{castm}_{22} \circ g_p \]
\[ \quad \text{castm}_{19} = \text{castm}_{23} \circ g_p \]
\[ \quad \text{in castm}_1 \text{ ys} \]
In a similar manner, the second clause of the function can be translated. The final result is shown below. Note that it is simplified by removing redundant \texttt{castm} to improve readability.
app :: Sum n m p -> Seq a n -> Seq a m -> Seq a p
app (Base (E (g1,h1)) (E (g2,h2))) (Nil (E (g3,h3))) ys =
  let castm1 (Nil e) = Nil (castm2 e)
    castm1 (Cons x y e) = Cons x y (castm5 e)
    castm2 (E (g,h)) = E (castm6 (g,h))
    castm6 (g,h) = ((castm7 g), (castm8 h))
    castm7 g x = castm10 (g (castm9 x))
    castm8 h x = castm12 (h (castm11 x))
    castm5 (E (g,h)) = E (castm13 (g,h))
    castm13 (g,h) = ((castm14 g), (castm15 h))
    castm14 g x = castm16 (g (castm17 x))
    castm15 h x = castm18 (h (castm19 x))
    castm9 = castm20 ∘ h_p
    castm10 x = x
    castm11 x = x
    castm22 x = x
    castm21 x = x
    castm17 x = x
    castm18 x = x
    castm23 x = x
    in castm1 ys
app (Step p’ (E (g1,h1)) (E (g2,h2))) (Cons x xs (E (g3,h3))) ys =
  let castm1 (Nil x) = Nil (castm2 x)
6.3 Hinze and Cheney Trie Example

The following GRDT example is introduced by Cheney and Hinze in [4]. We slightly simplify the program and leave out some constructors and clauses.

Example 16 Consider

data Trie k v =
    forall k1 k2.(k=Either k1 k2) => Tp (Trie k1 v) (Trie k2 v)

merge :: Trie k v -> Trie k v -> Trie k v
merge (Tp ta tb) (Tp ta' tb') =
    (Tp1 (merge2 ta3 ta'4)5 (merge6 tb7 tb'8)9)10

This example demonstrates the use of transitivity and decomposition in the derivation. From the first parameter of the pattern, we conclude the constraint $k =$
Either \(k_a\) \(k_b\); from the second parameter, we have \(k = Either k'_a k'_b\). By transitivity and decomposition, we conclude from \(Either k_a k_b = k = Either k'_a k'_b\) that \(k_a = k'_a\) and \(k_b = k'_b\), which allows us to type check the recursive calls.

We employ the heuristic from Section 5. Note that the program above is already justified. The constraint generated from the program is:

\[
t = Trie k v \rightarrow Trie k v \rightarrow Trie k v, v = Sk_1, k = Sk_2, k_a = Sk_3, k_b = Sk_4 k,
\]

\[
k'_a = Sk_5, k'_b = Sk_6, k, t_1 = (Trie k_{11} v_1) \rightarrow (Trie k_{12} v_1) \rightarrow (Trie k_1 v_1),
\]

\[
t_1 = t_5 \rightarrow t_9 \rightarrow t_{10}, t_2 = (Trie k_2 v_2) \rightarrow (Trie k_2 v_2) \rightarrow (Trie k_2 v_2),
\]

\[
t_2 = t_3 \rightarrow t_4 \rightarrow t_5, t_3 = Trie k_a v, t_4 = Trie k'_a v,
\]

\[
t_6 = (Trie k_6 v) \rightarrow (Trie k_6 v) \rightarrow (Trie k_6 v), t_6 = t_7 \rightarrow t_8 \rightarrow t_9, t_7 = Trie k'_b v,
\]

\[
t_8 = Trie k'_b v, t_{10} = (Trie k v)
\]

There are two minimal unsat constraints which corresponding to locations \(\{2, 3, 4\}\) and \(\{6, 7, 8\}\). Suppose we chose locations \(4\) and \(8\) to be casted. Then the pre-term generated from the marked program is as following:

\[
data Trie k v =
\]

\[
forall k_1 k_2.Tp (Trie k_1 v) (Trie k_2 v) (E k (Either k_1 k_2))
\]

\[
merge : Trie k v \rightarrow Trie k v \rightarrow Trie k v
\]

\[
merge (Tp ta tb e@(E (g,h))) (Tp ta' tb' e'@(E (g',h'))) =
\]

\[
Tp (merge ta (castm_1 ta')) (merge tb (castm_2 tb')) (E (castm_{11}, castm_{12}))
\]

For simplicity, we only show the derivation of the first recursive call \((merge ta ta')\). \((merge tb tb')\) follows similarly. The following are the decomposition rules for user
defined datatypes.

\[(\text{Trie}) \quad i : C t M (T r ie \ k \ v) (T r ie \ k' \ v) \iff i_1 : C t M (T r ie \ Sk_1 \ v) (S e q \ Sk_1 \ v), \]
\[i_2 : C t M (T r ie \ Sk_2 \ v) (S e q \ Sk_2 \ v), \]
\[i_3 : C t M (E \ k (E i t h e r \ Sk_1 \ Sk_2)) \]
\[(E \ k' (E i t h e r \ Sk_1 \ Sk_2)) \]
\[\text{castm}_i \leadsto \lambda (T r ie \ t_1 \ t_2 \ e). T r ie (\text{castm}_{i_1} \ t_1) \]
\[\quad (\text{castm}_{i_2} \ t_2) \]
\[\quad (\text{castm}_{i_3} \ e) \]

\[(E) \quad i : C t M (E \ (g, h)) (E \ (g', h')) \iff i' : C t M (g, h) (g', h') \]
\[\text{castm}_i \leadsto \lambda (E \ (g, h)). E (\text{castm}_{i'} \ (g, h)) \]

\[(\text{Either}) \quad k : C t (E i t h e r \ a \ b) (E i t h e r \ a' \ b') \iff \lambda x. p r o j e c t_L (\text{castm}_k (i n j e c t_L x)) \]
\[: C t \ a \ a', \]
\[\lambda x. p r o j e c t_R (\text{castm}_k (i n j e c t_R x)) \]
\[: C t \ b \ b' \]

The program text gives rise to the following constrains. Note that ... represents the constraints for the second recursive call. The justifications for locations does not involve cast remain unchanged. The constructors are modified to accept proof terms as arguments. Only those locations which require a cast will have its type
6.3 Hinze and Cheney Trie Example

$t_i$ split into $a_i$ and $b_i$.

\[ t = \text{Trie} \ k \ v \to \text{Trie} \ k \ v \to \text{Trie} \ k \ v, v = Sk_1, k = Sk_2, \]
\[ k_a = Sk_3 \ k, b_b = Sk_4 \ k, k_a' = Sk_5 \ k, b_b' = Sk_6 \ k, \]
\[ ((g : Ct \ k \ (\text{Either} \ k_a \ k_b), h : Ct \ (\text{Either} \ k_a \ k_b) \ k), \]
\[ g' : Ct \ k' \ (\text{Either} \ k_a' \ k_b'), h' : Ct \ (\text{Either} \ k_a' \ k_b') \ k') \supset \]
\[ (t_1 = (\text{Trie} \ k_{11} \ v_{11}) \to (\text{Trie} \ k_{12} \ v_{11}) \to (E \ a_{11} \ b_{12}) \to (\text{Trie} \ k_1 \ v_1), \]
\[ t_1 = t_5 \to t_9 \to (E \ a_{11} \ b_{12}) \to t_{10}, t_2 = \text{Trie} \ k_2 \ v_2 \to \text{Trie} \ k_2 \ v_2 \to \text{Trie} \ k_2 \ v_2, \]
\[ t_2 = t_3 \to b_4 \to t_5, t_3 = \text{Trie} \ k_a \ v, a_4 = \text{Trie} \ k_a' \ v, t_{10} = (\text{Trie} \ k \ v), 4 : CtM \ a_4 \ b_4, \]
\[ \ldots \]
\[ 11 : CtM \ a_{11} \ b_{11}, a_{11} = k, b_{11} = (\text{Either} \ k_a'' \ k_b''), \]
\[ 12 : CtM \ a_{12} \ b_{12}, a_{12} = (\text{Either} \ k_a'' \ k_b''), b_{12} = k \]

Constraint solving proceeds as follows.

\[ t = \text{Trie} \ k \ v \to \text{Trie} \ k \ v \to \text{Trie} \ k \ v, v = Sk_1, k = Sk_2, k_a = Sk_3 \ k, b_b = Sk_4 \ k, \]
\[ k_a' = Sk_5 \ k, k_b' = Sk_6 \ k, g : Ct \ k \ (\text{Either} \ k_a \ k_b), h : Ct \ (\text{Either} \ k_a \ k_b) \ k, \]
\[ g' : Ct \ k' \ (\text{Either} \ k_a' \ k_b'), h' : Ct \ (\text{Either} \ k_a' \ k_b') \ k' \]

is final. In the other case, we have that

\[ t = \text{Trie} \ k \ v \to \text{Trie} \ k \ v \to \text{Trie} \ k \ v, v = Sk_1, k = Sk_2, k_a = Sk_3 \ k, b_b = Sk_4 \ k, \]
\[ k_a' = Sk_5 \ k, k_b' = Sk_6 \ k, g : Ct \ k \ (\text{Either} \ k_a \ k_b), h : Ct \ (\text{Either} \ k_a \ k_b) \ k, \]
\[ g' : Ct \ k \ (\text{Either} \ k_a' \ k_b'), h' : Ct \ (\text{Either} \ k_a' \ k_b') \ k, \]
\[ t_1 = (\text{Trie} \ k_{11} \ v_{11}) \to (\text{Trie} \ k_{12} \ v_{11}) \to (E \ a_{11} \ b_{12}) \to (\text{Trie} \ k_1 \ v_1), \]
\[ t_1 = t_5 \to t_9 \to (E \ a_{11} \ b_{12}) \to t_{10}, t_2 = \text{Trie} \ k_2 \ v_2 \to \text{Trie} \ k_2 \ v_2 \to \text{Trie} \ k_2 \ v_2, \]
\[ t_2 = t_3 \to b_4 \to t_5, t_3 = \text{Trie} \ k_a \ v, a_4 = \text{Trie} \ k_a' \ v, t_{10} = (\text{Trie} \ k \ v), 4 : CtM \ a_4 \ b_4, \]
\[ \ldots \]
\[ 11 : CtM \ a_{11} \ b_{11}, a_{11} = k, b_{11} = (\text{Either} \ k_a'' \ k_b''), \]
\[ 12 : CtM \ a_{12} \ b_{12}, a_{12} = (\text{Either} \ k_a'' \ k_b''), b_{12} = k \]
6.3 Hinze and Cheney Trie Example

\[
\begin{align*}
t & = \text{Trie } k \text{ } v \rightarrow \text{Trie } k \text{ } v \rightarrow \text{Trie } k \text{ } v, v = Sk_1, k = Sk_2, k_a = Sk_3, k_b = Sk_4, k, \\
& \quad \quad k'_a = Sk_5, k'_b = Sk_6, k, g : \text{Ct } k \text{ (Either } k_a \text{ } k_b \text{)}, h : \text{Ct } (\text{Either } k_a \text{ } k_b) \text{, } k, \\
g' & : \text{Ct } k \text{ (Either } k_a' \text{ } k_b' \text{)}, h' : \text{Ct } (\text{Either } k_a' \text{ } k_b') \text{, } k, \\
t_1 & = (\text{Trie } k_{11} \text{ } v_1) \rightarrow (\text{Trie } k_{12} \text{ } v_1) \rightarrow (E \text{ } a_{11} \text{ } b_{12}) \rightarrow (\text{Trie } k_1 \text{ } v_1), \\
t_1 & = t_5 \rightarrow t_9 \rightarrow (E \text{ } a_{11} \text{ } b_{12}) \rightarrow t_{10}, t_2 = \text{Trie } k_2 \text{ } v_2 \rightarrow \text{Trie } k_2 \text{ } v_2 \rightarrow \text{Trie } k_2 \text{ } v_2, \\
t_2 & = t_3 \rightarrow b_4 \rightarrow t_5, t_3 = \text{Trie } k_a \text{ } v, a_4 = \text{Trie } k'_a \text{ } v, t_{10} = (\text{Trie } k \text{ } v), 4 : \text{CtM } a_4 \text{ } b_4, \\
a_{11} & = k, b_{11} = (\text{Either } k_a' \text{ } k_b' \text{)}, a_{12} = (\text{Either } k_a'' \text{ } k_b''), b_{12} = k, \ldots \\
11 : & \text{CtM } a_{11} \text{ } b_{11}, 12 : \text{CtM } a_{12} \text{ } b_{12}
\end{align*}
\]

\[
\begin{align*}
\Rightarrow & \quad \text{merge } \ldots = \\
& \quad \quad \text{Tp } (\text{merge } ta \text{ (castm}_4 \text{ } ta')) (\text{merge } tb \text{ (castm}_8 \text{ } tb')) \\
& \quad \quad \quad \quad \quad \quad \text{(E } \text{ (castm}_1 \text{ } \text{castm}_2)) \\
\rightarrow_{\text{Trans1*}} & \quad t = \text{Trie } k \text{ } v \rightarrow \text{Trie } k \text{ } v \rightarrow \text{Trie } k \text{ } v, v = Sk_1, k = Sk_2, k_a = Sk_3, k_b = Sk_4, k, \\
& \quad \quad k'_a = Sk_5, k'_b = Sk_6, k, g : \text{Ct } k \text{ (Either } k_a \text{ } k_b \text{)}, h : \text{Ct } (\text{Either } k_a \text{ } k_b) \text{, } k, \\
g' & : \text{Ct } k \text{ (Either } k_a' \text{ } k_b' \text{)}, h' : \text{Ct } (\text{Either } k_a' \text{ } k_b') \text{, } k, \\
t_1 & = (\text{Trie } k_{11} \text{ } v_1) \rightarrow (\text{Trie } k_{12} \text{ } v_1) \rightarrow (E \text{ } a_{11} \text{ } b_{12}) \rightarrow (\text{Trie } k_1 \text{ } v_1), \\
t_1 & = t_5 \rightarrow t_9 \rightarrow (E \text{ } a_{11} \text{ } b_{12}) \rightarrow t_{10}, t_2 = \text{Trie } k_2 \text{ } v_2 \rightarrow \text{Trie } k_2 \text{ } v_2 \rightarrow \text{Trie } k_2 \text{ } v_2, \\
t_2 & = t_3 \rightarrow b_4 \rightarrow t_5, t_3 = \text{Trie } k_a \text{ } v, a_4 = \text{Trie } k'_a \text{ } v, t_{10} = (\text{Trie } k \text{ } v), 4 : \text{CtM } a_4 \text{ } b_4, \\
a_{11} & = k, b_{11} = (\text{Either } k_a' \text{ } k_b' \text{)}, a_{12} = (\text{Either } k_a'' \text{ } k_b''), b_{12} = k, \\
& \quad \quad (\text{Either } k_a \text{ } k_b) = (\text{Either } k_a'' \text{ } k_b''), \ldots \\
13 : & \text{CtM } (\text{Either } k_a \text{ } k_b) \text{ (Either } k_a'' \text{ } k_b''), 14 : \text{CtM } k \text{ } k
\end{align*}
\]

\[
\begin{align*}
\Rightarrow & \quad \text{merge } \ldots = \\
& \quad \quad \text{let } \text{castm}_1 = \text{castm}_3 \circ g \\
& \quad \quad \quad \text{castm}_2 = \text{castm}_4 \circ h \\
& \quad \quad \text{in } \text{Tp } (\text{merge } ta \text{ (castm}_4 \text{ } ta')) (\text{merge } tb \text{ (castm}_8 \text{ } tb')) \\
& \quad \quad \quad \quad \quad \quad \text{(E } \text{ (castm}_1 \text{ } \text{castm}_2))
\end{align*}
\]
⇝

\[ t = \text{Trie } k \ v \rightarrow \text{Trie } k \ v \rightarrow \text{Trie } k \ v, v = Sk_1, k = Sk_2, \ k_a = Sk_3, k_b = Sk_4 \ k, \]
\[ k'_a = Sk_5 \ k, k'_b = Sk_6 \ k, \]
\[ g: Ct \ k \ (\text{Either } k_a \ k_b), h: Ct \ (\text{Either } k_a \ k_b) \ k, \]
\[ g': Ct \ k \ (\text{Either } k'_a \ k'_b), h': Ct \ (\text{Either } k'_a \ k'_b) \ k, \]
\[ t_1 = (\text{Trie } k_{11} \ v_1) \rightarrow (\text{Trie } k_{12} \ v_1) \rightarrow (E \ a_{11} \ b_{12}) \rightarrow (\text{Trie } k_1 \ v_1), \]
\[ t_1 = t_5 \rightarrow t_9 \rightarrow (E \ a_{11} \ b_{12}) \rightarrow t_{10}, t_2 = \text{Trie } k_2 \ v_2 \rightarrow \text{Trie } k_2 \ v_2 \rightarrow \text{Trie } k_2 \ v_2, \]
\[ t_2 = t_3 \rightarrow b_4 \rightarrow t_5, t_3 = \text{Trie } k_a \ v, a_4 = \text{Trie } k'_a \ v, t_{10} = (\text{Trie } k \ v), \]
\[ 4 : CtM \ (\text{Trie } k'_a \ v) \ (\text{Trie } k_2 \ v_2), a_{11} = k, b_{11} = (\text{Either } k''_a \ k''_b), \]
\[ a_{12} = (\text{Either } k''_a \ k''_b), b_{12} = k, (\text{Either } k_a \ k_b) = (\text{Either } k''_a \ k''_b) \]
\[ \ldots \]

⇝

\[ \text{merge ... = } \]
\[ \text{let castm}_{11} = \text{castm}_{13} \circ \ g \]
\[ \text{castm}_{12} = \text{castm}_{14} \circ \ h \]
\[ \text{castm}_{13} \ x = x \]
\[ \text{castm}_{14} \ x = x \]
\[ \text{in } Tp \ (\text{merge } ta \ (\text{castm}_{4} \ ta')) \ (\text{merge } tb \ (\text{castm}_{8} \ tb')) \]
\[ (E \ (\text{castm}_{11}, \text{castm}_{12})) \]

⇝

\[ t = \text{Trie } k \ v \rightarrow \text{Trie } k \ v \rightarrow \text{Trie } k \ v, v = Sk_1, k = Sk_2, \ k_a = Sk_3, k_b = Sk_4 \ k, \]
\[ k'_a = Sk_5 \ k, k'_b = Sk_6 \ k, \]
\[ g: Ct \ k \ (\text{Either } k_a \ k_b), h: Ct \ (\text{Either } k_a \ k_b) \ k, \]
\[ g': Ct \ k \ (\text{Either } k'_a \ k'_b), h': Ct \ (\text{Either } k'_a \ k'_b) \ k, \]
\[ t_1 = (\text{Trie } k_{11} \ v_1) \rightarrow (\text{Trie } k_{12} \ v_1) \rightarrow (E \ a_{11} \ b_{12}) \rightarrow (\text{Trie } k_1 \ v_1), \]
\[ t_1 = t_5 \rightarrow t_9 \rightarrow (E \ a_{11} \ b_{12}) \rightarrow t_{10}, t_2 = \text{Trie } k_2 \ v_2 \rightarrow \text{Trie } k_2 \ v_2 \rightarrow \text{Trie } k_2 \ v_2, \]
\[ t_2 = t_3 \rightarrow b_4 \rightarrow t_5, t_3 = \text{Trie } k_a \ v, a_4 = \text{Trie } k'_a \ v, t_{10} = (\text{Trie } k \ v), \]
\[ a_{11} = k, b_{11} = (\text{Either } k''_a \ k''_b), a_{12} = (\text{Either } k''_a \ k''_b), b_{12} = k, \]
\[ (\text{Either } k_a \ k_b) = (\text{Either } k''_a \ k''_b), 15 : CtM \ (\text{Trie } Sk_a \ v) \ (\text{Trie } Sk_a \ v), \]
\[ 16 : CtM \ (\text{Trie } Sk'_a \ v) \ (\text{Trie } Sk'_a \ v), \]
\[ 17 : CtM \ (E \ k'_a \ (\text{Either } Sk_a \ Sk'_a)) \ (E \ k_2 \ (\text{Either } Sk'_a \ Sk_b)) \]
\[ \ldots \]
let castm\textsubscript{11} = castm\textsubscript{13} \circ g

\[
\text{castm}\textsubscript{14} \ast = x
\]

castm\textsubscript{4} (Tp \ a \ b \ e) = Tp (castm\textsubscript{15} \ a) (castm\textsubscript{16} \ b) (castm\textsubscript{17} \ e)

in Tp (merge \ (castm\textsubscript{4} \ ta')) \ (merge \ (castm\textsubscript{8} \ tb'))

(E (castm\textsubscript{11}, castm\textsubscript{12}))

\[\rightarrow_E \ t = \text{Trie } k \ v \rightarrow \text{Trie } k \ v \rightarrow \text{Trie } k \ v, v = Sk_1, k = Sk_2, k_a = Sk_3 \ k, k_b = Sk_4 \ k,
\]

\[k'_a = Sk_5 \ k, k'_b = Sk_6 \ k, g : Ct \ k \ (\text{Either } k_a \ k_b), h : Ct \ (\text{Either } k_a \ k_b) \ k,
\]

\[g' : Ct \ k \ (\text{Either } k'_a \ k'_b), h' : Ct \ (\text{Either } k'_a \ k'_b) \ k,
\]

\[t_1 = (\text{Trie } k_{11} \ v_{11}) \rightarrow (\text{Trie } k_{12} \ v_{12}) \rightarrow (E \ a_{11} \ b_{12}) \rightarrow (\text{Trie } k_1 \ v_1),
\]

\[t_2 = t_3 \rightarrow b_1 \rightarrow t_5, t_3 = \text{Trie } k_\ a \ v, a_4 = \text{Trie } k'_a \ v, t_{10} = (\text{Trie } k \ v),
\]

\[a_{11} = k, b_{11} = (\text{Either } k''_a \ k''_b), a_{12} = (\text{Either } k''_a \ k''_b), b_{12} = k,
\]

\[E (\text{Either } k_a \ k_b) = (\text{Either } k''_a \ k''_b), 15 : CtM (\text{Trie } Sk_a \ v) (\text{Trie } Sk_a \ v),
\]

\[16 : CtM (\text{Trie } Sk'_a \ v) (\text{Trie } Sk'_a \ v),
\]

\[18 : CtM (k'_a \ (\text{Either } Sk_a \ Sk'_a), (\text{Either } Sk_a \ Sk'_a) \rightarrow k_2)
\]

\[(k_2 \ (\text{Either } Sk_a \ Sk'_a), (\text{Either } Sk_a \ Sk'_a) \rightarrow k'_a)
\]

\[\rightarrow_E \ t = \text{Trie } k \ v \rightarrow \text{Trie } k \ v \rightarrow \text{Trie } k \ v, v = Sk_1, k = Sk_2, k_a = Sk_3 \ k, k_b = Sk_4 \ k,
\]

\[k'_a = Sk_5 \ k, k'_b = Sk_6 \ k, g : Ct \ k \ (\text{Either } k_a \ k_b), h : Ct \ (\text{Either } k_a \ k_b) \ k,
\]

\[g' : Ct \ k \ (\text{Either } k'_a \ k'_b), h' : Ct \ (\text{Either } k'_a \ k'_b) \ k,
\]

\[t_1 = (\text{Trie } k_{11} \ v_{11}) \rightarrow (\text{Trie } k_{12} \ v_{12}) \rightarrow (E \ a_{11} \ b_{12}) \rightarrow (\text{Trie } k_1 \ v_1),
\]

\[t_2 = t_3 \rightarrow b_1 \rightarrow t_5, t_3 = \text{Trie } k_\ a \ v, a_4 = \text{Trie } k'_a \ v, t_{10} = (\text{Trie } k \ v),
\]

\[a_{11} = k, b_{11} = (\text{Either } k''_a \ k''_b), a_{12} = (\text{Either } k''_a \ k''_b), b_{12} = k,
\]

\[E (\text{Either } k_a \ k_b) = (\text{Either } k''_a \ k''_b), 15 : CtM (\text{Trie } Sk_a \ v) (\text{Trie } Sk_a \ v),
\]

\[16 : CtM (\text{Trie } Sk'_a \ v) (\text{Trie } Sk'_a \ v),
\]

\[18 : CtM (k'_a \ (\text{Either } Sk_a \ Sk'_a), (\text{Either } Sk_a \ Sk'_a) \rightarrow k_2)
\]

\[(k_2 \ (\text{Either } Sk_a \ Sk'_a), (\text{Either } Sk_a \ Sk'_a) \rightarrow k'_a)
\]

\[\rightarrow \text{merge } \ldots \ =
\]

let castm\textsubscript{11} = castm\textsubscript{13} \circ g

\[
\text{castm}\textsubscript{8} (Tp \ a \ b \ e) = \ldots
\]

castm\textsubscript{17} (E (g,h)) = E (castm\textsubscript{18} (g,h))

in Tp (merge \ (castm\textsubscript{4} \ ta')) \ (merge \ (castm\textsubscript{8} \ tb'))

(E (castm\textsubscript{11}, castm\textsubscript{12}))
6.3 Hinze and Cheney Trie Example

$\rightarrow_{p_{air}}$

\[
t = \text{Trie } k \ v \rightarrow \text{Trie } k \ v \rightarrow \text{Trie } k \ v, v = Sk_1, k = Sk_2, k_a = Sk_3 \ k, k_b = Sk_4 \ k,
\]
\[
k'_a = Sk_5 \ k, k'_b = Sk_6 \ k, g : Ct \ k \ (\text{Either } k_a \ k_b), h : Ct \ (\text{Either } k_a \ k_b) \ k,
\]
\[
g' : Ct \ k \ (\text{Either } k'_a \ k'_b), h' : Ct \ (\text{Either } k'_a \ k'_b) \ k,
\]
\[
t_1 = (\text{Trie } k_{11} \ v_1) \rightarrow (\text{Trie } k_{12} \ v_1) \rightarrow (E \ a_{11} \ b_{12}) \rightarrow (\text{Trie } k_1 \ v_1),
\]
\[
t_1 = t_5 \rightarrow t_9 \rightarrow (E \ a_{11} \ b_{12}) \rightarrow t_{10}, t_2 = \text{Trie } k_2 \ v_2 \rightarrow \text{Trie } k_2 \ v_2 \rightarrow \text{Trie } k_2 \ v_2,
\]
\[
t_2 = t_3 \rightarrow b_4 \rightarrow t_5, t_3 = \text{Trie } k_a \ v, a_4 = \text{Trie } k'_a \ v, t_{10} = (\text{Trie } k) \ v,
\]
\[
a_{11} = k, b_{11} = (\text{Either } k''_a \ k''_b), a_{12} = (\text{Either } k''_a \ k''_b), b_{12} = k,
\]
\[
(E \text{Either } k_a \ k_b) = (\text{Either } k''_a \ k''_b), 15 : CtM \ (\text{Trie } Sk_a) \ v \ (\text{Trie } Sk_a) \ v,
\]
\[
16 : CtM \ (\text{Trie } Sk'_a) \ v \ (\text{Trie } Sk'_a) \ v,
\]
\[
19 : CtM \ (k'_a \rightarrow (\text{Either } Sk_a \ Sk'_a)) \ (k_2 \rightarrow (\text{Either } Sk_a \ Sk'_a)),
\]
\[
20 : CtM \ ((\text{Either } Sk_a \ Sk'_a) \rightarrow k'_a) \ ((\text{Either } Sk_a \ Sk'_a) \rightarrow k_2)
\]

\[\cdots\]

$\rightarrow_{\text{merge...}}$

\[
\text{let castm}_{11} = \text{castm}_{13} \circ g
\]

\[\cdots\]

\[
\text{castm}_{17} \ (E \ (g, h)) = E \ (\text{castm}_{18} \ (g, h))
\]
\[
\text{castm}_{18} \ (g, h) = ((\text{castm}_{19} \ g), (\text{castm}_{20} \ h))
\]
\[
in \ Tp \ (\text{merge } t_a \ (\text{castm}_{14} \ t_a')) \ (\text{merge } t_b \ (\text{castm}_{18} \ t_b'))
\]
\[
(E \ (\text{castm}_{11}, \text{castm}_{12}))
\]

$\rightarrow_{\text{Arrows}}$

\[
t = \text{Trie } k \ v \rightarrow \text{Trie } k \ v \rightarrow \text{Trie } k \ v, v = Sk_1, k = Sk_2, k_a = Sk_3 \ k, k_b = Sk_4 \ k,
\]
\[
k'_a = Sk_5 \ k, k'_b = Sk_6 \ k, g : Ct \ k \ (\text{Either } k_a \ k_b), h : Ct \ (\text{Either } k_a \ k_b) \ k,
\]
\[
g' : Ct \ k \ (\text{Either } k'_a \ k'_b), h' : Ct \ (\text{Either } k'_a \ k'_b) \ k,
\]
\[
t_1 = (\text{Trie } k_{11} \ v_1) \rightarrow (\text{Trie } k_{12} \ v_1) \rightarrow (E \ a_{11} \ b_{12}) \rightarrow (\text{Trie } k_1 \ v_1),
\]
\[
t_1 = t_5 \rightarrow t_9 \rightarrow (E \ a_{11} \ b_{12}) \rightarrow t_{10}, t_2 = \text{Trie } k_2 \ v_2 \rightarrow \text{Trie } k_2 \ v_2 \rightarrow \text{Trie } k_2 \ v_2,
\]
\[
t_2 = t_3 \rightarrow b_4 \rightarrow t_5, t_3 = \text{Trie } k_a \ v, a_4 = \text{Trie } k'_a \ v, t_{10} = (\text{Trie } k) \ v,
\]
\[
a_{11} = k, b_{11} = (\text{Either } k''_a \ k''_b), a_{12} = (\text{Either } k''_a \ k''_b), b_{12} = k,
\]
\[
(E \text{Either } k_a \ k_b) = (\text{Either } k''_a \ k''_b), 15 : CtM \ (\text{Trie } Sk_a) \ v \ (\text{Trie } Sk_a) \ v,
\]
\[
16 : CtM \ (\text{Trie } Sk'_a) \ v \ (\text{Trie } Sk'_a) \ v,
\]
\[
22 : CtM \ (\text{Either } Sk_a \ Sk'_a) \ (\text{Either } Sk_a \ Sk'_a),
\]
\[
23 : CtM \ (\text{Either } Sk_a \ Sk'_a) \ (\text{Either } Sk_a \ Sk'_a), 24 : CtM \ k'_a \ k_2
\]

\[\cdots\]
merge ...

let castm_{11} = castm_{13} \circ g

\ldots

\text{castm}_{18} \ (g, h) = ((\text{castm}_{19} \ g), (\text{castm}_{20} \ h))
\text{castm}_{19} \ g \ x = \text{castm}_{22} \ (g \ (\text{castm}_{21} \ x))
\text{castm}_{20} \ g \ x = \text{castm}_{24} \ (g \ (\text{castm}_{23} \ x))

in \ Tp \ (merge \ ta \ (castm_{4} \ ta')) \ (merge \ tb \ (castm_{8} \ tb'))

(E \ (castm_{11}, castm_{12}))

\Rightarrow_{\text{Trans}}\
\begin{align*}
t = Trie \ k \ v & \rightarrow Trie \ k \ v \rightarrow Trie \ k \ v, v = Sk_1, k = Sk_2, k_a = Sk_3, k_b = Sk_4, k, \k_a' = Sk_5, k_b' = Sk_6, k, b_1 = Trie \ k \ v, a_2 = Trie \ k_1 \ v, b_2 = Trie \ k_2 \ v, \\
a_3 = Trie \ k_3 \ v & \rightarrow Trie \ k_3 \ v \rightarrow Trie \ k_3 \ v, a_4 = Trie \ k_a \ v, a_5 = Trie \ k_a' \ v, \b_3 = b_4 \rightarrow b_5 \rightarrow a_2, g : Ct \ k \ (Either \ k_a \ k_b), h : Ct \ (Either \ k_a \ k_b) \ k, \\
g' : Ct \ k \ (Either \ k_a' \ k_b'), h' : Ct \ (Either \ k_a' \ k_b') \ k, \\
g \circ h : Ct \ (Either \ k_a \ k_b) \ (Either \ k_a' \ k_b'), \g \circ h' : Ct \ (Either \ k_a' \ k_b') \ (Either \ k_a \ k_b), \t_1 = (Trie \ k_{11} \ v_1) \rightarrow (Trie \ k_{12} \ v_1) \rightarrow (E a_{11} \ b_{12}) \rightarrow (Trie \ k_1 \ v_1), \\
t_1 = t_5 \rightarrow t_9 \rightarrow (E a_{11} \ b_{12}) \rightarrow t_{10}, t_2 = Trie \ k_2 \ v_2 \rightarrow Trie \ k_2 \ v_2 \rightarrow Trie \ k_2 \ v_2, \\
t_2 = t_3 \rightarrow b_4 \rightarrow t_5, t_3 = Trie \ k_a \ v, a_4 = Trie \ k_a' \ v, t_{10} = (Trie \ k \ v), \\
a_{11} = k, b_{11} = (Either \ k_a' \ k_b'), a_{12} = (Either \ k_a' \ k_b'), b_{12} = k, \(Either \ k_a \ k_b) = (Either \ k_a' \ k_b'), 15 : CtM \ (Trie \ Sk_a \ v) \ (Trie \ Sk_a \ v), \\
16 : CtM \ (Trie \ Sk_a' \ v) \ (Trie \ Sk_a' \ v), 21 : CtM \ k_2 \ k_a', \\
22 : CtM \ (Either \ Sk_a \ Sk_a') \ (Either \ Sk_a \ Sk_a'), \\
23 : CtM \ (Either \ Sk_a \ Sk_a') \ (Either \ Sk_a \ Sk_a'), 24 : CtM \ k_a' \ k_2
\ldots
\end{align*}
6.3 Hinze and Cheney Trie Example

⇝

Either

..., 16 : \(\text{CtM} (\text{Trie} \ Sk_2 v) (\text{Trie} \ Sk'_2 v)\), 21 : \(\text{CtM} k_2 k'_a\),

..., 22 : \(\text{CtM} (\text{Either} \ Sk_2 Sk'_2) (\text{Either} \ Sk_a Sk'_a)\),

..., 23 : \(\text{CtM} (\text{Either} \ Sk_2 Sk'_2) (\text{Either} \ Sk_a Sk'_a)\), 24 : \(\text{CtM} k'_a k_2\)

..., 24

,...

let castm11 = castm13 \circ g

..., 20

castm20 g x = castm24 (g (castm23 x))

in Tp (merge ta (castm1 ta')) (merge tb (castm8 tb'))

(E (castm11, castm12))
6.3 Hinze and Cheney Trie Example

\[
\begin{align*}
\rightarrow_{Trans1} \quad t &= \text{Trie } k \ v \rightarrow \text{Trie } k \ v \rightarrow \text{Trie } k \ v, v = S k_1, k = S k_2, k_a = S k_3, k_b = S k_4, k, \\
    k'_a &= S k_5, k'_b = S k_6, k, b_1 = \text{Trie } k \ v, a_2 = \text{Trie } k_1 \ v, b_2 = \text{Trie } k_2 \ v, \\
a_3 &= \text{Trie } k_3 \ v \rightarrow \text{Trie } k_3 \ v \rightarrow \text{Trie } k_3 \ v, a_4 = \text{Trie } k_a \ v, a_5 = \text{Trie } k'_a \ v, \\
b_3 &= b_4 \rightarrow b_5 \rightarrow a_2, g : \text{Ct } k (\text{Either } k_a \ k_b), h : \text{Ct } (\text{Either } k_a \ k_b) k, \\
g' : \text{Ct } k (\text{Either } k'_a \ k'_b), h' : \text{Ct } (\text{Either } k'_a \ k'_b) k, \\
g' \circ h : \text{Ct } (\text{Either } k_a \ k_b) (\text{Either } k'_a \ k'_b), \lambda x. \text{project}_L (g' \circ h(\text{inject}_L \ x)) : \text{Ct } k_a \ k'_a, \\
\lambda x. \text{project}_R (g' \circ h(\text{inject}_R \ x)) : \text{Ct } k_b \ k'_b, g \circ h' : \text{Ct } (\text{Either } k'_a \ k'_b) (\text{Either } k_a \ k_b), \lambda x. \text{project}_L (g \circ h'(\text{inject}_L \ x)) : \text{Ct } k'_a \ k_a, \\
t_1 &= (\text{Trie } k_{11} \ v_1) \rightarrow (\text{Trie } k_{12} \ v_1) \rightarrow (E \ a_{11} \ b_{12}) \rightarrow (\text{Trie } k_1 \ v_1), \\
t_1 &= t_5 \rightarrow t_9 \rightarrow (E \ a_{11} \ b_{12}) \rightarrow t_{10}, t_2 = \text{Trie } k_2 \ v_2 \rightarrow \text{Trie } k_2 \ v_2 \rightarrow \text{Trie } k_2 \ v_2, \\
t_2 &= t_3 \rightarrow b_4 \rightarrow t_5, t_3 = \text{Trie } k_a \ v, a_4 = \text{Trie } k'_a \ v, t_{10} = (\text{Trie } k \ v), \\
a_{11} &= k, b_{11} = (\text{Either } k'_a \ k'_b), a_{12} = (\text{Either } k'_a \ k'_b), b_{12} = k, \\
(Either \ k_a \ k_b) &= (\text{Either } k'_a \ k'_b), k_2 = k_a, 15 : \text{CtM } (\text{Trie } S k_a \ v) (\text{Trie } S k_a \ v), \\
16 : \text{CtM } (\text{Trie } S k'_a \ v) (\text{Trie } S k_a \ v), \\
25 : \text{CtM } k'_a \ k'_b, 22 : \text{CtM } (\text{Either } S k_a \ S k'_a) (\text{Either } S k_a \ S k'_a), \\
23 : \text{CtM } (\text{Either } S k_a \ S k'_a) (\text{Either } S k_a \ S k'_a), 26 : \text{CtM } k_a \ k_2 \\
\ldots
\end{align*}
\]

\[
\begin{align*}
\mapsto \quad \text{merge} \ldots &= \\
\text{let castm}_{11} &= \text{castm}_{13} \circ g \\
\text{...} \\
\text{castm}_{20} \ g \ x &= \text{castm}_{24} (g (\text{castm}_{23} \ x)) \\
\text{castm}_{24} &= \text{castm}_{25} (\lambda x \rightarrow \text{project}_L (g' \circ h(\text{inject}_L \ x))) \\
\text{castm}_{25} &= \text{castm}_{26} (\lambda x \rightarrow \text{project}_L (g \circ h'(\text{inject}_L \ x))) \\
in \ Tp (\text{merge } t a (\text{castm}_{1} \ t_2)) (\text{merge } t b (\text{castm}_{8} \ t_2)) \\
(E (\text{castm}_{11}, \text{castm}_{12}))
\end{align*}
\]
The final translation is

\[
\text{merge : : Trie k v -> Trie k v -> Trie k v} \\
\text{merge (Tp ta tb e @(E (g,h))) (Tp ta' tb' e' @(E (g',h'))) =}
\]

\[
data Trie k v = \\
\forall k1 k2. Tp (Trie k1 v) (Trie k2 v) (E k (Either k1 k2))
\]

\[
\text{data Trie k v =}
\]

\[
\forall k1 k2. Tp (Trie k1 v) (Trie k2 v) (E k (Either k1 k2))
\]
let let \( \text{castm}_11 = \text{castm}_{13} \circ g \)
\( \text{castm}_12 = \text{castm}_{14} \circ h \)
\( \text{castm}_{13} x = x \)
\( \text{castm}_{14} x = x \)
\( \text{castm}_4 (Tp a b e) = Tp (\text{castm}_{15} a) (\text{castm}_{16} b) (\text{castm}_{17} e) \)
\( \text{castm}_{17} (E (g, h)) = E (\text{castm}_{18} (g, h)) \)
\( \text{castm}_{18} (g, h) = ((\text{castm}_{19} g), (\text{castm}_{20} h)) \)
\( \text{castm}_{19} g x = \text{castm}_{22} (g (\text{castm}_{21} x)) \)
\( \text{castm}_{20} g x = \text{castm}_{24} (g (\text{castm}_{23} x)) \)
\( \text{castm}_{21} = \text{castm}_{25} (\lambda x \rightarrow \text{project}_L (g' \circ h \text{inject}_L x)) \)
\( \text{castm}_{24} = \text{castm}_{26} (\lambda x \rightarrow \text{project}_L (g \circ h' \text{inject}_L x)) \)
\( \text{castm}_{15} x = x \)
\( \text{castm}_{16} x = x \)
\( \text{castm}_{25} x = x \)
\( \text{castm}_{22} x = x \)
\( \text{castm}_{23} x = x \)
\( \text{castm}_{26} x = x \)
\( \text{castm}_8 (Tp a b e) = \ldots \)
\( \text{in} \ Tp (\text{merge} \text{ta} (\text{castm}_4 \text{ta}')) (\text{merge} \text{tb} (\text{castm}_8 \text{tb}')) \)
\( (E (\text{castm}_{11}, \text{castm}_{12})) \)
Conclusion

We showed that GRDT typing behavior can already be expressed in terms of type classes with existential types. The problem is that common implementations such as GHC fail to accept the resulting program. Therefore, we introduced a translation method from GRDTs to existential types where the resulting program is accepted by GHC. For the translation method to be successful we reject GRDT programs which make use of \textit{False} assumptions and require that types must be decomposable. We introduced a novel CHR-based proof term construction method. There are some connections to methods for finding paths in graphs and “ask” constraints which appear in the context of constraint-logic programming \cite{9}. We yet need to work out the exact details. We showed how to combine our method with a previously introduced constraint solving approach \cite{18} for constructing typing derivations for type classes with existential types. Each successful solution immediately allows us to construct a well-typed ET program which is equivalent to the original GRDT program.


[7] C. V. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in


Appendix A

Semantics of Expressions

We follow the ideal semantics of MacQueen, Plotkin and Sethi [13]. 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 $W$, 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

$$\mathcal{V} = W \perp + \mathcal{V} \to \mathcal{V}.$$ 

The meaning function on terms is as follows:

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

$$[\lambda u.e]_\eta = \lambda v. [e]_\eta[u := v]$$

$$[e \ e']_\eta = \begin{cases} [e]_\eta \in \mathcal{V} \to \mathcal{V} \land [e']_\eta \neq W \\
then ([e]_\eta)([e']_\eta) \\
else W \end{cases}$$
\[
[\text{let } x = e \text{ in } e']_\eta = \begin{cases} 
[\eta] & \text{if } [e]_\eta \neq \text{W} \\
[\eta]_x = [e]_\eta & \text{else}
\end{cases}
\]

Note that the above semantics is call–by value.
Appendix B

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

Consider rule instance (Arrow) \( i : CtM \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2) \iff i_1 : CtM \ b_1 \ a_1, i_2 : CtM \ a_2 \ b_2 \) such that \((i : CtM \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2))_j \in C\). Then, \( C \rightarrow_{Arrow} C - (i : CtM \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2))_j, (i_1 : CtM \ b_1 \ a_1)_j, (i_2 : CtM \ a_2 \ b_2)_j \). Similarly, we define justified (T) rule applications.
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 \(C = \{f_1: Ct \ a_1 \ b_1, ..., f_n: Ct \ a_n \ b_n\}\), \(i: CtM \ a \ b, C \rightarrow^* D_1\) and \(castm_i \rightsquigarrow^* e_1\) and \(i: CtM \ a \ b, C \rightarrow^* D_2\) and \(castm_i \rightsquigarrow^* e_2\) such that both CHR derivations are good. We say that \(i: CtM \ a \ b, C \rightarrow^* D_1\) is shorter than \(i: CtM \ a \ b, C \rightarrow^* D_2\) 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 \(CtM\)s we compare the sum of the individual sizes of resulting expressions.

**Lemma 7** Let \(i: CtM \ t \ t, C \rightarrow^* D\) be a good derivation. Then, \(castm_i \rightsquigarrow^* e\) where \(e\) is equivalent to the identity.

**Lemma 8** 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.

\[
\begin{align*}
C \\
\Rightarrow & \quad \ldots \\
\Rightarrow & \quad C_1, (g : Ct t_1 t_2)_{t_1}, (i : CtM t'_1 t''_2)_{L_1} \quad l_1 \not\in L_1 \\
\Rightarrow_{\text{Trans1}} & \quad C_1, (g : Ct t_1 t_2)_{t_1), (j : CtM t_2 t''_2)_{(t_1) \cup L_1}, t_1 = t'_1 \quad (1) \\
\Rightarrow & \quad \ldots \\
\Rightarrow & \quad C_2, (g : Ct t_1 t_2)_{t_1}, (k : CtM t''_1 t''_2)_{L_2} \quad l_1 \in L_2 \quad (2) \\
\Rightarrow_{\text{Trans1}} & \quad C_2, (g : Ct t_1 t_2)_{t_1}, (n : CtM t_2 t''_2)_{L_2}, t_1 = t''_1 \\
\Rightarrow & \quad \ldots \\
\Rightarrow & \quad D
\end{align*}
\]

W.l.o.g., in the derivation steps between (1) and (2) we only apply
CHR on \((j : CtM t_2 t''_2)_{(t_1) \cup L_1}\) or its successors, i.e. those resulting
from (Trans1), (T) and (Arrow) rules.

First, we show that only (Trans1) or (Id) rules could have been ap-
plied on \((j : CtM t_2 t''_2)_{(t_1) \cup L_1}\) or its successors. Assume the con-
trary, that is some (T) (or (Arrow)) rule has been applied. E.g., as-
sume that the (Pair) rule, a special case of (T), has been applied on
\((j : CtM t_2 t''_2)_{(t_1) \cup L_1}\). Then,

\[
\begin{align*}
\ldots, (g : Ct t_1 t_2)_{t_1}, t_2 = (t_3, t_4), t'_2 = (t_5, t_6), \ (j : CtM t_2 t''_2)_{(t_1) \cup L_1} \\
\Rightarrow_{\text{Pair}} & \quad \ldots, (g : Ct t_1 t_2)_{t_1}, t_2 = (t_3, t_4), t'_2 = (t_5, t_6), \\
& \quad (j_1 : CtM t_3 t_5)_{(t_1) \cup L_1}, (j_2 : CtM t_4 t_6)_{(t_1) \cup L_1}
\end{align*}
\]

However, then we obtain a cycle among types. E.g., assume that \((j_1 : CtM t_3 t_5)_{(t_1) \cup L_1}\) 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 \((g : Ct t_1 (t_1, t_4))_{t_1}\).
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 7. Hence, the steps between (1) and (2) are redundant. Hence, we obtain a shorter derivation. □

**Lemma 9**  
CHR\(_s\) 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. □
Proofs

C.1 Proof of Theorem 1

Definition 6 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 $(\forall t. t = t' \in C \iff (Ct t t' \in C' \land Ct t' t \in C'))$. We call $C'$ the "Ct" equivalent of $C$; and $C$ the "Eq" equivalent of $C'$.

Lemma 10 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 $C \vdash^= C_1 = t_2$ 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 derivation.

Case:

$\frac{t = t' \in C}{C \vdash^= t = t'}$
Because we have $t = t' \in C$, we know $C t t' \in C'$ and $C t' t \in C'$.
Thus $P_p \models C' \supset (C t t', C t' t)$.

$\circ$ Case:

\[
\begin{align*}
C \vdash^= t_1 &= t_2 & C' \vdash^= t_2 &= t_3 \\
C \vdash^= t_1 &= t_3
\end{align*}
\]

By induction, we have

\[
P_p \models C' \supset (C t_1 t_2, C t_2 t_1, C t_2 t_3, C t_3 t_2)
\]

By the type class instance

\[
\forall a_1, a_3.(C t_1 a_3 \leftrightarrow \exists a_2.(C t_1 a_2 \land C t_2 a_3))
\]

We conclude

\[
P_p \models C' \supset (C t_1 t_2, C t_2 t_1, C t_2 t_3, C t_3 t_2) \supset (C t_1 t_3, C t_3 t_1)
\]

$\circ$ Case:

Other cases are similar.

(Direction $\Leftarrow$)

$\circ$ Case:

Suppose the type class instance

\[
\forall a.(C t a a \leftrightarrow True)
\]

is applied. Then we have

\[
P_p \models True \supset C t t
\]
We also have

\[ \text{True} \vdash_{=} t = t \]

\( \circ \) **Case:**

Suppose the type class instance

\[
\forall a_1, a_3. (\text{Ct } a_1 a_3 \leftrightarrow \exists a_2. (\text{Ct } a_1 a_2 \land \text{Ct } a_2 a_3))
\]

is applied. Then we have

\[ P_p \models \exists t_2. (\text{Ct } t_1 t_2 \land \text{Ct } t_2 t_3) \supset \text{Ct } t_1 t_3 \]

Easily, we also obtain

\[ t_1 = t_2 \land t_2 = t_3 \vdash_{=} t_1 = t_3 \]

\( \circ \) **Case:**

Other cases are similar. \(\square\)

**Lemma 11** \( C, \Gamma \vdash_{T \text{cast}} t \rightarrow t' \) iff \( P_p \models C \supset \text{Ct } t t' \)

**Proof:** Follows directly from the rule (M). \(\square\)

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

**Lemma 12** Let \( e \) be a GRDT expression and \( e' \) be its fully casted version. Let \( P_p \) a full and faithful program theory representing all GRDTs type constructors mentioned in \( e \). Silently, we transform the GRDT constructors mentioned in \( e \) to TCET constructors. We have that \( C, \Gamma \vdash_{G_e} e : t \) iff \( C', \Gamma \vdash_{T} e' : t \) where \( C' \) is the “Ct” equivalent of \( C \).

**Proof:** The proof is done in two directions.

\( (\text{Direction } \Rightarrow) \) We proof by induction on derivation.
\(\circ\) Case \((\text{Eq})\):

\[
\frac{C, \Gamma \vdash^{G_e} e : t \quad C \vdash^= e \ t = t'}{C, \Gamma \vdash^{G_e} e : t'}
\]

By the induction hypothesis, we have

\[
C', \Gamma \vdash^T e' : t \quad \text{(1)}
\]

Also by Lemma 10 and \(C \vdash^= e \ t = t'\) we have

\[
P_p \vdash C' \supset (Ct \ t'', Ct \ t' \ t) \quad \text{(2)}
\]

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 \(t \rightarrow t'\) and the second \(t'' \rightarrow t\). Thus by Lemma 11, we know that \(Ct \ t'\) and \(Ct \ t'' \ t\) can be derived from the context. By the \((\text{Trans})\) type class instance, we can derive \(Ct \ t'' \ t'\). Then by Lemma 11, 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

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

This is equivalent to

\[
C', \Gamma \vdash^T e' : t'
\]
C.1 Proof of Theorem 1

◦ Case (App):

\[
\begin{array}{c}
\frac{C, \Gamma \vdash e_1 : t_2 \rightarrow t \quad C, \Gamma \vdash e_2 : t_2}{C, \Gamma \vdash e_1 e_2 : t}
\end{array}
\]

By the induction hypothesis, we have

\[
C', \Gamma \vdash e' : t_2 \rightarrow t \quad C', \Gamma \vdash e'' : t_2
\]

By application of rule (App), we obtain

\[
C', \Gamma \vdash (e'_1 e'_2) : t \quad (1)
\]

Note that we always have \( C \vdash e = t = t \). Thus we conclude

\[
C', \Gamma \vdash (\text{cast } (e'_1 e'_2)) : t
\]

◦ Case:
Other cases are similar.

(Direction ⇔)
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 \( e' \equiv (\text{cast } e'') \).

◦ \( e'' = x \)

\[
\begin{array}{c}
\frac{C', \Gamma \vdash \text{cast } : t \rightarrow t' \quad C', \Gamma \vdash e'' : t}{C', \Gamma \vdash (\text{cast } e'') : t'}
\end{array}
\]

Because \( e'' = x \), then \([e''] = e''. \) Therefore, we have

\[
C, \Gamma \vdash^{Ge} [e''] : t \quad (1)
\]
By $C', \Gamma \vdash^T \text{cast} : t \rightarrow t'$ and Lemma 11, we obtain

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

Together with Lemma 10, we have

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

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

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

Because $[\text{cast } e''] = [e'']$, then we have

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

This is equivalent to

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

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

$$\begin{array}{c}
C', \Gamma \vdash^T \text{cast} : t \rightarrow t' \\
C', \Gamma.x : t_1 \vdash^T e'' : t_2 \\
C', \Gamma \vdash^T e'' : t
\end{array}$$

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

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

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

By applying the (Abs) rule, we obtain

$$C, \Gamma \vdash^{G_e} [e''] : t \quad (1)$$
By $C', \Gamma \vdash^T \text{cast} : t \to t'$ and Lemma 11, we obtain

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

Together with Lemma 10, 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 $[\text{cast } e''] = [e'']$, then we have

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

This is equivalent to

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

$\circ e'' = (e''', e''')$

$$C', \Gamma \vdash^T Ct : t \to t' \quad \frac{C', \Gamma \vdash^T e'''_2 : t_2 \to t \quad C', \Gamma \vdash^T e'''_1 : t_2} {C', \Gamma \vdash^T e'' : t}$$

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 $C', \Gamma \vdash^T \text{cast} : t \rightarrow t'$ and Lemma 11, we obtain

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

Together with Lemma 10, we have

$$C \vdash_e t = t' \quad (2)$$

By (1) and (2), we conclude

$$C, \Gamma \vdash G_c [\varepsilon'] : t'$$

Because $[\text{cast } \varepsilon''] = [\varepsilon'']$, then we have

$$C, \Gamma \vdash G_c [\text{cast } \varepsilon''] : t'$$

This is equivalent to

$$C, \Gamma \vdash G_c [\varepsilon''] : t'$$

\begin{itemize}
  \item \textbf{Case:}
  Other cases are similar.
\end{itemize}

\section*{C.2 Proof of Lemma 1}

\textbf{Proof:} The proof is done on induction over the proof term construction derivation. W.l.o.g we combine rule $(\forall E)$ with rules (Id),(Var),(Arrow) and (T). We also combine $(\exists E)$ with (Trans).

\begin{itemize}
  \item \textbf{Case:}(Id)

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

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

  \item \textbf{Case:}(Var)

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

\begin{itemize}
  \item Case: (Trans)
  \[
  f = \lambda g. \lambda x. f_2 (g (f_1 x))
  \]
  
  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 $\Gamma \vdash f : a_1 \rightarrow a_3$.

  \item Case: (Arrow)
  Similar to (Trans).

  \item Case: (T)
  Similar to (Trans).

  \item Case: ($\circ$)

  \[
  f : Ct a b \leftrightarrow f_1 : c_1, \ldots, f_n : c_n \quad f_i : c_i \leftrightarrow F_i \quad F \models F_i \text{ for } i = 1, \ldots, n
  \]

  By induction, we have $\bigcup_i^\circ \Gamma_i \vdash f : a \rightarrow b$. Because $\bigcup_i^\circ \Gamma_i \subseteq \Gamma$ derived from $F \models F_i$, then we conclude $\Gamma \vdash f : a \rightarrow b$.
\end{itemize}

\section*{C.3 Proof of Lemma 2}

\textbf{Proof:} Straightforward proof by construction of $f$. \hfill \Box
C.4 Proof of Theorem 2

Theorem 2 follows directly from the following lemma.

**Lemma 13** Let $C, \Gamma \vdash^T e : t$, $C, \Gamma \vdash^T e \leadsto e'$ and $\Gamma'$ such that $C \leadsto \Gamma'$. Then $\Gamma \cup \Gamma' \vdash^E e' : t$.

**Proof:** The proof is done through induction on derivation.

- **Case (K):**
  
  $$(K : \forall \bar{a}, \bar{b}. D \Rightarrow t \rightarrow T \bar{a}) \leadsto$$
  
  $$(K' : \forall \bar{a}, \bar{b}. t \rightarrow E t_1 t'_1 \rightarrow ... \rightarrow E t_n t'_n \rightarrow T \bar{a})$$
  
  $C, \Gamma \vdash^T e : [\bar{t}/\bar{a}]t \leadsto e'$
  
  $\vdash P_p \mid C \supset (\overrightarrow{g,h}) : [\bar{t}/\bar{a}]D$
  
  $\vdash C, \Gamma \vdash^T Ke : T \bar{t} \leadsto K' e' \overset{E}{\rightarrow} (g,h)$

  By the induction hypothesis, we have

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

  Also we have

  $$K' : \forall \bar{a}, \bar{b}. t \rightarrow E t_1 t'_1 \rightarrow ... \rightarrow E t_n t'_n \rightarrow T \bar{a} \quad (2)$$

  Note that here we assume an ordering among the constraints. Derived from $P_p \mid C \supset (\overrightarrow{g,h}) : [\bar{t}/\bar{a}]D$, we have

  $$g_i : Ct \ t_i \overset{t'_i}{\leftrightarrow} C \text{ and } h_i : Ct' \ t_i \overset{t_i}{\leftrightarrow} C$$

  W.l.o.g we can assume $g_i, h_i \notin \Gamma$. Hence by Lemma 1, we have

  $$\Gamma \cup \Gamma' \vdash^E g_i : t_i \rightarrow t'_i \text{ and } \Gamma \cup \Gamma' \vdash^E h_i : t'_i \rightarrow t_i \text{ where } i = 1 ... n$$
Thus we can obtain that
\[ \Gamma \cup \Gamma' \vdash^E E (g_i, h_i) : E t_i t'_i \text{ where } i = 1 \ldots n \] (3)

From (1),(2),(3) and rule (K), we conclude
\[ \Gamma \cup \Gamma' \vdash^E K' e' E (g, h) : T \bar{t} \]

\circ Case (Reduce):

\[ D \subseteq C \quad f : Ct t_1 t_2 \leftrightarrow D \]
\[ C, \Gamma \vdash^T \text{cast} : t_1 \rightarrow t_2 \leadsto f \]

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 1
\[ \Gamma \cup \Gamma' \vdash^E f : t_1 \rightarrow t_2 \]

\circ Case (Pat):

\[ p : t_1 \vdash \forall \bar{b} (D \vdash \Gamma p \Gamma p') \quad \bar{b} \cap \text{fv}(C, \Gamma, t_2) = \emptyset \]
\[ C \land D, \Gamma \cup \Gamma_p \vdash^T e : t_2 \leadsto e' \]
\[ C, \Gamma \vdash^T p \rightarrow e : t_1 \rightarrow t_2 \leadsto p' \rightarrow e' \]

By the induction hypothesis, we have
\[ \Gamma \cup \Gamma_p \cup \Gamma_C \cup \Gamma_D \vdash^T e' : t_2 \]

where \( C \leadsto \Gamma_C \) and \( D \leadsto \Gamma_D \).

Also by Lemma 14, we have \( p' \vdash \forall \bar{b}.(\Gamma_p \cup \Gamma_D) \). Thus we conclude
\[ \Gamma \cup \Gamma_C \vdash^E p' \rightarrow e' : t_1 \rightarrow t_2 \]

\circ Other cases are standard. \( \Box \)

**Lemma 14**  Given \( p : t_1 \vdash \forall \bar{b} (D \vdash \Gamma_p \Gamma p') \) then \( p' \vdash \forall \bar{b}.(\Gamma_p \cup \Gamma') \) where \( D \leadsto \Gamma' \).

**Proof:** Standard by induction on derivation. \( \Box \)
C.5 Proof of Theorem 3

Theorem 3 follows directly from the following lemma.

Lemma 15 Let $C, \Gamma \vdash^E e : t$ and all types appearing in assumption constraints in intermediate derivations are decomposable. The $C, \Gamma \vdash^E e : t \rightsquigarrow e'$ for some $e'$.

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

- Case (Reduce):

\[
\begin{align*}
D \subseteq C & \quad f : C t_1 t_2 \leftrightarrow D \\
\hline
C, \Gamma \vdash^T \text{cast} : t_1 \rightarrow t_2 \rightsquigarrow f
\end{align*}
\]

Given all the types are decomposable, by Lemma 2, we know $f : C t_1 t_2 \leftrightarrow C$ for some $f$ if $C t_1 t_2 \leftrightarrow C$. Thus the rule (Reduce) always produces a $f$.

- Case:

Other rules are standard. \hfill \Box

C.6 Proof of Lemma 3

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 (T). We also combine ($\exists E$) with (Trans).

- Suppose the rule applied is (Id):

\[
i : CtM a b, C \quad \rightarrow \quad a = b, C \rightarrow^* D'
\]

\[
\text{cast}_{i} \quad \rightsquigarrow \quad \lambda x.x
\]

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):

\[ i : CtM \ a \ b, C \rightarrow \ a_g = a, \ j : CtM \ b_g, C \rightarrow \ D' \]

\[ \text{castm}_i \rightsquigarrow \text{castm}_j \circ g \]

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

\[
\begin{align*}
\text{(Trans)} & \\
\hline
f = \text{castm}_j \circ g & \\
\hline
f : Ct \ a \ b \leftrightarrow g : Ct \ a \ b_g, \text{castm}_j : Ct \ b_g, b
\end{align*}
\]

\[ f : Ct \ a \ b \leftrightarrow D \]

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):

\[ i : CtM \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2), C \rightarrow i_1 : CtM \ b_1 \ a_1, i_2 : CtM \ a_2 \ b_2, C \rightarrow \ D' \]

\[ \text{castm}_i \rightsquigarrow \lambda g.\lambda x.e \text{castm}_i (g (\text{castm}_i \ x)) \]

Also we have

\[
\begin{align*}
\text{(Trans)} & \\
\hline
f = \lambda g.\lambda x.e \text{castm}_i (g (\text{castm}_i \ x)) & \\
\hline
f : Ct \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2) \leftrightarrow \text{castm}_i : Ct \ b_1 \ a_1, \text{castm}_i : Ct \ a_2 \ b_2
\end{align*}
\]

\[ f : Ct \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2) \leftrightarrow D \]

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 \( D \subseteq C \).

(T) is similar to (Arrow). \( \square \)

C.7 Proof of Lemma 4

Proof: W.l.o.g we combine rule \((\forall E)\) with rules \((\text{Id}),(\text{Var}),(\text{Arrow})\) and \((\text{T})\). We also combine \((\exists E)\) with \((\text{Trans})\).
C.7 Proof of Lemma 4

○ Case (Id).

\[ \lambda x.x : Ct a a \leftrightarrow True \]

Then we have

\[ i : CtM a a, C \rightarrow_{Id} a = a, C \]

\[ \text{castm}_i \rightsquigarrow \lambda x.x \]

○ Case (Var).

\[ f : Ct a b \leftrightarrow f : Ct a b \]

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

\[ i : CtM a b, C \rightarrow_{\text{Trans1}} j : CtM b b, C \rightarrow_{Id} C \]

\[ \text{castm}_i \rightsquigarrow \text{castm}_i \circ f \rightsquigarrow \lambda x.x \circ f \]

○ Case (Trans).

\[ (\text{Trans}) \]

\[ f = \lambda x.f_2 (f_1 x) \]

\[ f : Ct a_1 a_3 \leftrightarrow f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3 \]

We have

\[ i : CtM a_1 a_3, f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3 \]

\[ \text{castm}_i \]

\[ \rightarrow_{\text{Trans1}} j : Ct a_2 a_3, f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3 \]

\[ \rightsquigarrow \text{castm}_j \circ f_1 \]

\[ \rightarrow_{\text{Trans1}} k : CtM a_3 a_3, f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3 \]

\[ \rightsquigarrow \text{castm}_k \circ f_2 \circ f_1 \]

\[ \rightarrow_{Id} f_2 \circ f_1 : Ct a_1 a_2, f_2 : Ct a_2 a_3 \]

\[ \rightsquigarrow \lambda x.x \circ f_2 \circ f_1 \]
C.8 Proof of Lemma 5

Proof: Let $f : Ct a b \leftrightarrow C$, from Lemma 3, 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 $e_2$. □
C.9 Proof of Lemma 6

**Proof:** Suppose there are castms left in $e''$, there must be unsolved $CtM$ constraints in the final store $C'_{i}$. In this case, $C'_{i}$ is a ‘bad’ final store which is treated as failure. □

C.10 Proof of Theorem 4

**Proof:** The proof is done by induction on derivation.

- **Case (Abs):**

\[
\Gamma . x : a, e \vdash (e' \downarrow F \ 1\ t)
\]

\[
\Gamma , \lambda x . e \vdash ((cast_{i} (\lambda x . e')) \downarrow F \land i : CtM (a \rightarrow t) \ b \mid b)
\]

Because

$C$ solves $(F \land i : CtM (a \rightarrow t) \ b)$

By induction, we have $\phi(\Gamma . x : a) \cup \Gamma' \vdash^{E} e' : \phi(t_e)$. Thus we derive $\phi(\Gamma) \cup \Gamma' \vdash^{E} \lambda x . e' : \phi(a \rightarrow t_e)$.

Also we have $cast_{i} \leadsto g_{i}$. By Lemma 1, we have $\Gamma' \vdash^{E} g_{i} : \phi(a \rightarrow t_e) \rightarrow b$. Because $b$ is fresh, we conclude

$\phi(\Gamma) \cup \Gamma' \vdash^{E} ((cast_{i} (\lambda x . e')) : \phi(b)$

- **Case (App):**

\[
\Gamma , e_1 \vdash (e'_1 \downarrow F_1 \downarrow t_1) \quad \Gamma , e_2 \vdash (e'_2 \downarrow F_2 \downarrow t_2)
\]

\[
\Gamma , (e_1 e_2) \vdash ((cast_{i} (e'_1 e'_2)) \downarrow F_1 \land F_2 \land t_1 = t_2 \rightarrow a \land i : CtM a b \mid b)
\]

Because

$C$ solves $(F_1 \land F_2 \land t_1 = t_2 \rightarrow a \land i : CtM a b)$
By induction, we have $\phi(\Gamma) \cup \Gamma' \vdash_E e'_1 : \phi(t_1)$ and $\phi(\Gamma) \cup \Gamma' \vdash_E e'_2 : \phi(t_2)$. Thus we derive $\phi(\Gamma) \cup \Gamma' \vdash_E (e'_1 , e'_2) : \phi(a)$. Also we have $\text{cast}_i \rightsquigarrow g_i$. By Lemma 1, we have $\Gamma' \vdash_E g_i : \phi(a) \rightarrow b$. Because $b$ is fresh, we conclude

$$\phi(\Gamma) \cup \Gamma' \vdash_E ((\text{cast}_i (e'_1 e'_2)) : \phi(b))$$

○ Case (Pat):

\[
p \vdash p \forall b. (D \Gamma p \Gamma p' t) \quad \Gamma \cup \Gamma_p, e \vdash (e' \Gamma F e', t_e)
\]

\[
\Gamma, p \rightarrow e \vdash (p' \rightarrow e' \Gamma a = t \rightarrow t_e \land (\bar{b} = \text{Sk} (a, fv(\Gamma))) \land (D \supset F_e) \Gamma a)
\]

Given

$$C \text{ solves } (a = t \rightarrow t_e \land (\bar{b} = \text{Sk} (a, fv(\Gamma))) \land (D \supset F_e))$$

we derive

$$C \land D \text{ solves } F_e$$

Note that $D \rightsquigarrow \Gamma_p$. Thus by induction we have $\phi(\Gamma \cup \Gamma_p) \vdash_E e' : \phi(t_e)$. Also by induction and Lemma 16, we have $p' \vdash_E \forall b. (\Gamma_p \Gamma t)$. Because $\bar{b} = \text{Sk} (a, fv(\Gamma))$, we know that $\bar{b} \cap fv(\phi(C)) = \emptyset$. Also we know that $\bar{b} \cap fv(\phi(\Gamma)) = \emptyset$. Thus we derive $\phi(\Gamma \cup \Gamma' \vdash_E p' \rightarrow e' : \phi(a)$.

○ Case Other cases are standard.

\[\square\]

Lemma 16 Given $p \vdash (D \Gamma p \Gamma t)$. Then $p' \vdash_E (\Gamma t)$.

Proof: The proof is standard by induction on derivation. \[\square\]
C.11 Proof of Theorem 5

Proof: Let $M$ be the set of marked locations, $U$ be the set of unmarked locations and $I$ be the set of all locations.

W.l.o.g. we assume no nested patterns and the $CtM$ constraints are fully substituted by the equality constraints. Then we have

$$F \leftrightarrow C_o \land (C \supset (C_e \land \bigwedge_{i \in I} CtM \; t_i \; t'_i))$$

and

$$F' \leftrightarrow C_o \land (C \supset (C_e \land \bigwedge_{i \in U} CtM \; t_i \; t'_i \land \bigwedge_{i \in M} t_i t'_i = t'_i))$$

Suppose $F'$ is not solvable w.r.t $S$, let it be one of the $CtM$ constraints namely $CtM \; t_i \; t'_i$ where $C \supset CtM \; t_i \; t'_i$ is not solvable.

Because $S$ solves $F$, we can make $F'$ solvable w.r.t $S$ by retracting some equality constraints $t_m = t'_m$ and adding in $CtM \; t_m \; t'_m$, where $m \in M$.

Note that $CtM \; t_i \; t'_i$ is transformed into $CtM \; [t_m/t_m]t_i \; [t'_m/t_m]t'_i$ which $S$ solves $C \supset CtM \; [t_m/t_m]t_i \; [t'_m/t_m]t'_i$. However, we know that if $S$ solves $C \supset CtM \; [t_m/t]t_i \; [t'_m/t]t'_i$, then $S$ solves $C \supset CtM \; t_i \; t'_i$. This contradicts with the assumption that $C \supset CtM \; t_i \; t'_i$ is not solvable w.r.t $S$.

Also guaranteed by the heuristic, we have True solves $C_o \land (C \supset (C_e \land \bigwedge_{i \in M} t_i = t'_i))$. Thus, $S$ solves $C_o \land (C \supset (C_e \land \bigwedge_{i \in M} t_i = t'_i))$.

Together, we conclude $S$ solves $F'$.

\qed