The Curry-Howard isomorphism adapted for imperative program synthesis and reasoning

Iman Poernomo [1] and John N.Crossley [2]

[1] School of Computer Science and Software Engineering
Monash University, Caulfield, Victoria, Australia 3145.
ihp@csse.monash.edu.au
[2] School of Computer Science and Software Engineering
Monash University, Clayton, Victoria, Australia 3800.
jnc@csse.monash.edu.au

Abstract. The Curry-Howard isomorphism permits the representation of intuitionistic logic as a constructive type theory. It has often been exploited in the implementation of interactive theorem provers. It also forms the basis of the the proofs-as-programs paradigm, an approach to the synthesis of functional programs from intuitionistic proofs (see e.g., [1–3]). In this paper, we present a constructive logical system for reasoning about imperative programs to which the Curry-Howard isomorphism may be adapted. This allows us to take advantage of the isomorphism for theorem proving implementation, and for the synthesis of correct *imperative* programs, following the proofs-as-programs paradigm.
Keywords: Constructive logic, Curry-Howard isomorphism, proofs-as-programs, imperative program synthesis.

1 Introduction

It is a well-known that intuitionistic logic can be presented as a constructive type theory where proofs correspond to terms, formulae to types, logical rules correspond to type inference and proof simplification corresponds to term evaluation. This property, known as the Curry-Howard isomorphism, is often used in the implementation of interactive theorem provers. Amongst other advantages, the isomorphism permits the automation of proof simplification, and complex proof tactics and parametrized lemmata to be given simply as functions over terms.

Various authors have used the Curry-Howard isomorphism for the synthesis of functional programs from constructive proofs [1–3] – an approach known as the *proofs-as-programs paradigm*. The idea is as follows. A description of required program behaviour is given as a formula, P, for which an intuitionistic proof is given. The constructive nature of the proof allows us to transform this proof into a functional program whose behaviour meets the requirements of P. For example, the formula $\forall x : Int.\forall y : Int.\exists z : Int.\gcd(x, y, z)$ can be interpreted as a specification of a program that takes x and y as input values, and returns as output value z, the greatest common divisor of x and y. An intuitionistic proof

of this formula enables the construction of a witness term for z. The proof is then transformed into a functional program f satisfying the formula as its specification. In most modern approaches to proofs-as-programs, the proof is represented as a term of a constructive type theory, with the required program given as a term of a simply typed lambda calculus (such as a subset of the programming language *ML*). The transformation is given as an extraction mapping from the former type theory to the latter.

This paper provides a logical system for reasoning about *imperative* programs, to which the Curry-Howard isomorphism and a proofs-as-programs approach may be adapted. We describe a logic that is similar in use and presentation to Hoare Logic [6], describing the possible range of imperative program side-effects (see below, Section 2) by means of pre- and post-conditions. Our logic uses a natural deduction proof system. Therefore our calculus can be understood as a constructive type theory where proofs correspond to terms and formulae to types.

We can give an extraction map from proof-terms of this type theory to programs in Objective Caml (*OCaml*), a variant of *ML*. This enables *OCaml* programs to be extracted from proofs of their specifications. This approach to synthesis adapts approaches to intuitionistic proofs-as-programs synthesis. In particular we adapt the work of [11, 1], where a notion of *modified realizability* defines when a program of a simply typed lambda calculus satisfies a formula as a specification.

This paper proceeds as follows: In section 2 we present some preliminary syntactic and semantic assumptions about imperative programs. In section 3, we explain how the formulae of our calculus describe side-effect and return values of imperative programs. Our calculus is presented in section 4. Section 5 outlines how we adapt the Curry-Howard isomorphism and proofs-as-programs style to our calculus. We also outline how these results are implemented and we provide a small case study illustrating our work. We briefly review related work and provide concluding remarks in section 6.

2 Preliminary definitions and assumptions

Our results assume a set of pre-defined *OCaml* programs, *Prog*. Our calculus is used to build larger programs from the basic constructs of *OCaml* and this set of (black-box) programs.

Our results are parametrized with respect to a many-sorted algebraic signature $\mathcal{ADT}$. We assume that every closed term of $\mathcal{ADT}$ can be executed as a (side-effect-free) program of *OCaml*, and each sort is a type of *OCaml*. We use `typewriter` font to denote when a $\mathcal{ADT}$ term is to be used in an *OCaml* program, and Times Roman font when terms are used in our calculus. For our present purposes, we take $\mathcal{ADT}$ to include a simply typed lambda calculus over the basic data types of *OCaml* with appropriate predicates, and require disjoint union types: `type 'a 'b disjointUnion=Inl of 'a | Inr of 'b;;`. Given a formula P, and an interpretation ι of the variables $Var_{\mathcal{ADT}}$, we write $\Vdash^{\iota}_{\mathcal{ADT}} P$

when P is true in the intended model (given by the evaluation semantics of *OCaml*) for $\mathcal{ADT}$ under ι and $\Vdash_{\mathcal{ADT}} P$ when $\Vdash^{\iota}_{\mathcal{ADT}} P$ for every ι. For instance, $\Vdash_{\mathcal{ADT}} \forall x : int.\ (fun\ y{-}{>}\ y)(x) = x$ holds, because, for any *OCaml* integer `x`, the lambda application (`fun y->y`)(`x`) is equal to `x`.

Our results are given with respect to a set *StateRef*, whose elements denote possible *OCaml* state references. We consider a subset of *OCaml* programs, *OCaml*(*StateRef*), whose only state references are elements of *StateRef*. In this subset, the state references of *StateRef* can be treated as global, and we assume their declaration as implicit, prior to executing code. For instance, if $\mathtt{r} \in$ *StateRef*, then $\mathtt{r} := !\mathtt{r} + 1; !\mathtt{r};; \in$ *OCaml*(*StateRef*), where we assume that `r` has previously been given a declaration of the form `let r = ref 0`. We assume that the reference types of *Prog* correspond only to data types of $\mathcal{ADT}$. For example, if *int* is a data type of $\mathcal{ADT}$, then `int ref` is a permissible type for elements of *StateRef*.

We represent the *state* of a program's execution by a collection of value assignments, one for each state reference of *StateRef*. Formally, we define the set of states Σ to consist of functions, $\sigma : \mathit{StateRef} \to \mathcal{ADT}$. Each $\sigma \in \Sigma$ represents a possible state of the computer's memory, where $\sigma(\mathtt{s})$ is the value assigned to state reference $\mathtt{s} \in$ *StateRef*. For example, if σ is the state after executing $\mathtt{r} := 40$, then $\sigma(\mathtt{r}) = 40$, because we know that `r` stores an integer reference `ref 40`.

A *side-effect* is the result of executing an imperative program: a transition from an initial state of a computer's memory to a final state. Because *OCaml*(*StateRef*) programs only use global state references from *StateRef*, we can define a simple semantics for representing the side-effect of a program's execution. Given an *OCaml*(*StateRef*) program, we write $\langle \mathtt{p}, \sigma \rangle \hat{\triangleright} \langle \mathtt{l}, \sigma' \rangle$ to mean that, in initial state σ, executing the program `p` will result in the value `l` and a final state σ'. For instance, for $\mathtt{r} \in$ *StateRef*, then $\langle \mathtt{r} := !\mathtt{r} + 1; !\mathtt{r};;, \sigma \rangle \hat{\triangleright} \langle \mathtt{l}, \sigma' \rangle$ holds, when, for instance $\sigma(\mathtt{r}) = 14$ and $\sigma'(\mathtt{r}) = 15$. The execution of an imperative program will produce a whole range of side-effects, depending upon the initial state of the memory. We formally denote such a range by a set-theoretic relation, a *side-effect relations*, between initial and final states. The set of side-effect relations, *Rel*, is defined as a subset of $\mathcal{P}(\Sigma \times \Sigma)$. As usual, we write $\sigma\ R\ \sigma'$ if $(\sigma, \sigma') \in R$. A side-effect relation *defines* the behaviour of an imperative program, when the relation's state pairs consist of the possible initial and final states for the program. Formally, a side-effect relation R defines an *OCaml*(*StateRef*) program `p` when, for any states $\sigma, \sigma' \in \Sigma$, $\sigma\ R\ \sigma' \Leftrightarrow \langle \mathtt{p}, \sigma \rangle \hat{\triangleright} \langle *, \sigma' \rangle$ for some value $*$.

We assume a semantic consequence relation $\Vdash_{\iota}$ that holds between elements of *Rel* and formulae of $\mathcal{ADT}$. Th relation $R \Vdash_{\iota} A$ is to be read as "A is true of the side-effects of R, for the interpretation ι". We then write $\Vdash_{\iota} A$ if $R \Vdash_{\iota} A$ for all side-effects R.

3 Specification of side-effects and return values

Our logic is used to specify and reason about two aspects of imperative program behaviour: possible side-effects and possible return values.

Possible side-effects are specified as pre- and post-conditions, in a single formula style (as opposed to the semantically equivalent Hoare-triple style of [6]). Because a program's side-effect is described in terms of initial and final state reference values, prior to and after execution, these initial and final state reference values are respectively denoted by the name of the state reference with a $()_i$ and with a $()_f$ subscript. For instance, the formula $r_f > r_i$ describes every possible side-effect of the program $\mathtt{r} :=!\mathtt{r} + 1$. It specifies side-effects whose final value of $\mathtt{r}$, denoted by r_f, is greater than the initial value, denoted by r_i.

Possible return values are specified as the required constructive content of a formula, in the same way that functional programs are specified in constructive type theories. So, for instance, the formula $\exists y : int.Prime(y) \wedge y > r_i$ describes a return value f of a program, such that, for any side-effect, f is prime and greater than the final value of $\mathtt{r}$.

Formulae The formulae of our logic are defined as usual. Our quantifiers are sorted, and take sorts from $\mathcal{ADT}$. For instance, if $int \in \mathcal{ADT}$, then $\exists x : int.x = 0$ and $\forall x : int.x = x$ are well-formed. To enable the specification of side-effects, our formulae involve terms of $\mathcal{ADT}$, extended over the subscripted *StateRef* symbols that denote initial and final state reference values. For instance, if $r \in$ *StateRef* then $r_i * 20 + r_f$ is a well-formed term that may be used in our logic.

Specification of side-effects So, in order to define when a formula is true of a program's execution, we define when a formula is true of a side-effect relation from *Rel*. Take a formula P of our calculus. Let σ and σ' be initial and final states for some side-effect relation. We write $P^{\sigma'}_{\sigma}$ for the formula formed from P by replacing every initial state reference value symbol s_i ($\mathtt{s} \in$ *StateRef*) by an actual initial state reference value $\sigma(\mathtt{s})$, and similarly for final state references.

Then, given a relation $R \in$ *Rel* and an initial state σ and a final state σ' such that $\sigma\ R\ \sigma'$, then we write $R \Vdash^{\sigma}_{\iota} P$ when $R \Vdash_{\iota} P^{\sigma'}_{\sigma}$. A formula P is *true of a side-effect relation* R if $R \Vdash^{\sigma}_{\iota} P$ for every initial state σ and every interpretation ι. A formula is *true of a program* if it is true of the relation defining the program.

Specification of return values A theorem of our logic also specifies possible return values by extending the way that a formula of intuitionistic logic specifies a functional program according to the proofs-as-program paradigm. To understand this further, we need the definition of the Skolem form of a formula A, written $Sk(A)$ from [4]. If $\mathtt{p}$ is a pure (side-effect-free) *ML* program, then $\mathtt{p}$ is called an intuitionistic modified realizer of A when $\vdash_{\mathcal{ADT}} Sk(A)[\mathtt{p}/f_A]$.

In the proofs-as-programs approaches of [1] and [11], a formula A specifies a functional program $\mathtt{p}$, if, and only if, the program is an intuitionistic modified realizer of A. The goal of this kind of constructive program synthesis is to transform a proof of a specification into an intuitionistic modified realizer of A.

In our context we say that an *OCaml*(*StateRef*) program $\mathtt{p}$ is a *return-value modified realizer* of a formula, A, when, for every σ, σ' such that $\langle \mathtt{p}, \sigma\rangle \hat{\triangleright} \langle \mathtt{a}, \sigma'\rangle$,

$$L \equiv \texttt{s} := \texttt{t} \mid (l_1;\ \texttt{if}\ [\![A]\!]\ \texttt{then}\ l_2\ \texttt{else if}[\![B]\!]\ \texttt{then}\ l_3) \mid X \mid l_1; l_2 \mid (l_1;\ \texttt{while}\ [\![P]\!]\ \texttt{do}\ l_2;\ \texttt{done}) \mid \texttt{()} \mid \texttt{any} \mid P$$

X ranges over a set of label variables, L_{Var}, $\texttt{s} \in \mathit{StateRef}$, t is any term taken from the closure of $\mathcal{ADT}$ extended by elements $!\texttt{s}$ ($\texttt{s} \in \mathit{StateRef}$), and P ranges over names of pre-programmed *OCaml* programs, *Prog*.

Fig. 1. The grammar for L, our labelling language.

a is an intuitionistic modified realizer of $A^{\sigma'}_{\sigma}$ and we write p rv–mr A. A formula A of our logic specifies the return value of an imperative program p if p rv–mr A.

As a simple example, the Skolem form of the formula $A \equiv \exists y : int.y = r_i{+}1$ is $f_A = r_i{+}1$. It is true that, for any initial state σ, $\langle \texttt{r} :=!\texttt{r} + 1; !\texttt{r}, \sigma\rangle \hat{\triangleright} \langle \sigma(\texttt{r}){+}1, \sigma'\rangle$. Also, $(f_A = r_i + 1)^{\sigma'}_{\sigma}$ is $f_A = \sigma(\texttt{r}) + 1$. So, for every initial state σ, $\sigma(\texttt{r}){+}1$ is an intuitionistic modified realizer of A. In this way, the formula A can be interpreted as a specification of an imperative program that returns an integer value equal to the value of r (prior to execution) plus 1.

4 The Calculus

Labels The labels of the above example can be viewed as executable programs. However, in general, our calculus derives assertions about a *non-executable* language, L, defined in Fig. 1.

Assertions of our logic are of the form $l \bullet P$, consisting of a label l, denoting a side-effect relation, and a formula P that is taken as a statement involving the side-effect relation. Our calculus provides a set of rules for constructing new labelled formulae from known labelled formulae. An example derivation is

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \texttt{s} :=!\texttt{s} + 1 \bullet s_f = s_i + 1 & \texttt{s} :=!\texttt{s} + 1 \bullet s_f = s_i + 1\end{array}}{\texttt{s} :=!\texttt{s} + 1; \texttt{n} :=!\texttt{s}; \texttt{s} :=!\texttt{s} + 1 \bullet n_f = s_i + 1 \wedge s_f = n_f + 1}\ \mathit{seq} \qquad (1)$$

Here, given proofs that $s_f = s_i + 1$ is true of the possible side-effects associated with the label $\texttt{s} :=!\texttt{s} + 1$, then we can prove that $n_f = s_i + 1 \wedge s_f = n_f + 1$ must be true of the possible side-effects associated with the new label $\texttt{s} :=!\texttt{s} + 1; \texttt{n} :=!\texttt{s}; \texttt{s} :=!\texttt{s} + 1$ ($\texttt{s}, \texttt{n} \in \mathit{StateRef}$). Our calculus is similar to Hoare-like systems, whose theorems consist of programs and program behaviour descriptions, built in tandem.

Terms of L are to be understood as representing side-effect relations that may be programmable. This language contains basic *OCaml* imperative constructs and the pre-programmed *Prog* terms. The side-effect relations associated with these constructs are as with actual programs: e.g., $\texttt{s} := \texttt{t}$ should be interpreted as the side-effect relation associated with setting the value of s to t. Our language also contains non-executable terms: any, label variables L_{Var} and terms of the

$$\frac{\begin{array}{c}\Delta \vdash l_1 \bullet (A \vee B)[\bar{s}_f/\bar{v}] \\ \Delta_1, X; l_2 \bullet A \vdash X; l_2 \bullet C \quad \Delta_2, Y; l_3 \bullet B \vdash Y; l_3 \bullet C\end{array}}{\Delta_1, \Delta_2, \Delta \vdash \left(\begin{array}{l} l_1; \\ \texttt{if} \quad [\![A[\bar{s}_f/\bar{v}]]\!] \ \texttt{then}\ l_2 \\ \texttt{else} \\ \quad \texttt{if}\ [\![B[\bar{s}_f/\bar{v}]]\!] \ \texttt{then}\ l_3 \end{array}\right) \bullet C}\ \mathit{if{-}then{-}else}$$

where $\bar{s}_f$ are all the final state identifiers that occur in A or B.

$$\frac{\Delta \vdash \texttt{any} \bullet A}{\Delta \vdash l \bullet A}\ \mathit{any} \qquad \text{where } l \text{ is any label of } L.$$

$$\frac{l_1 \bullet A \quad \mathsf{Frame}(l_2, \bar{s})}{l_1; l_2 \bullet A}\ \mathit{seq}_1 \qquad \frac{\mathsf{Frame}(l_1, \bar{s}) \quad l_2 \bullet A}{l_1; l_2 \bullet A}\ \mathit{seq}_2$$

where $\mathsf{Frame}(l_2, \bar{s})$ is a statement of the form $l_2 \bullet s_f^1 = s_i^1 \wedge \ldots s_f^n = s_i^n$ where $s_f^1, \ldots, s_f^n$ is the set of all final state reference value identifiers in A.

$$\frac{\begin{array}{c}\Gamma_2 \vdash \texttt{any} \bullet (P \vee \neg P)[\bar{s}_i/\bar{v}] \quad \Gamma_3, X \bullet \mathit{Inv}, X; w \bullet P \vdash X; w \bullet \mathit{Inv} \\ \Gamma_1 \vdash \mathit{init} \bullet \mathit{Inv} \quad \Gamma_4, X \bullet \mathit{Inv}, X; w \bullet \neg P \vdash X; w \bullet R\end{array}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \vdash \left(\begin{array}{l} \mathit{init}; \\ \texttt{while}\ [\![P[\bar{s}_i/\bar{v}]]\!] \ \texttt{do} \\ \qquad w; \qquad \texttt{done} \end{array}\right) \bullet R}\ \mathit{loop}$$

provided the following holds. Let t be a term of sort *int* that does not contain any final state reference identifiers, and whose only state reference identifiers are $\bar{s}_i$. Then we have proofs:

$$\begin{array}{c} \vdash \mathit{init} \bullet v \geq 0 \\ w \bullet v \geq 0, w \bullet P[\bar{s}_i/\bar{v}], w \bullet S \vdash w \bullet t[\bar{s}_f/\bar{s}_i] \geq 0 \wedge t[\bar{s}_f/\bar{s}_i] < v \end{array}$$

where $\bar{s}_f$ consists of the corresponding final state reference identifier for each initial identifier of $\bar{s}_i$.

Fig. 2. The structural rules of our calculus.

form $[\![A]\!]$, where A is a formula. The `any` term denotes the supremum of Rel. Variables are intended to denote arbitrary program side-effect relations, and $[\![A]\!]$ terms denote (possibly unknown) boolean functions. The latter terms are used in our conditional- and loop-style constructs. For example, assume `s` and `r` are of type `ref float`, and the predicate $\mathit{IsInteger}(x)$ is defined to mean that `x` : `float` is a whole number. Then, a theorem of our logic is

$$\left(\begin{array}{l} \texttt{s} :=\!\texttt{!r} + 1; \\ \texttt{if} \quad [\![\mathit{Even}(s_f)]\!] \quad \texttt{then s} :=\!\texttt{!s}/2 \\ \texttt{else} \\ \quad \texttt{if} \quad [\![\neg \mathit{Even}(s_f)]\!] \ \texttt{then s} := (\texttt{!s} + 1)/2 \end{array}\right) \bullet \mathit{IsInteger}(s_f)$$

The side-effect relation for this theorem's label should be apparent, apart from the non-executable terms $[\![\mathit{Even}(s_f)]\!]$ and $[\![\neg \mathit{Even}(s_f)]\!]$. These two terms denote

boolean functions that will return `true` depending, respectively, on whether `s` is even or not, after executing $\mathtt{s} :=!\mathtt{r} + 1$. The label is not executable until we have found such boolean functions.[1]

A label can have various interpretations as differing side-effect relations, because of the presence of label variables and $\mathcal{ADT}$ term variables in formula of $[\![A]\!]$ terms. Given any interpretation ι of a label, we can form a unique interpretation of the $\mathcal{ADT}$ term variables, ι_{Var}. For the remainder of this paper, when we write interpretation, we shall assume a pair of interpretations, $\langle \iota : L \to Rel, \iota_{Var} : Var_{\mathcal{ADT}} \to \mathcal{ADT}$, but shall often write ι for either of these interpretations, when the context is clear.

In the semantics of our calculus, a theorem $\vdash l \bullet A$ is interpreted as a truth about the side-effect relation denoted by l: for every state σ, $\iota(l) \Vdash^{\sigma}_{\iota_{Var}} A$. In [10] we show soundness and completeness with respect to this interpretation.

Rules Our calculus has a natural deduction presentation, where $\Gamma \vdash l : A$ means that the labelled formula $l \bullet A$ has been derived from a list of labelled assumption formulae Γ. The basic rules of our natural deduction-style calculus consist of logical rules and structural rules. The logical rules are as usual for a many sorted, intuitionistic first order logic, except that each formula A is now replaced by a labelled formula $w \bullet A$. All of the actual rules may be inferred from our later Fig. 4 or from Fig. 1 of [11]. Here is exists-elimination as an example (here x must not occur free in Q):

$$\frac{\Delta_1 \vdash w \bullet \exists y : T.P \quad \Delta_2, w \bullet P[x/y] \vdash w \bullet Q}{\Delta_1, \Delta_2 \vdash w \bullet Q}\ \exists -E$$

The structural rules are given[2] in Fig. 2. For the purposes of reasoning about an intended model, the calculus can be extended with axioms and schema (including induction schema). For the purposes of this paper, we do not deal with schema. We shall assume a set of axioms $\mathcal{AX}$, consisting of labelled formulae satisfying $l \bullet A \in \mathcal{AX}$ if, and only if, $R \Vdash^{\sigma}_{\iota} A$ for every interpretation.

Logical rules allow us to use intuitionistic logic to reason about a particular side-effect relation. A logical rule involves the same label for premises and conclusion. Thus, a logical rule defines how to conclude new properties from old properties about a a particular side-effect relation.

By contrast, structural rules allow us to build new side-effect relation descriptions from old, and to prove properties of these new descriptions. The $(if-then-else)$ rule allows us to build a conditional label from given labels, and to derive a truth about the corresponding side-effect relation. Similarly, the $(loop)$[3] and (seq_i) $(i = 1, 2)$ rules allow us to construct loop and sequencing labels, deriving truths about the results. The (any) rule tells us that any formula

[1] These are easy to program, but in general, it is not always easy or possible to program the appropriate boolean functions.

[2] For a tuple of states $\bar{\mathtt{s}} = \{\mathtt{s}_1, \ldots, \mathtt{s}_n\}$ or for terms, $\bar{\mathtt{t}} = \{\mathtt{t}_1, \ldots, \mathtt{t}_n\}$, of the same arity and type, we write $\bar{\mathtt{s}} := \bar{\mathtt{t}}$ for $\mathtt{s}_1 := \mathtt{t}_1; \ldots; \mathtt{s}_n := \mathtt{t}_n$.

[3] The conditions for application of the $(loop)$ rule correspond to the requirement for a variant in Hoare logic. They guarantee that the concluding label denotes a side-effect relation for a terminating loop.

that is true of all side-effects must be true of the side-effect relation for any given label.

Given a theorem $l \bullet A$, if we can obtain an *OCaml*(*StateRef*) program $\mathtt{p}$ that has the side-effect relation given by *all* interpretations of l, then we have a program that satisfies A as a specification. In this case, we write $\mathtt{p}\ \mathsf{s}\ l \bullet A$.

Our labels are non-executable definitions of side-effect relations, and consequently, in isolation, they do not readily suggest how to find corresponding programs. Unlike refinement calculi [9], we do not use a refinement processes to derive such a program. Instead, we adapt the proofs-as-programs paradigm, obtaining such imperative programs from proofs of theorems.

5 Adapting the Curry-Howard isomorphism

Our calculus forms a type theory, *LTT*, with proofs represented as terms (called proof-terms), labelled formulae represented as types, and logical deduction given as type inference. The type theory's inference rules are given in Fig. 4 and 5 – we omit the full syntax of proof-terms, but this may be inferred from the inference rules.

We define a proof-term normalization relation $\rightsquigarrow$, defined in Fig. 3, extending the β-reduction of the lambda calculus. As in the Curry-Howard isomorphism for intuitionistic logic, proof simplification is represented by proof-term normalization. For example, the reduction

$$\mathsf{app}((\mathsf{abstract}\ x^{l\bullet A}.\ b^{l\bullet B})^{l\bullet A\to B}, a^{l\bullet A}) \rightsquigarrow b[a/x]^{l\bullet B}$$

corresponds to simplifying a proof of $l \bullet B$ by deleting an occurrence of the ($\to$-I) rule that is immediately followed by an ($\to$-E) inference.

Proof-term normalization does not delete any proof-terms corresponding to structural rules. This corresponds to the fact that a structural rule builds a larger label from given labels, but there are no matching rules for reversing this construction. In a proof, the introduction of a labelled formula through a structural rule cannot be eliminated. Rule 9 of Fig. 3 corresponds to the fact that proofs before and after a structural rule may be simplified, provided the structural rule remains. Thus, the presence of a structural rule represents a form of non-reversible design-decision about the form of the concluding label and, consequently, about the form of the interpreted label's side-effect relation. For example, in the following proof simplification, the structural rule (seq_1) remains fixed, while the pair of ($\to$-I) and ($\to$-E) are deleted:

$$\mathsf{app}((\mathsf{abstract}\ x^{l;m\bullet A}.\ \mathsf{seq}(a, e^{\mathsf{Frame}(m,\bar{s})}, 1)^{l;m\bullet B})^{l;m\bullet A\to B}, a^{l\bullet A}) \rightsquigarrow b[a/x]^{l;m\bullet B}$$

Theorem 1 (Strong normalization). *The theory LTT is strongly normalizing: given any proof-term p, repeated application of* $\rightsquigarrow$ *will lead to a term q such that* $q \rightsquigarrow r$ *only when* $q \equiv r$.

See [10] for the proof of this and other proof-theoretic results (the Church-Rosser property and type decidability).

1. $\mathsf{app}(\mathsf{abstract}\ X.\ a^{w\bullet A\to B}, b^{l\bullet A}) \rightsquigarrow a[b/X]^{w\bullet B}$
3. $\mathsf{specific}(\mathsf{use}\ x:S.\ a^{w\bullet\forall x:S.A}, v:S) \rightsquigarrow a[v/i]^{w\bullet A[v/x]}$
4. $\pi_1(\langle a,b\rangle^{w\bullet A\wedge B}) \rightsquigarrow a^{w\bullet A}$
5. $\pi_2(\langle a,b\rangle^{w\bullet A\wedge B}) \rightsquigarrow b^{w\bullet B}$
6. $\mathsf{case}\ \mathsf{Inl}(a)^{w\bullet A\vee B}\ \mathsf{of}\ \mathsf{Inl}(x^{w\bullet A}).b^{w\bullet C},\ \mathsf{Inr}(y^{w\bullet B}).c^{w\bullet C} \rightsquigarrow b[a/x]^{w\bullet C}$
7. $\mathsf{case}\ \mathsf{Inr}(a)^{w\bullet A\vee B}\ \mathsf{of}\ \mathsf{Inl}(x^{w\bullet A}).b^{w\bullet C},\ \mathsf{Inr}(y^{w\bullet B}).c^{w\bullet C} \rightsquigarrow c[a/y]^{w\bullet C}$
8. $\mathsf{select}\ (\mathsf{show}(v,a)^{w_1\bullet\exists y.P})\ \mathsf{in}\ x^{X\bullet P}.y.b^{w_2\bullet C} \rightsquigarrow b[a/x][v/y]^{w_2[w_1/X]\bullet C}$
9. $a[b/x]$ and $b \rightsquigarrow c$ entails $a[b/x] \rightsquigarrow a[c/x]$

Fig. 3. The 9 reduction rules inductively defining $\rightsquigarrow$.

$$\frac{}{\vdash \mathsf{Axiom}(n)^{l\bullet A}}\ Ax^{a} \qquad \frac{}{x^{l\bullet A}\vdash x^{l\bullet A}}\ Ass\!-\!I$$

$$\frac{\Delta, x^{w\bullet A}\vdash b^{w\bullet B}}{\Delta\vdash \mathsf{abstract}\ x.\ b^{w\bullet A\to B}}\ \to\!-I \qquad \frac{\Delta\vdash a^{w\bullet A}\quad \Delta'\vdash p^{w\bullet A\to B}}{\Delta,\Delta'\vdash \mathsf{app}(p,a)^{w\bullet B}}\ \to\!-E$$

$$\frac{\Delta\vdash p^{w\bullet A[y/x]}}{\Delta\vdash \mathsf{use}\ x:T.\ p^{w\bullet\forall x:T.A}}\ \forall\!-I \qquad \frac{\Delta\vdash p^{w\bullet\forall x:T.A}}{\Delta\vdash \mathsf{specific}(p,c)^{w\bullet A[c/x]}}\ \forall\!-E$$

$$\frac{\Delta\vdash p^{w\bullet P[a/y]}}{\Delta\vdash \mathsf{show}(a,p)^{w\bullet\exists y:T.P}}\ \exists\!-I \qquad \frac{\Delta_1\vdash p^{w\bullet\exists y:T.P}\quad \Delta_2, x^{w\bullet P[x/y]}\vdash q^{w\bullet Q}}{\Delta_1,\Delta_2\vdash \mathsf{select}\ (p)\ \mathsf{in}\ y.x.q^{w\bullet Q}}\ \exists\!-E$$

$$\frac{\Delta\vdash a^{w\bullet A}\quad \Delta'\vdash b^{w\bullet B}}{\Delta,\Delta'\vdash \langle a,b\rangle^{w\bullet A\wedge B}}\ \wedge\!-I \qquad \frac{\Delta\vdash p^{w\bullet A_1\wedge A_2}}{\Delta\vdash \pi_i(p)^{w\bullet A_i}}\ \wedge\!-E_i$$

$$\frac{\Delta\vdash a^{w\bullet\bot}}{\Delta\vdash \mathsf{abort}(a)^{w\bullet A}}\ \bot\!-E \qquad \frac{\Delta\vdash p^{w\bullet A_i}}{\Delta\vdash \#p^{w\bullet A_1\vee A_2}}\ \vee\!-I$$

$$\frac{\Delta\vdash p^{w\bullet A\vee B}\quad \Delta_1, x^{w\bullet A}\vdash a^{w\bullet C}\quad \Delta_2, y^{w\bullet B}\vdash b^{w\bullet C}}{\Delta_1,\Delta_2,\Delta\vdash \mathsf{case}\ p\ \mathsf{of}\ \mathsf{Inl}(x).a,\ \mathsf{Inr}(y).b^{w\bullet C}}\ \vee\!-E$$

[a] n is a constant, uniquely associated with the axiom $A\in\mathcal{AX}$

Fig. 4. Type inference rules of *LTT* corresponding to the logical rules of our calculus.

Program synthesis From a proof of a theorem, we can extract an imperative program which satisfies the theorem as a specification. Here, a program p is said to satisfy a theorem $l\bullet A$ when 1) the theorem's label designates the side-effect relation of the extracted imperative program (p mr $l\bullet A$), and 2) the program's return value is a return value modified realizer for the theorem's formula (p rv–mr $l\bullet A$). When this is true, we say that p is a modified realizer of $l\bullet A$, and write p mr $l\bullet A$.

We define an extraction map extract : *LTT* $\to$ *OCaml*(*StateRef*), satisfying the following extraction theorem, which tells us that the map produces a program satisfying the specification from a proof of the specification.

Theorem 2 (Extraction of modified realizers). *Given any* $\vdash p^{l\bullet A}\in LTT$ (any, *elements of* L_{Var} *do not occur in* l), $\mathsf{extract}(p^{l\bullet A})$ mr $l\bullet A$.

Proof. See [10] for details. The idea is to define two maps, ipart and ppart – the former defined over proof-terms to *OCaml* programs, the latter is from proof-terms to the terms used by formulae of our logic. The ppart has a similar definition to functional program extraction maps for intuitionistic logic of [1, 11]. For instance, we define $\mathsf{ppart}(\mathsf{show}(a, p^{w \bullet P[a/y]})^{w \bullet \exists y:T.P})$ to be $(a, \mathsf{ppart}(p))$ if P is not Harrop, and a otherwise. We can show that $\vdash p'^{l \bullet Sk(A)[\mathsf{ppart}(p)/f_A]}$ for some proof-term p'. So, interpreting l as a side-effect relation R, $(\sigma, \sigma') \in R$, $R \Vdash_\iota Sk(\mathsf{ppart}(p)/f_A)^{\sigma'}_{\sigma}$. We define the *OCaml* program $\mathsf{pview}(a) \equiv \texttt{fun}\ \bar{\mathtt{s}}_\mathtt{i} :: \bar{\mathtt{s}}_\mathtt{f}\ \texttt{->}\ \mathsf{ppart}(a)$, taking the set of state reference values of A, $\bar{s}_i$ and $\bar{s}_f$, as *OCaml* variables. Then, given a program $\bar{\mathtt{s}} := \bar{\mathtt{i}}; \mathtt{p}; \bar{\mathtt{s}} := \bar{\mathtt{f}}$ that has the same side-effects as R over $\bar{\mathtt{s}}$, evaluating $\mathsf{pview}(p)\mathtt{i\ f}$ will yield an intuitionistic modified realizer of $A^{\sigma'}_{\sigma}$, where $\bar{\mathtt{i}}$ and $\bar{\mathtt{f}}$ are state references of the same type and arity as $\bar{s}$, not used in A, l or $\mathtt{p}$.

The program $\mathsf{ipart}(p)$ looks like label l, but replaces any non-executable terms with equivalent executable programs, yielding a side-effect-equivalent program. Because proofs that introduce non-executable aspects $[\![A]\!]$ into a label always contain a proof of the formula $A \vee P$, from which an executable decision procedure for $[\![A]\!]$ can be extracted, using ppart and pview. For instance, if p is of the form $\mathsf{itecase}\ a^{l_1 \bullet (A \vee B)[\bar{s}_f/\bar{v}]}\ \mathsf{of}\ \mathsf{Inl}(t).b^{X; l_2 \bullet C},\ \mathsf{Inr}(u).c^{Y; l_3 \bullet C}$, then $\mathsf{ipart}(p)$ is

$$\begin{array}{l}\texttt{match}\ (\bar{\mathtt{i}} := \bar{\mathtt{s}}; \mathsf{ipart}(a); \bar{\mathtt{f}} := \bar{\mathtt{s}}; (\mathsf{pview}(a)\ \bar{\mathtt{i}}\ \bar{\mathtt{f}}))\ \texttt{with} \\ \quad \mathtt{Inl(x_t)} \texttt{->}\ \mathsf{ipart}(b)[\mathtt{()/X}] \mid \mathtt{Inr(x_u)} \texttt{->}\ \mathsf{ipart}(c)[\mathtt{()/Y}]\end{array}$$

where the state identifiers $\bar{\mathtt{i}}$, $\bar{\mathtt{f}}$ do not occur in $\mathsf{ipart}(a)$, $\mathsf{ipart}(b)$, $\mathsf{ipart}(c)$ and are of the same arity and typing as $\bar{s}$, the only state identifiers used in $A \vee B$.

Defining $\mathsf{extract}(p) \equiv \bar{\mathtt{i}} := \overline{\mathtt{si}}; \mathsf{ipart}(\mathtt{p}); \bar{\mathtt{f}} := \overline{\mathtt{sf}}; (\mathsf{pview}(\mathtt{p})\bar{\mathtt{i}}\ \bar{\mathtt{f}})$ will then satisfy the theorem. □

Implementation We have implemented our calculus by encoding *LTT* within *OCaml*. The proof-terms and labelled formula types are defined as data-types, the *LTT* typing relation is represented as a pairs of terms of the respective data-types, and the rules of the calculus treated as functions over such pairs. One of the advantages of our calculus is that it has a natural deduction presentation. This makes it easier to reason with than, say, the usual Hilbert-style presentations of Hoare-style Logics. Further, the Curry-Howard isomorphism can be exploited to enable intuitive user-defined development of proof tactics and parametrized lemmata, treated here as *OCaml* functions over proof-terms. In this way, the user can developing a proof in the way mathematicians naturally reason: using hypotheses, formulating conjectures, storing and retrieving lemmata, usually in a top-down, goal-directed fashion. The strong normalization property can be used to simplify proofs, which is valuable in the understanding and development of large proofs.

Example We demonstrate our work with an example: software to support an automated bank teller machine (ATM). We wish to derive a program that will process the ATM user's input of a bank card and a personal identification number (PIN). The user is permitted at most three attempts at entering their PIN. We require that either 1) the return value will be the account details, given that the user can enter a correct PIN for the bank card, or else 2) a message with a

$$\frac{\Delta \vdash p^{l_1 \bullet (A \vee B)[\bar{s}_f/\bar{v}]} \quad \Delta_1, x^{X;l_2 \bullet A} \vdash a^{X;l_2 \bullet C} \quad \Delta_2, y^{Y;l_3 \bullet B} \vdash b^{Y;l_3 \bullet C}}{\Delta_1, \Delta_2, \Delta \vdash \mathsf{itecase}\ p\ \mathsf{of}\ \mathsf{Inl}(x).a,\ \mathsf{Inr}(y).b^{l_1;\mathtt{if}\ [\![A]\!]\ \mathtt{then}\ l_2\ \mathtt{else\ if}\ [\![B]\!]\ \mathtt{then}\ l_3 \bullet C}}\ \mathit{if{-}then{-}else}$$

$$\frac{\Delta \vdash p^{\mathsf{any} \bullet A}}{\Delta \vdash <(p)^{l \bullet A}}\ \mathit{any} \qquad \frac{\Delta_1 \vdash p^{l_i \bullet A} \quad \Delta_2 \vdash q^{\mathsf{Frame}(l_{(i+1)\ \mathsf{mod}\ 2}, \bar{s})}}{\Delta_1, \Delta_2 \vdash \mathsf{seq}(p, q, i)^{l_l; l_2 \bullet A}}\ \mathit{seq}_i (i = 1, 2)$$

$$\frac{\begin{array}{c} \Gamma_2 \vdash c^{\mathsf{any} \bullet (P \vee \neg P)[\bar{s}_i/\bar{v}]} \quad \Gamma_3, x^{X \bullet Inv}, y^{X;w \bullet P} \vdash l^{X;w \bullet Inv} \\ \Gamma_1 \vdash i^{init \bullet Inv} \quad \Gamma_4, x^{X \bullet Inv}, y^{X;w \bullet \neg P} \vdash r^{X;w \bullet R} \end{array}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \vdash \mathsf{loop}(i, c, x.y.l, x.y.r)^{init;\mathtt{while}\ [\![P[\bar{s}_i/\bar{v}]]\!]\ \mathtt{do\ w;\ done} \bullet R}}\ \mathit{loop}$$

Fig. 5. Type inference rules of the LTT corresponding to structural rules.

correct error code will be returned, and the user's card will be retained by the ATM.

Domain assumptions In order to reason about programs constructed for the domain of this example, we need to define the sets $\mathcal{ADT}$, *StateRef*, *Prog* and $\mathcal{AX}$. We shall take $\mathcal{ADT}$ to contain several types, in addition to the standard types of *ML*: 1) `account`, a record type. An instance represents a bank account, where the `pin` field stores the PIN associated with the account, `accountOwner` field identifies the account name, and `balance` field stores the balance. 2) `card`, a record type. An instance represents the state of a given bank card in the ATM: `cardOwner` field identifies the account name associated with the card, and `retain` field is `true` if the card is to be retained by the ATM. 3) `notFound`, consisting of a single element, `Null`, used to denote a failed search. 4) `errorCode`, a type consisting of various error codes, such as `RETAIN_CARD` and `INCORRECT_PIN`. 5) `message`, an instance represents a message to be returned back to a user via the ATM's GUI, consisting of a string `detail` and an error code `error`.

Our set of state references *StateRef* is as follows: 1) `pin` : `ref int`, the PIN that has been input by the user. 2) `attempt` : `ref int`, the current number of attempts made by a user at inputting a PIN. 3) `userCard` : `ref card`, representing the state of the user's bank card in the ATM. 4) `accounts` : `ref account list`, a list of elements of `account`, representing all the accounts held by the bank.

The function $findAccount : int \rightarrow disjointUnion(account, notFound) \in \mathcal{ADT}$, given a requested PIN number p, will search through each element of `accounts`. If `accounts` contains an element a such that $a.pin = p$ and $a.accountOwner = card_f.cardOwner$, then `Inl(a)` is returned. If no match can be made, the term `Inr(Null)` is returned. The predicate $CorrectErrorCode : message \in \mathcal{ADT}$ is such that, if $\mathtt{q} \bullet CorrectErrorCode(x)$ holds, then, for any side-effect of a relation interpreting `q`, the information conveyed by the message's error code is correct in a sense prescribed by the domain, and represented by axioms $\mathcal{AX}$. The program `checkErrorCode` $\in Prog$ is defined so that, if its side-effects allow a message that carries correct error information, then it takes an appropriate action. In particular, if the side-effects of `checkErrorCode` permit the formation of a message that the card is to be retained, and this error code

is correct, then `checkErrorCode` will instruct the ATM to retain the card. We shall use the following axiom:

$$\mathsf{Axiom}(axCE)^{\mathsf{Frame}(\texttt{checkErrorCode},[pin;attempt;accounts])} \tag{2}$$

The pre-programmed `inputNumber` $\in Prog$ reads the PIN from the user of the ATM via a buffer.

Formal specification of requirements Our program's requirements can be specified by a formula, letting r denote a return value for a possible side-effect of the program:

$$\begin{array}{l}\forall x : account.r = Inl(x) \rightarrow findAccount(pin_f) \wedge \\ \forall y : message.r = Inr(y) \rightarrow CorrectErrorCode(y) \wedge card_f.retain = true\end{array} \tag{3}$$

Taking r for the Skolem function, then formula (3) is the Skolem form of $p \bullet Req$ where $Req \equiv$

$$\begin{array}{l}\exists a : account.findAccount(pin_f) = a \vee \\ \qquad \exists m : message.CorrectErrorCode(m) \ \wedge cardRetained_f = true\end{array} \tag{4}$$

By our adaptation of the proofs-as-programs paradigm, to synthesize a program that has return values `r` satisfying (3), we first proved (4) for some p, (but we omit the proof for reasons of space) and then applied extract over the corresponding proof-term and in conclusion obtained the proof-term:

$$\begin{array}{l}\mathsf{case}\ \mathsf{seq}(loop_5, \pi_1(\mathsf{Axiom}(axCE)), 1)\mathsf{of}\ \mathsf{Inl}(x). \\ \qquad\qquad \mathsf{Inr}(\mathsf{show}((answer, \mathsf{app}(lemCEC, x))), \mathsf{Inr}(y).lemFA\end{array}$$

with type q; `checkError` $\bullet R$, which, after application of extract, yields the required program of the form:

```
attempt := ref 0; while B do inputNumber; attempt :=!attempt + 1 done;
match C with Inl(x)-> Inr(answer)|Inr(y)-> Inl(findAccount(pin))
```

The specification (4) does not constrain the structure of the label `p`. For example, the specification does not define how many times (if any) the side-effect relation should allow the user to input the PIN before retaining the card. Through proving (4) with our calculus and the axioms $\mathcal{AX}$, we derive a form of `p` that makes such constraints upon the possible side-effects of the program to be extracted. In this way, our example demonstrates how the choice of axioms and given programs influences the structure of a derived label that satisfies a given specification.

6 Related work and conclusions

Various authors have given type-theoretic treatments to imperative program logics. It has been shown in [5] how a Hoare-style logic may be embedded within the Calculus of Construction through a monad-based interpretation of predicate transformer semantics, with an implementation in the Coq theorem prover [3]. Various forms of deductive program synthesis, with its roots in constructive logic and the Curry-Howard isomorphism, has been used successfully by [7], [8] and

[12]. The difference between our approach and those mentioned is that we do not use a meta-logical embedding of an imperative logic into a constructive type theory, but rather give a new logic that can be presented directly as a type theory.

Apart from the novelty of our approach, our results are useful because they present a unified means of synthesizing imperative program according to specifications of both side-effects and return values. Further, from the perspective of theorem prover implementation, the advantage of our calculus over others is the use of a natural deduction calculus for reasoning about imperative programs and the consequent adaption of the Curry-Howard isomorphism.

References

1. Ulrich Berger and Helmut Schwichtenberg. Program extraction from classical proofs. In D. Leivant, editor, *Logic and Computational Complexity, International Workshop LCC '94, Indiapolis, IN, USA, October 1994*, pages 77–97, 1995.
2. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
3. Thierry Coquand. Metamathematical Investigations of a Calculus of Constructions. In Piergiuorgio Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, 1990.
4. John Newsome Crossley, Iman Poernomo, and Martin Wirsing. Extraction of structured programs from specification proofs. In Didier Bert, Christine Choppy, and Peter Mosses, editors, *Workshop on Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 419–437, 1999.
5. J. C. Filliatre. Proof of imperative programs in type theory. In *International Workshop, TYPES '98, Kloster Irsee, Germany*, volume 1657 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
6. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 1969.
7. Zohar Manna and Richard J. Waldinger. The deductive synthesis of imperative LISP programs. In *National Conference on Artificial Intelligence*, pages 155–160, 1987.
8. Mihhail Matskin and Enn Tyugu. Strategies of Structural Synthesis of Programs. In *Proceedings 12th IEEE International Conference Automated Software Engineering*, pages 305–306. IEEE Computer Society, 1998.
9. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 1994.
10. Iman Poernomo. *The Curry-Howard isomorphism adapted for imperative program synthesis and reasoning*. PhD thesis, Monash University, Australia. In preparation.
11. Iman Poernomo and John Newsome Crossley. Protocols between programs and proofs. In Kung-Kiu Lau, editor, *Logic Based Program Synthesis and Transformation, 10th International Workshop, LOPSTR 2000 London, UK, July 24-28, 2000, Selected Papers*, volume 2042 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2001.
12. Jamie Stark and Andrew Ireland. Towards automatic imperative program synthesis through proof planning. In *Proceedings 13th IEEE International Conference Automated Software Engineering*, pages 44–51, 1999.