Implementing Cut Elimination: A Case Study of Simulating Dependent Types in Haskell*

Chiyan Chen Dengping Zhu Hongwei Xi

Computer Science Department
Boston University

{chiyan,zhudp,hwxi}@cs.bu.edu

Abstract. Gentzen's Hauptsatz – *cut elimination theorem* – in sequent calculi reveals a fundamental property on logic connectives in various logics such as classical logic and intuitionistic logic. In this paper, we implement a procedure in Haskell to perform cut elimination for intuitionistic sequent calculus, where we use types to guarantee that the procedure can only return a cut-free proof of the same sequent when given a proof of a sequent that may contain cuts. The contribution of the paper is two-fold. On the one hand, we present an interesting (and somewhat unexpected) application of the current type system of Haskell, illustrating through a concrete example how some typical use of dependent types can be simulated in Haskell. On the other hand, we identify several problematic issues with such a simulation technique and then suggest some approaches to addressing these issues in Haskell.

1 Introduction

The type system of Haskell, which was originally based on the Hindley-Milner type system [13], has since evolved significantly. With various additions (e.g., type classes [8], functional dependencies [11], higher-rank polymorphism, existential types), the type system of Haskell has become increasingly more expressive as well as more complex. In particular, type-checking in Haskell is now greatly involved. Though there is so far no direct support for dependent types in Haskell, many examples have appeared in the literature that make interesting use of types in Haskell in capturing the kind of program invariants that are usually caught by making use of dependent types.

Gentzen's sequent calculi [7] LJ (for intuitionistic logic) and LK (for classical logic) have played an essential rôle in various studies such as logic programming and theorem proving that are of proof-theoretical nature. The main theorem of Gentzen, *Hauptsatz*, implies that these sequent calculi enjoy the famous subformula property and are thus consistent. Let us use $\Gamma \vdash A$ for a sequent in the sequent calculus for LJ, where Γ and A represent a sequence of formulas and a formula, respectively. Then Gentzen's Hauptsatz for LJ essentially states that the following rule **(Cut)**:

$$\frac{\Gamma \vdash A_1 \quad \Gamma, A_1 \vdash A_2}{\Gamma \vdash A_2}\ \textbf{(Cut)}$$

* Partially supported by the NSF Grants No. CCR-0224244 and No. CCR-0229480

is admissible or can be eliminated (from a logical derivation that makes use of it). Thus Gentzen's Hauptsatz is also known as *cut elimination theorem*.

While there exist various proofs of cut elimination in the literature, few of them are amenable to mechanization (in formal systems). In [17], three proofs of cut elimination (for intuitionistic, classical and linear sequent calculi, respectively) are encoded in the Elf system [16], which supports logical programming based on the LF Logical Framework [9]. There, logical derivations are represented through the use of higher-order abstract syntax [15], and totality (termination and exhaustive coverage) checks are performed to insure that the encodings indeed correspond to some valid proofs of cut elimination. However, we emphasize that it is out of the scope of the paper to compare functional programming with theorem proving. The cut elimination theorem is merely chosen as an interesting example. We could have, for instance, chosen a different example such as implementing a continuation-passing style (CPS) transformation in a typeful manner [2].

In Haskell, it is difficult to adopt a representation for logic derivations that is based on higher-order abstract syntax as the function space is simply too vast. When compared to Elf, which does not support general recursion, the function space in Haskell is far richer. Therefore, if higher-order abstract syntax is chosen to represent logical derivations, there are to be many representations that do not actually correspond to any logical derivations. Instead, we choose a first-order representation for logical derivations. We are to implement a function *cutDER* such that the type of *cutDER* guarantees that if d_1 and d_2 represent logical derivations of the sequents $\Gamma \vdash A_1$ and $\Gamma, A_1 \vdash A_2$, respectively, then the evaluation of *cutDER* d_1 d_2 always returns a logical derivation of $\Gamma \vdash A_2$ if it terminates. However, there is currently no facility in Haskell allowing us to guarantee that *cutDER* is a total function.

The primary contribution of the paper is two-fold. On the one hand, we present an interesting (and somewhat unexpected) application of the current type system of Haskell, illustrating through a concrete example how some typical use of dependent types can be simulated in Haskell. While it is certainly possible to present a general account for simulating dependent types in Haskell, we feel that such a presentation is not only less interesting but also difficult to follow. The example we present already contains all the nuts and bolts that a programmer needs to use this kind of programming style in general. On the other hand, we identify some problematic issues with such a simulation technique and then make some suggestions to address these issues in Haskell. Overall, we feel that though simulating dependent types in Haskell can occasionally lead to elegant (and often small) examples (which are often called *pearls* in the functional programming community), this programming style seems to have some serious difficulties in handling larger and more realistic examples that require some genuine use of dependent types, and we are to substantiate this feeling by pointing out such difficulties in some concrete Haskell programs.

The rest of the paper is organized as follows. In Section 2, we present some basic techniques developed for simulating dependent types in Haskell. We then give a detailed proof of cut elimination (for the implication fragment of the

```
data EQ a b = EQcon (a -> b) (b -> a)

idEQ :: EQ a a
idEQ = EQcon (\x -> x) (\x -> x)

symEQ :: EQ a b -> EQ b a
symEQ (EQcon to from) = EQcon from to

transEQ :: EQ a b -> EQ b c -> EQ a c
transEQ (EQcon to1 from1) (EQcon to2 from2) =
  EQcon (to2 . to1) (from1 . from2)

pairEQ :: EQ a1 b1 -> EQ a2 b2 -> EQ (a1, a2) (b1, b2)
pairEQ (EQcon to1 from1) (EQcon to2 from2) =
  EQcon (\(x1, x2) -> (to1 x1, to2 x2))
        (\(x1, x2) -> (from1 x1, from2 x2))

fstEQ :: EQ (a1, a2) (b1, b2) -> EQ a1 b1
fstEQ (EQcon to from) = -- bot = let x = x in x
  EQcon (\x -> fst (to (x, bot))) (\x -> fst (from (x, bot)))

sndEQ :: EQ (a1, a2) (b1, b2) -> EQ a2 b2
sndEQ (EQcon to from) = -- bot = let x = x in x
  EQcon (\x -> snd (to (bot, x))) (\x -> snd (from (bot, x)))
```

Fig. 1. Constructing Proofs Terms for Type Equality

intuitionistic propositional logic) in Section 3 and relate it to an implementation in Haskell. We mention some closely related work and then conclude in Section 4.

2 Techniques for Simulating Dependent Types

We present in this section some basic techniques developed for simulating dependent types in Haskell. A gentle introduction to dependent types can be found, for instance, in [15]. Also, an interesting use of dependent types in encoding cut-elimination proofs can be found in [18].

2.1 Proof Terms for Type Equality

In the approach to simulating dependent types that we will present shortly, a key step is the use of terms in encoding equality on certain types (the purpose of such encoding will soon be made clear in the following presentation). In Figure 1, we first declare a binary type constructor *EQ*. Given two types τ_1 and τ_2, if there is a term *EQcon* $\underline{to}$ $\overline{from}$ of the type *EQ* τ_1 τ_2, then we can use $\underline{to}$ and $\overline{from}$ to coerce terms of the types τ_1 and τ_2 into terms of the types τ_2 and τ_1, respectively. In fact, we only *intend* to use *EQ* to form a closed type *EQ* τ_1 τ_2 if τ_1 and τ_2 are equal, and *EQcon* to form a closed term *EQcon* $\underline{to}$ $\overline{from}$ if $\underline{to}$ and $\overline{from}$ are (equivalent to) identity functions. However, this cannot be formally enforced. In [1], this issue is addressed by defining $EQ\ \tau_1\ \tau_2$ as $\forall f.f\ \tau_1 \rightarrow f\ \tau_2$. Unfortunately, this definition is not suitable for our purpose as there seems no way of defining functions such as *fstEQ* and *sndEQ* in Figure 1 if this definition is adopted. For instance, suppose that a function F is given of the type $\forall f.f\ (\tau_1, \tau_2) \rightarrow f\ (\tau_1', \tau_2')$; in order to use F to construct a function of the type $\forall f.f\ \tau_1 \rightarrow f\ \tau_1'$, we need to know that the pairing type constructor $(\cdot,\cdot)$ is 1-1 on its first argument; however, it is unclear as to how this information can be expressed in the type system of Haskell.

Conceptually, a term of type $EQ\ \tau_1\ \tau_2$ is intended to show that the types τ_1 and τ_2 are equal. Therefore, we use the name *proof term for type equality* or simply *proof term* for such a term. In Figure 1, we present a proof term *idEQ* and various functions for constructing proof terms. For instance, if pf_1 and pf_2 are proof terms of types $EQ\ \tau_1\ \tau_1'$ and $EQ\ \tau_2\ \tau_2'$, respectively, then *pairEQ* $pf_1\ pf_2$ is a proof term of the type $EQ\ (\tau_1, \tau_2)\ (\tau_1', \tau_2')$; if *pf* is a term of the type $EQ\ (\tau_1, \tau_2)\ (\tau_1', \tau_2')$, then *fstEQ pf* and *sndEQ pf* are proof terms of the types $EQ\ \tau_1\ \tau_1'$ and $EQ\ \tau_2\ \tau_2'$, respectively; if pf_1 and pf_2 are proof terms of types $EQ\ \tau_1\ \tau_2$ and $EQ\ \tau_2\ \tau_3$, respectively, then *transEQ* $pf_1\ pf_2$ is a proof term of the type $EQ\ \tau_1\ \tau_3$.

Let $\underline{\mathrm{TC}}$ be a type constructor that takes n types $\tau_1, \ldots, \tau_n$ to form a type $\underline{\mathrm{TC}}\ \tau_1\ \ldots\ \tau_n$. Then the following rule derives $\underline{\mathrm{TC}}\ \tau_1\ \ldots\ \tau_n \equiv \underline{\mathrm{TC}}\ \tau_1'\ \ldots\ \tau_n'$ from $\tau_1 \equiv \tau_1', \ldots, \tau_n \equiv \tau_n'$,

$$\frac{\tau_1 \equiv \tau_1' \quad \cdots \quad \tau_n \equiv \tau_n'}{\underline{\mathrm{TC}}\ \tau_1\ \ldots\ \tau_n \equiv \underline{\mathrm{TC}}\ \tau_1'\ \ldots\ \tau_n'}\ \textbf{(\underline{tc}iEQ)}$$

where $\equiv$ stands for equality on types, and for each $1 \leq k \leq n$, the following rule derives $\tau_i \equiv \tau_i'$ from $\underline{\mathrm{TC}}\ \tau_1\ \ldots\ \tau_n \equiv \underline{\mathrm{TC}}\ \tau_1'\ \ldots\ \tau_n'$:

$$\frac{\underline{\mathrm{TC}}\ \tau_1\ \ldots\ \tau_n \equiv \underline{\mathrm{TC}}\ \tau_1'\ \ldots\ \tau_n'}{\tau_k \equiv \tau_k'}\ \textbf{(\underline{tc}e}\boldsymbol{k}\textbf{EQ)}$$

We often need a function $\underline{\mathrm{tc}}$iEQ of the following type:

$$EQ\ a_1\ a_1' \rightarrow \ldots \rightarrow EQ\ a_n\ a_n' \rightarrow EQ\ (\underline{\mathrm{TC}}\ a_1 \ldots\ a_n)\ (\underline{\mathrm{TC}}\ a_1' \ldots\ a_n')$$

and functions $\underline{\mathrm{tc}}$ekEQ of the following types,

$$EQ\ (\underline{\mathrm{TC}}\ a_1 \ldots\ a_n)\ (\underline{\mathrm{TC}}\ a_1' \ldots\ a_n') \rightarrow EQ\ a_k\ a_k'$$

where k ranges from 1 to n. We say that $\underline{\mathrm{tc}}$iEQ is the type equality introduction function associated with $\underline{\mathrm{TC}}$ and $\underline{\mathrm{tc}}$ekEQ $(1 \leq k \leq n)$ are type equality elimination functions associated with $\underline{\mathrm{TC}}$. Note that if the binary type constructor *EQ* is defined as $\lambda a.\lambda a'.\forall f.f\ a \rightarrow f\ a'$, then it becomes rather difficult, if not impossible, to define type equality elimination functions.

When presenting some programs in Haskell later, we also need the following functions *toEQ* and *fromEQ*:

```
toEQ :: EQ a b -> (a -> b)
toEQ (EQcon to from) = to

fromEQ :: EQ a b -> (b -> a)
fromEQ (EQcon to from) = from
```

If a proof term *pf* of type $EQ\ \tau_1\ \tau_2$ is given, then *toEQ pf* and *fromEQ pf* act like coercion functions between type τ_1 and type τ_2. We will later point out some actual uses of *toEQ* and *fromEQ*.

2.2 Representing Formulas and Sequences of Formulas

We use P for primitive propositions and $\bot$ for falsehood. The syntax for logic formulas is given as follows,

$$\text{formulas } A ::= P \mid \bot \mid A_1 \land A_2 \mid A_1 \lor A_2 \mid A_1 \supset A_2$$

where the logic connectives are standard. We use $|A|$ for the size of formula A, which is the number of logic connectives in A. We are to use types to encode formulas, so we introduce the following type constructors in Haskell.

```
data BOT = BOT -- encoding falsehood
data LAND a b = LAND a b -- encoding conjunction
data LOR a b = LOR a b -- encoding disjunction
data LIMP a b = LIMP a b -- encoding implication
```

For instance, the type *LIMP BOT* (*LAND BOT BOT*) represents the formula $\bot \supset (\bot \land \bot)$. We do not indicate how primitive propositions are encoded at this moment as this is not important for our purpose. Also, in the following presentation, we only deal with the implication connective $\supset$. It should be clear that the other connectives ($\land$ and $\lor$) can be treated in a similar (and likely simpler) manner. We need the following functions (the type equality introduction function and type equality elimination functions associated with the type constructor *LIMP*) for handling proof terms for type equality:

```
limpiEQ :: EQ a1 a2 -> EQ b1 b2 -> EQ (LIMP a1 b1) (LIMP a2 b2)
limpe1EQ :: EQ (LIMP a1 b1) (LIMP a2 b2) -> EQ a1 a2
limpe2EQ :: EQ (LIMP a1 b1) (LIMP a2 b2) -> EQ b1 b2
```

We omit the actual implementations of these functions, which are similar to the implementations of *pairEQ*, *fstEQ* and *sndEQ*.

We use Γ for a sequence of formulas defined as follows,

$$\text{formula sequences } \Gamma ::= \emptyset \mid \Gamma, A$$

where $\emptyset$ for the empty sequence. We use the unit type () to represent $\emptyset$, and the type (g, a) to represent Γ, A if g and a represent Γ and A, respectively. A judgment of the form $\Gamma \ni A$ means that the formula A occurs in the sequence Γ, we use $\mathcal{I}$ for derivations of such judgments, which can be constructed from applying the following rules:

$$\frac{}{\Gamma, A \ni A}\ \textbf{(one)} \qquad \frac{\Gamma \ni A}{\Gamma, A' \ni A}\ \textbf{(shift)}$$

To represent derivations $\mathcal{I}$ of judgments of the form $\Gamma \ni A$, we would like to declare a datatype constructor *IN* and associate with it two term constructors *INone* and *INshi* of the following types,

$$\mathit{INone} : \forall a.\forall g'.\mathit{IN}\ a\ (g', a) \qquad \mathit{INshi} : \forall a.\forall g'.\forall a'.\mathit{IN}\ a\ g' \to \mathit{IN}\ a\ (g', a')$$

which correspond to the rules **(one)** and **(shift)**, respectively. Such a datatype constructor is called a recursive datatype constructor [19] and is not available in Haskell. Instead, we declare *IN* as follows for representing derivations $\mathcal{I}$:

```
data IN a g =
    forall g'. INone (EQ g (g',a))
  | forall g' a'. INshi (EQ g (g',a')) (IN a g')
```

Essentially, the declaration introduces a binary type constructor *IN* and assigns the two (term) constructors *INone* and *INshi* the following types:

$$\begin{array}{rcl} \mathit{INone} & : & \forall g.\forall a.\forall g'.\mathit{EQ}\ g\ (g',a) \rightarrow \mathit{IN}\ a\ g \\ \mathit{INshi} & : & \forall g.\forall a.\forall g'.\forall a'.\mathit{EQ}\ g\ (g',a') \rightarrow \mathit{IN}\ a\ g' \rightarrow \mathit{IN}\ a\ g \end{array}$$

Assume that types g and a represent Γ and A, respectively. Then a term of the type $\mathit{IN}\ a\ g$ represents a derivation of $\Gamma \ni A$. This probably becomes more clear if the rules **(one)** and **(shift)** are presented in the following manner:

$$\frac{\Gamma = \Gamma', A}{\Gamma \ni A}\ \textbf{(one)} \qquad\qquad \frac{\Gamma = \Gamma', A' \quad \Gamma' \ni A}{\Gamma \ni A}\ \textbf{(shift)}$$

For instance, a derivation of $\Gamma, A_1, A_2, A_3 \ni A_1$ can be represented by the following term:

$$\mathit{INshi}(\mathit{idEQ})(\mathit{INshi}(\mathit{idEQ})(\mathit{INone}(\mathit{idEQ})))$$

We are also in need of the type equality introduction function associated with *IN*, which returns a proof term of type $\mathit{EQ}\ (\mathit{IN}\ \tau_1\ \tau_2)\ (\mathit{IN}\ \tau_1'\ \tau_2')$ when given two proof terms of the types $\mathit{EQ}\ \tau_1\ \tau_1'$ and $\mathit{EQ}\ \tau_2\ \tau_2'$. We define the following function *iniEQ* in Haskell to serve this purpose:

```
iniEQ :: EQ a1 a2 -> EQ g1 g2 -> EQ (IN a1 g1) (IN a2 g2)
iniEQ pf1 pf2 = EQcon to from
  where
    to (INone pf) =
      INone (transEQ (transEQ (symEQ pf2) pf) (pairEQ idEQ pf1))
    to (INshi pf i) =
      INshi (transEQ (symEQ pf2) pf) (toEQ (iniEQ pf1 idEQ) i)

    from (INone pf) =
      INone (transEQ (transEQ pf2 pf) (pairEQ idEQ (symEQ pf1)))
    from (INshi pf i) =
       INshi (transEQ pf2 pf) (toEQ (iniEQ (symEQ pf1) idEQ) i)
```

Please notice some heavy use of proof terms in the implementation of *iniEQ*. We now briefly explain why the first clause in the function *to* is well-typed: *to* needs to be assigned the type $\mathit{IN}\ a_1\ g_1 \rightarrow \mathit{IN}\ a_2\ g_2$; assume that *INone pf* is given the type $\mathit{IN}\ a_1\ g_1$; then *pf* has the type $\mathit{EQ}\ g_1\ (G, a_1)$ for some type G; and it can be verified that $\mathit{transEQ}\ (\mathit{transEQ}\ (\mathit{symEQ}\ \mathit{pf}_2)\ \mathit{pf})\ (\mathit{pairEQ}\ \mathit{idEQ}\ \mathit{pf}_1)$ can be assigned the type $\mathit{EQ}\ g_2\ (G, a_2)$ (assuming pf_1 and pf_2 have the types $\mathit{EQ}\ a_1\ a_2$ and $\mathit{EQ}\ g_1\ g_2$, respectively); so the following term

$$\mathit{INone}\ (\mathit{transEQ}\ (\mathit{transEQ}\ (\mathit{symEQ}\ \mathit{pf}_2)\ \mathit{pf})\ (\mathit{pairEQ}\ \mathit{idEQ}\ \mathit{pf}_1))$$

can be assigned the type $IN\ a_2\ g_2$.

We point out that we seem unable to define the type equality elimination functions associated with *IN*. Fortunately, we do not need these functions when implementing cut elimination.

Definition 1. *Given two sequences Γ and Γ' of formulas, we write $\Gamma \supset \Gamma'$ if $\Gamma' \ni A$ implies $\Gamma \ni A$ for every formula A.*

This definition corresponds to the following type definition or type synonym in Haskell:

```
type SUP g g' = forall a. IN a g' -> IN a g
```

Assume g and g' represent Γ and Γ', respectively. Then a term of the type $SUP\ g\ g'$ essentially represents a proof of $\Gamma \supset \Gamma'$.

2.3 Representing Derivations

To simplify the presentation, we focus on a fragment of intuitionistic propositional logic that only supports the following three logical derivation rules:

$$\frac{\Gamma \ni A}{\Gamma \vdash A}\ \textbf{(AXI)} \qquad \frac{\Gamma \ni A_1 \supset A_2 \quad \Gamma \vdash A_1 \quad \Gamma, A_2 \vdash A}{\Gamma \vdash A}\ (\supset\textbf{L}) \qquad \frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2}\ (\supset\textbf{R})$$

We use $h(\mathcal{D})$ for the height of derivation $\mathcal{D}$, which is defined as usual. In order to represent logical derivations constructed from applying the above three logic derivation rules, we declare a binary datatype *DER* as follows:

```
data DER g a =
    DERaxi (IN a g)
  | forall a1 a2. DERimpl (IN (LIMP a1 a2) g) (DER g a1) (DER (g, a2) a)
  | forall a1 a2. DERimpr (EQ a (LIMP a1 a2)) (DER (g, a1) a2)
```

Clearly, the term constructors *DERaxi*, *DERimpl* and *DERimpr* correspond to the rules **(AXI)**, $(\supset$**L)** and $(\supset$**R)**, respectively. If we can now prove that there is a total function of the following type,

$$\forall g. \forall a_1. \forall a_2. DER\ g\ a_1 \rightarrow DER\ (g, a_1)\ a_2 \rightarrow DER\ g\ a_2$$

then the rule **(Cut)** is admissible in sequent calculus for the implication fragment of intuitionistic propositional logic.

Also, we are to be in need of the type equality introduction function associated with *DER*, which is implemented in Figure 2. We are not able to implement the type equality elimination functions associated with *DER*, and fortunately we will not need them, either.

```
deriEQ :: EQ g1 g2 -> EQ a1 a2 -> EQ (DER g1 a1) (DER g2 a2)
deriEQ pf1 pf2 = EQcon to from
  where
    to (DERaxi i) = DERaxi (toEQ (iniEQ pf2 pf1) i)
    to (DERimpl i d1 d2) = DERimpl i' d1' d2'
      where
        i' = toEQ (iniEQ idEQ pf1) i
        d1' = toEQ (deriEQ pf1 idEQ) d1
        d2' = toEQ (deriEQ (pairEQ pf1 idEQ) pf2) d2
    to (DERimpr pf d') = DERimpr (transEQ (symEQ pf2) pf) d''
      where d'' = toEQ (deriEQ (pairEQ pf1 idEQ) idEQ) d'

    from (DERaxi i) = DERaxi (fromEQ (iniEQ pf2 pf1) i)
    from (DERimpl i d1 d2) = DERimpl i' d1' d2'
      where
        i' = fromEQ (iniEQ idEQ pf1) i
        d1' = fromEQ (deriEQ pf1 idEQ) d1
        d2' = fromEQ (deriEQ (pairEQ pf1 idEQ) pf2) d2
    from (DERimpr pf d') = DERimpr (transEQ pf2 pf) d''
      where d'' = fromEQ (deriEQ (pairEQ pf1 idEQ) idEQ) d'
```

Fig. 2. The type equality introduction function associated with *DER*

2.4 Implementing Some Lemmas

We now show that the following structural rules are all admissible:

$$\frac{\Gamma \vdash A}{\Gamma, A' \vdash A} \qquad \frac{\Gamma, A_1, A_2 \vdash A}{\Gamma, A_2, A_1 \vdash A} \qquad \frac{\Gamma \ni A \quad \Gamma, A \vdash A'}{\Gamma \vdash A'}$$

This should make it clear that the formulation of logic derivation rules presented here is equivalent to, for instance, the one in [18].

Lemma 1. *Assume $\Gamma \supset \Gamma'$. Then $\Gamma, A \supset \Gamma', A$ holds for every formula A.*

Proof. The straightforward proof of the lemma corresponds to the following implementation of *shiSUP* in Haskell:

```
shiSUP :: SUP g g' -> SUP (g, a) (g', a)
shiSUP f = \i -> case i of
  INone pf -> INone (pairEQ idEQ (sndEQ pf))
  INshi pf i -> INshi idEQ (f (toEQ (iniEQ idEQ (symEQ (fstEQ pf))) i))
```

We briefly explain why the first clause in the definition of *shiSUP* is well-typed. Assume that *INone pf* is assigned the type $IN\, b\ (g', a)$, where b is a type variable, and we need to show that *INone* (*pairEQ idEQ* (*sndEQ pf*)) can be assigned the type $IN\, b\ (g, a)$: Note that pf is assigned the type $EQ\ (g', a)\ (G, b)$ for some type G, and thus *pairEQ idEQ* (*sndEQ pf*) can be assigned the type $EQ\ (g, a)\ (g, b)$. Therefore, *INone* (*pairEQ idEQ* (*sndEQ pf*)) can be assigned the type $IN\, b\ (g, a)$. We encourage the reader to figure out the reasoning behind the well-typedness of the second clause in the implementation of *shiSUP*, which is considerably more “twisted”.

Lemma 2. *Assume $\Gamma \supset \Gamma'$ and $\mathcal{D} :: \Gamma' \vdash A$. Then we can construct $\mathcal{D}' :: \Gamma \vdash A$ such that $h(\mathcal{D}') = h(\mathcal{D})$.*

Proof. Note that the lemma implies the admissibility of the following derivation rule:

$$\frac{\Gamma \supset \Gamma' \quad \Gamma' \vdash A}{\Gamma \vdash A}\ \textbf{(Super)}$$

The proof is by structural induction on $\mathcal{D}$, which corresponds to the following implementation of the function *supDER* in Haskell:

```
supDER :: SUP g g' -> DER g' a -> DER g a
supDER f = \d -> case d of
  DERaxi i -> DERaxi (f i)
  DERimpl i d1 d2 ->
    DERimpl (f i) (supDER f d1) (supDER (shiSUP f) d2)
  DERimpr pf d -> DERimpr pf (supDER (shiSUP f) d)
```

Of course, it needs to be verified that the height of the derivation returned by *shiSUP* is the same as that of the one taken as an argument of *shiSUP*.

Lemma 3 (Weakening). *Assume $\mathcal{D} :: \Gamma \vdash A$. Then for each formula A', there exists a derivation $\mathcal{D}' :: \Gamma, A' \vdash A$ such that $h(\mathcal{D}') = h(\mathcal{D})$.*

Proof. Note that the lemma implies the admissibility of the rule **(Weakening)**. The proof is by showing $\Gamma, A' \supset \Gamma$ and then applying Lemma 2, which corresponds to the following implementation of the function *weakDER* in Haskell:

```
weakSUP :: SUP (g, a) g -- the type = forall a'. IN a' g -> IN a' (g, a)
weakSUP i = INshi idEQ i

weakDER :: DER g a -> DER (g, a') a
weakDER = supDER weakSUP
```

Lemma 4 (Exchange). *Assume $\mathcal{D} :: \Gamma, A_1, A_2 \vdash A$. Then there exists a derivation $\mathcal{D}' :: \Gamma, A_2, A_1 \vdash A$ such that $h(\mathcal{D}') = h(\mathcal{D})$.*

Proof. Note that the lemma implies the admissibility of the rule **(Exchange)**. The proof is by showing $\Gamma, A, A' \supset \Gamma, A', A$ and then applying Lemma 2, which corresponds to the following implementation of the function *exchDER* in Haskell:

```
exchSUP :: SUP ((g, a2), a1) ((g, a1), a2)
exchSUP (INone pf) = INshi idEQ (INone (pairEQ idEQ (sndEQ pf)))
exchSUP (INshi pf1 (INone pf2)) =
   INone (pairEQ idEQ (sndEQ (transEQ (fstEQ pf1) pf2)))
exchSUP (INshi pf1 (INshi pf2 i)) = INshi idEQ (INshi idEQ (fromEQ pf' i))
  where pf' = iniEQ idEQ (fstEQ (transEQ (fstEQ pf1) pf2))

exchDER :: DER ((g, a1), a2) a -> DER ((g, a2), a1) a
exchDER = supDER exchSUP
```

Again, some considerably complicated proof terms are used in the implementation of *exchDER*.

Lemma 5 (Contraction). *Assume $\mathcal{D} :: \Gamma, A \vdash A'$. If $\Gamma \ni A$ is derivable, then there exists a derivation $\mathcal{D}' :: \Gamma \vdash A'$ such that $h(\mathcal{D}') = h(\mathcal{D})$.*

```
cutDER :: DER g a1 -> DER (g, a1) a2 -> DER g a2
cutDER d1 d2 = case d2 of
  DERaxi i -> case i of
    INone pf -> toEQ (deriEQ idEQ (sndEQ pf)) d1
    INshi pf i' -> DERaxi (fromEQ (iniEQ idEQ (fstEQ pf)) i')
  DERimpl i d21 d22 -> case i of
    INone pf -> case d1 of
      DERaxi i' -> contractDER i' d2
      DERimpl i' d11 d12 ->
        DERimpl i' d11 (cutDER d12 (exchDER (weakDER d2)))
      DERimpr pf' d1' ->
        cutDER (toEQ (deriEQ idEQ pf1)
                     (cutDER (toEQ (deriEQ idEQ pf0) d') d1'))
               d''
          where
            pf0 = limpe1EQ (transEQ (symEQ (sndEQ pf)) pf')
            pf1 = limpe2EQ (transEQ (symEQ pf') (sndEQ pf))
            d' = cutDER d1 d21
            d'' = cutDER (weakDER d1) (exchDER d22)
    INshi pf i' -> DERimpl i'' d' d''
      where
        i'' = fromEQ (iniEQ idEQ (fstEQ pf)) i'
        d' = cutDER d1 d21
        d'' = cutDER (weakDER d1) (exchDER d22)
  DERimpr pf d2' -> DERimpr pf (cutDER (weakDER d1) (exchDER d2'))
```

Fig. 3. Implementing Cut Elimination

Proof. Note that the lemma implies the admissibility of the rule **(Contraction)**. The proof is by showing that $\Gamma \ni A$ implies $\Gamma \supset \Gamma, A$ and then applying Lemma 2, which corresponds to the following implementation of the function *contractDER* in Haskell:

```
contractSUP :: IN a g -> SUP g (g, a)
contractSUP i = \j -> case j of
  INone pf ->  toEQ (iniEQ (sndEQ pf) idEQ) i
  INshi pf j' -> fromEQ (iniEQ idEQ (fstEQ pf)) j'

contractDER :: IN a g -> DER (g, a) a' -> DER g a'
contractDER i = supDER (contractSUP i)
```

We are now ready to establish that the rule **(Cut)** is admissible.

3 Implementing Cut Elimination

In this section, we prove the admissibility of the rule **(Cut)** in the sequent calculus for the implication fragment of the intuitionistic propositional logic. Meanwhile, we also implement a procedure in Haskell to perform cut elimination, which tightly corresponds to this proof.

Theorem 1 (Admissibility of Cut). *Assume that $\mathcal{D}_1 :: \varGamma \vdash A_1$ and $\mathcal{D}_2 :: \varGamma, A_1 \vdash A_2$. Then there exists a derivation of $\varGamma \vdash A_2$.*

Proof. The proof is by induction on the triple $\langle |A|, h(\mathcal{D}_2), h(\mathcal{D}_1)\rangle$, lexicographically ordered. We proceed by analyzing the structure of $\mathcal{D}_2$.

– $\mathcal{D}_2$ is of the following form:

$$\frac{\mathcal{I} :: \varGamma, A_1 \ni A_2}{\varGamma, A_1 \vdash A_2}\ \textbf{(AXI)}$$

In this case, we analyze the structure of $\mathcal{I}$.

- $\mathcal{I}$ is of the following form,

$$\frac{}{\varGamma, A \ni A}\ \textbf{(one)}$$

where $A = A_1 = A_2$. Then $\mathcal{D}_1$ is a derivation of $\varGamma \vdash A_2$.

- $\mathcal{I}$ is of the following form

$$\frac{\mathcal{I}_1 :: \varGamma \ni A_2}{\varGamma, A_1 \ni A_2}\ \textbf{(shift)}$$

Then a derivation of $\varGamma \vdash A_2$ can be constructed as follows:

$$\frac{\mathcal{I}_1 :: \varGamma \ni A_2}{\varGamma \vdash A_2}\ \textbf{(AXI)}$$

– $\mathcal{D}_2$ is of the following form:

$$\frac{\mathcal{I} :: \varGamma, A_1 \ni A_{11} \supset A_{12} \quad \mathcal{D}_{21} :: \varGamma, A_1 \vdash A_{11} \quad \mathcal{D}_{22} :: \varGamma, A_1, A_{12} \vdash A_2}{\varGamma, A_1 \vdash A_2}\ (\supset\textbf{L})$$

We now analyze the structure of $\mathcal{I}$.

- $\mathcal{I}$ is of the following form,

$$\frac{}{\varGamma, A_1 \ni A_{11} \supset A_{12}}\ \textbf{(one)}$$

where $A_1 = A_{11} \supset A_{12}$. In this case, we need to analyze the structure of $\mathcal{D}_1$.

  * $\mathcal{D}_1$ is of the following form:

$$\frac{\mathcal{I}' :: \varGamma \ni A_1}{\varGamma \vdash A_1}\ \textbf{(AXI)}$$

Then applying Lemma 5 to $\mathcal{I}'$ and $\mathcal{D}_2$, we obtain a derivation of $\varGamma \vdash A_2$.

  * $\mathcal{D}_1$ is of the following form:

$$\frac{\mathcal{I}' :: \varGamma \ni A' \supset A'' \quad \mathcal{D}_{11} :: \varGamma \vdash A' \quad \mathcal{D}_{12} :: \varGamma, A'' \vdash A_1}{\varGamma \vdash A_1}\ (\supset\textbf{L})$$

Applying Lemma 3 to $\mathcal{D}_2$, we obtain a derivation $\mathcal{D}_{2w}$ of $\Gamma, A_1, A'' \vdash A_2$. Applying Lemma 4 to $\mathcal{D}_{2w}$, we obtain a derivation $\mathcal{D}_{2we}$ of $\Gamma, A'', A_1 \vdash A_2$. Note that $h(\mathcal{D}_{2we}) = h(\mathcal{D}_{2w}) = h(\mathcal{D}_2)$. By induction hypothesis on $\mathcal{D}_{12}$ and $\mathcal{D}_{2we}$, we have a derivation $\mathcal{D}'$ of $\Gamma, A'' \vdash A_2$. Therefore, we can derive $\Gamma \vdash A_2$ as follows:

$$\frac{\mathcal{I}' :: \Gamma \ni A' \supset A'' \quad \mathcal{D}_{11} :: \Gamma \vdash A' \quad \mathcal{D}' :: \Gamma, A'' \vdash A_2}{\Gamma \vdash A_2}\ (\supset\mathbf{L})$$

∗ $\mathcal{D}_1$ is of the following form:

$$\frac{\mathcal{D}_1' :: \Gamma, A_{11} \vdash A_{12}}{\Gamma \vdash A_{11} \supset A_{12}}\ (\supset\mathbf{R})$$

This is the most interesting case in this proof. By induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_{21}$, we have a derivation $\mathcal{D}'$ of $\Gamma \vdash A_{11}$. Applying Lemma 3 to $\mathcal{D}_1$, we obtain a derivation $\mathcal{D}_{1w}$ of $\Gamma, A_{12} \vdash A_1$. Applying Lemma 4 to $\mathcal{D}_{22}$, we obtain a derivation of $\mathcal{D}_{22e}$ of $\Gamma, A_{12}, A_1 \vdash A_2$. Note that $h(\mathcal{D}_{1w}) = h(\mathcal{D}_1)$ and $h(\mathcal{D}_{22e}) = h(D_{22})$. By induction hypothesis in $\mathcal{D}_{1w}$ and $\mathcal{D}_{22e}$, we have a derivation $\mathcal{D}''$ of $\Gamma, A_{12} \vdash A_2$. By induction hypothesis on $\mathcal{D}'$ and $\mathcal{D}_1'$, we have a derivation $\mathcal{D}'''$ of $\Gamma \vdash A_{12}$, and then by induction hypothesis on $\mathcal{D}'''$ and $\mathcal{D}''$, we have a derivation of $\Gamma \vdash A_2$.

- $\mathcal{I}$ is of the following form:

$$\frac{\mathcal{I}' :: \Gamma \ni A_{11} \supset A_{12}}{\Gamma, A_1 \ni A_{11} \supset A_{12}}\ (\mathbf{shift})$$

Then by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_{21}$, we have a $\mathcal{D}'$ derivation of $\Gamma \vdash A_{11}$. Applying Lemma 3 to $\mathcal{D}_1$, we obtain a derivation $\mathcal{D}_{1w}$ of $\Gamma, A_{12} \vdash A_1$. Applying Lemma 4 to $\mathcal{D}_{22}$, we obtain a derivation $\mathcal{D}_{22e}$ of $\Gamma, A_{12}, A_1 \vdash A_2$. Note that $h(\mathcal{D}_{1w}) = h(\mathcal{D}_1)$ and $h(\mathcal{D}_{22e}) = h(\mathcal{D}_{22})$. By induction hypothesis on $\mathcal{D}_{1w}$ and $\mathcal{D}_{22e}$, we have a derivation $\mathcal{D}''$ of $\Gamma, A_{12} \vdash A_2$. Therefore, a derivation of $\Gamma \vdash A_2$ can be constructed as follows:

$$\frac{\mathcal{I}' :: \Gamma \ni A_{11} \supset A_{12} \quad \mathcal{D}' :: \Gamma \vdash A_{11} \quad \mathcal{D}'' :: \Gamma, A_{12} \vdash A_2}{\Gamma \vdash A_2}\ (\supset\mathbf{L})$$

– $\mathcal{D}_2$ is of the following form,

$$\frac{\mathcal{D}_2' :: \Gamma, A_1, A_{21} \vdash A_{22}}{\Gamma, A_1 \vdash A_{21} \supset A_{22}}\ (\supset\mathbf{R})$$

where $A_2 = A_{21} \supset A_{22}$. Applying Lemma 3 to $\mathcal{D}_1$, we obtain a derivation $\mathcal{D}_{1w}$ of $\Gamma, A_{21} \vdash A_1$ such that $h(\mathcal{D}_{1w}) = h(\mathcal{D}_1)$. Applying Lemma 4 to $\mathcal{D}_2'$, we obtain a derivation $\mathcal{D}_{2e}'$ of $\Gamma, A_{21}, A_1 \vdash A_{22}$ such that $h(\mathcal{D}_{2e}') = h(\mathcal{D}_2')$. By induction hypothesis on $\mathcal{D}_{1w}$ and $\mathcal{D}_{2e}'$, we have a derivation $\mathcal{D}'$ of $\Gamma, A_{21} \vdash A_{22}$. Therefore, a derivation of $\Gamma \vdash A_2$ can be constructed as follows:

$$\frac{\mathcal{D}' :: \Gamma, A_{21} \vdash A_{22}}{\Gamma \vdash A_{21} \supset A_{22}}\ (\supset\mathbf{R})$$

We have covered all the cases in this implication fragment of intuitionistic propositional logic. The presented proof corresponds tightly to the actual Haskell implementation in Figure 3.

We would like to point out that an implementation of cut elimination for the entire intuitionistic propositional logic can be found at [3].

4 Related Work and Conclusion

We have recently seen various interesting examples in which the type system of Haskell is used to capture certain rather sophisticated programming invariants (e.g., some of such examples can be found in [1, 4, 6, 10, 12, 14]). In many of such examples, the underlying theme seems, more or less, to be simulating or approximating some typical use of dependent types through the use of some advanced features in the type system of Haskell. However, we have presented an example that identifies in greater clarity some problematic issues with such a seemingly cute programming style.

In [12], an approach to simulating dependent types is presented that relies on the type class mechanism in Haskell. In particular, it makes heavy use of multi-parameter type classes with functional dependencies [11]. This is an approach that seems entirely different from ours. With this approach, there is no need to manually construct proof terms, which are instead handled automatically through the type class mechanism in Haskell. However, this approach is greatly limited in its ability to simulate dependent types. For instance, it does not even seem possible to handle a simple type constructor like *IN*. On the other hand, we feel that our approach to simulating dependent types is both more intuitive and more general and it can handle the examples in [12] with ease.

The way we use proof terms for type equality clearly bears a great deal of resemblance to the one described in [1], where an approach to typing dynamic typing is proposed. There, the binary type constructor *EQ* is defined as follows,

```
type EQ a b = forall f. f a -> f b
```

taking the view of Leibniz on equality. With this definition, functions such as *idEQ*, *symEQ* and *transEQ* can be elegantly constructed. However, it is impossible to implement type equality elimination functions such as *fstEQ* and *sndEQ* in Haskell. To see the reason, let us assume that x is a variable of the type $\forall f.f(a_1, a_2) \rightarrow f(b_1, b_2)$; in order to construct a term of the type $\forall f.f(a_1) \rightarrow f(b_1)$, we need a function π_1 on types such that $\pi_1(a_1, a_2) = a_1$; but there is no such function on types in Haskell.[1] This is a rather serious limitation when the issue of simulating dependent types is concerned.

Certain use of proof terms for type equality can also be found in [4], where some examples are presented to promote programming with type representations. However, these examples, which are relatively simple, do not involve extensive use of proof terms for type equality. In particular, no examples there

[1] It would actually be problematic to add such a function into Haskell: What would then be something like $\pi_1(int)$? Should it be defined or undefined?

involve any use of type equality elimination functions. As a consequence, the difficulty in constructing proof terms is never clearly mentioned. Only recently is the need for type equality elimination functions identified [5].

Our approach to simulating dependent types seems not amenable to certain changes. For instance, we may declare the type constructor *IN* as follows, interchanging the positions of the two arguments of *IN*:

```
data IN g a =
    forall g'. INone (EQ g (g',a))
  | forall g' a'. INshi (EQ g (g',a')) (IN g' a)
```

However, such a minor change mandates that many proof terms in the implementation of cut elimination be completely reconstructed, which, already a rather time-consuming task, is further exacerbated by the fact that type error reporting in neither (current version of) GHC nor (current version of) Hugs offers much help in fixing wrongly constructed proof terms (except for identifying the approximate location of such terms).

We have pointed out that for some type constructors it is difficult or even impossible to implement the associated type equality elimination functions. A simple and direct approach to address the issue is to treat *EQ* as a *primitive* type constructor. In addition, *idEQ*, *symEQ* and *transEQ* as well as *toEQ* and *fromEQ* can be supplied as primitive functions, and for each type constructor $\underline{\mathrm{TC}}$, the type equality introduction function and type equality elimination functions associated with $\underline{\mathrm{TC}}$ can also be assumed to be primitive. In this way, we not only obviate the need for implementing type equality introduction/elimination functions but can also guarantee that if there is a closed term *pf* of type $EQ\ \tau_1\ \tau_2$, then τ_1 equals τ_2 and the terms *toEQ pf* and *fromEQ pf* are both equivalent to the identity function of the type $\tau_1 \to \tau_1$ (as long as the previously mentioned primitive functions are correctly provided). Of course, this simple approach does not address at all the difficulty in constructing proof terms, for which we may need to introduce guarded datatypes [19] or phantom types [5] into Haskell. However, such an introduction is likely to significantly complicate the (already rather involved) type inference in Haskell, and its interaction with the type class mechanism in Haskell, which is largely unclear at this moment, needs to be carefully investigated. As an comparison, we also include in [3] an implementation of cut elimination theorem written in a language that essentially extends ML with guarded recursive datatypes [19]. Clearly, this is a much cleaner implementation when compared with the one in Haskell.

In summary, we have presented an interesting example to show how certain typical use of dependent types can be simulated through the use of some advanced features of Haskell. When compared to various closely related work, we have identified in clearer terms some problematic issues with such a simulation technique, which we hope can be of help for the further development of Haskell or other similar programming languages.

References

1. Baars, A. I., and Swierstra, S. D. Typing Dynamic Typing. In *Proceedings*

of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP '02) (Pittsburgh, PA, October 2002).

2. Chen, C., and Xi, H. Implementing Typeful Program Transformations. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics Based Program Manipulation* (San Diego, CA, June 2003), pp. 20–28.
3. Chen, C., Zhu, D., and Xi, H. Implementing Cut Elimination: A Case Study of Simulating Dependent Types in Haskell, October 2003. Available at: `http://www.cs.bu.edu/~hwxi/academic/papers/CutElim`.
4. Cheney, J., and Hinze, R. A Lightweight Implementation of Generics and Dynamics. In *Proceedings of Haskell Workshop* (Pittsburgh, PA, October 2002), ACM Press, pp. 90–104.
5. Cheney, J., and Hinze, R. Phantom Types. Technical Report CUCIS-TR2003-1901, Cornell University, 2003. Available as `http://techreports.library.cornell.edu:8081/ Dienst/UI/1.0/Display/ cul.cis/TR2003-1901`.
6. Fridlender, D., and Indrika, M. Do we need dependent types? *Functional Pearl in the Journal of Functional Programming 10*, 4 (July 2000), 409–415.
7. Gentzen, G. Untersuchungen über das logische Schlieβen. *Mathematische Zeitschrift 39* (1935), 176–210, 405–431.
8. Hall, C. V., Hammond, K., Jones, S. L. P., and Wadler, P. L. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems 18*, 2 (March 1996), 109–138.
9. Harper, R. W., Honsell, F., and Plotkin, G. D. A framework for defining logics. *Journal of the ACM 40*, 1 (January 1993), 143–184.
10. Hinze, R. Manufacturing Datatypes. *Journal of Functional Programming 11*, 5 (September 2001), 493–524. Also available as Technical Report IAI-TR-99-5, Institut für Informatik III, Universität Bonn.
11. Jones, M. P. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming* (Berlin, Germany, 1999), vol. 1782 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 230–244.
12. McBride, C. Faking It. *Journal of Functional Programming 12*, 4 & 5 (July 2002), 375–392.
13. Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 3 (December 1978), 348–375.
14. Okasaki, C. From Fast Exponentiation to Square Matrices: An Adventure in Types. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming* (September 1999).
15. Pfenning, F. *Computation and Deduction.* Cambridge University Press. (to appear).
16. Pfenning, F. Logic programming in the LF logical framework. In *Logical Frameworks*, I. G. Huet and G. Plotkin, Eds. Cambridge University Press, 1991, pp. 149–181.
17. Pfenning, F. Structural Cut Elimination. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science* (San Diego, CA, June 1995), pp. 156–166.
18. Pfenning, F. Structural Cut Elimination I: intuitionistic and classical logic. *Information and Computation 157*, 1/2 (March 2000), 84–141.
19. Xi, H., Chen, C., and Chen, G. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, LA, January 2003), pp. 224–235.