Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 155 (2006) 521–542
www.elsevier.com/locate/entcs

Practical Program Extraction from Classical Proofs

Yevgeniy Makarov[1]

Computer Science Department
Indiana University
Bloomington, IN 47405, USA

Abstract

It is well-known that a constructive proof of a Π^0_2 formula F written as a λ-term via Curry-Howard isomorphism, computes a function that witnesses F. Murthy [14] outlined an extension of this result to classical logic, with the double-negation rule mapped to Felleisen's control operator $\mathcal{C}$ [9]. Since $\mathcal{C}$ is similar to call/cc operator in Scheme and SML/NJ, this opens a possibility of extracting programs in these languages from classical proofs. However, even though the basic idea has appeared in the literature, there appears to be little work that uses Griffin's extension of the Curry-Howard isomorphism to extract practical programs in real programming languages.
In this article, we fill in missing steps in Murthy's argument and extend his method to encompass more interesting proofs by allowing additional ubiquitous inference rules: equality rules, rules for atomic formulas (such as transitivity of $\leq$), and rules for pairs of dual decidable atoms (such as $\leq$ and $>$). We illustrate the usefulness of this extension with a complete example of program extraction.

Keywords: Program extraction, classical logic, Curry-Howard isomorphism, control operators, lambda calculus, functional programming.

1 Introduction

Given a Π^0_2 specification $F \equiv \forall \boldsymbol{x} \exists y\, G[\boldsymbol{x}, y]$, and a proof $\mathcal{D}$ for it, there are several methods for extracting from $\mathcal{D}$ a program that witnesses F, i.e. computing a function f such that $\forall \boldsymbol{x}\, G[\boldsymbol{x}, f(\boldsymbol{x})]$ holds. These include Curry-Howard correspondence, as well as modified realizability [6] and Gödel's Dialectica interpretation [12]. These techniques were initially restricted to proofs in

[1] Email: emakarov@cs.indiana.edu

doi:10.1016/j.entcs.2005.11.071

constructive (intuitionistic) logic. Indeed, an actual program extraction experiment [7] reports that the limitation to constructive logic is a major hurdle in implementing program synthesis from proofs.

In 1990, Griffin [11] extended the Curry-Howard isomorphism to classical logic by observing that inference by contradiction corresponds to Felleisen's control operator $\mathcal{C}$. In his thesis [13], Murthy further explained why a term, in the λ-calculus extended with $\mathcal{C}$, extracted from a proof of F as above in the NuPRL type system, computes a function f that witnesses F. Murthy used the Gödel-Gentzen negative translation of classical logic into minimal logic, which corresponds to CPS translation, to obtain his result. In [14] he outlined an alternative approach, using reductions directly on classical derivations, without first translating them into minimal logic. However, although Griffin's $\mathcal{C}$-extension of Curry-Howard isomorphism is the simplest method of program extraction from classical proofs, there appears to be little work that uses it to extract practical programs in real programming languages. Quoting [7]: "although it is well-known that there are various techniques for getting computational content from classical proofs, these have not been seriously exploited."

In this paper we show that Griffin's extension suffices to straightforwardly convert classical derivations into demonstrably-correct programs. To apply the method to actual proofs of interest, we extend it to account for additional inference rules. We demonstrate the usefulness of these extensions by providing a complete example of program extraction.

The main idea of the paper is straightforward and can be traced to Friedman [10]. Suppose we are given a formula F which does not contain $\bot$ (falsehood), and a classical natural deduction derivation $\mathcal{D}$ of F. Replace in $\mathcal{D}$ all occurrences of $\bot$ by the derived formula F. The replacement might deform double-negation elimination (DNE) and *ex-falso-quodlibet* (EFQ) inferences, which after substitution would take the shapes

$$\frac{(D \to F) \to F}{D} \quad \text{and} \quad \frac{F}{D} \tag{1}$$

However, some of these deformed inferences can be repaired by the following reductions.

$$\begin{array}{c} \vdots \\ \dfrac{(D \to F) \to F}{\begin{array}{c} D \\ \vdots \\ F \end{array}} \end{array} \quad \to \quad \dfrac{\begin{array}{c} \vdots \\ (D \to F) \to F \end{array} \quad \dfrac{\begin{array}{c} D^u \\ \vdots \\ F \end{array}}{D \to F}\,(u)}{F} \tag{2}$$

$$\begin{array}{c} \vdots \\ \dfrac{F}{D} \\ \vdots \\ F \end{array} \quad \to \quad \begin{array}{c} \vdots \\ F \end{array} \tag{3}$$

These reductions are subject to the stipulation that, in either case, the displayed derivation of F from D does not close any assumptions on which the premise D depends. Note also that in order to preserve the conclusion of the derivation, i.e., to have subject reduction property, the formula F used to replace $\bot$ must also be the last formula of the derivation.

The reductions (2) and (3) correspond to the control operators $\mathcal{C}$ and $\mathcal{A}$ of [9], respectively. Note that they can be applied *anywhere* in the derivation, under the stipulation above.

Suppose F contains no $\forall$ or $\to$, and a derivation $\mathcal{D}$ of F reduces, under the usual detour reductions (see e.g. [17]) and the reductions above, to a closed normal derivation $\mathcal{D}'$. Then it can be shown that $\mathcal{D}'$ does not have instances of (1), and is thus a normal derivation in minimal logic. Thus if F is of the form $\exists x\, G$ where G is quantifier-free and does not contain $\to$, then one can extract from $\mathcal{D}'$ a witness for G using a theorem about the structure of normal derivations (cf. [17], Section 6.2).

We define a type system whose judgments $\Gamma \vdash_F t\colon D$ intend to convey "term t is extracted from a derivation of $\Gamma \vdash D$, given as a subderivation of a derivation of formula F." A central property on which the theorem about correctness of extracted programs relies is subject reduction: if $\Gamma \vdash_F t\colon F$ and t reduces to t' then $\Gamma \vdash_F t'\colon F$. This remains the case whether t reduces to t' via call-by-value, call-by-name, or indeed any strategy that satisfies some natural conditions (see Properties 3.1 and 3.2 in Section 3). This seems natural, since the concept of a classical derivation does not presuppose any evaluation strategy.

It is well-known that confluence of reductions is destroyed by control operators: a term may reduce to different values under different strategies. However, due to subject reduction, if $\vdash_F t\colon F$ holds, and t reduces to some value v, then $\vdash_F v\colon F$. This fact is sufficient for extraction of terms that witness Σ^0_1 formulas.

To illustrate our method, we derive in Section 5 a complete example, borrowed from [5]. The theorem F in question is this:

Proposition 1.1 *Suppose* $f\colon \mathsf{nat} \to \mathsf{nat}$ *is unbounded, i.e., there exists a function* g *such that* $f(g(y')) > y'$ *for all* y'. *If* $f(0) \le y$ *then* $f(x) \le y < f(x+1)$ *for some* x.

Proof. Towards contradiction, suppose no such x exists. We prove by induction on x that $f(x) \leq y$. The base case is given. For induction step, assume that $f(x) \leq y$. If $y < f(x+1)$ then we have a contradiction with our assumption, so $f(x+1) \leq y$. Putting $x = g(y)$ we get $f(g(y)) \leq y$, contradicting the assumption that $f(g(y)) > y$. □

Section 5 contains a formal proof the formula

$$\exists x\, f(x) \leq y \wedge y < f(x+1)$$

and a program with free variables f, g and y extracted from the proof.

The rest of the paper is structured as follows. In Section 2 we recall the Curry-Howard isomorphism, introduce natural deduction for minimal logic, and consider term reduction rules and subject reduction property. Section 3 describes classical logic and proves subject reduction for the DNE and EFQ rules. Section 4 considers additional practical inferences, such as equality rules, rules for atomic formulas (e.g., transitivity of $\leq$) and rules for pairs of dual decidable atoms (such as $\leq$ and $>$). Section 5 applies these results to our program extraction example. Finally, we discuss in Section 6 connections to related literature, and directions for future research.

2 Logical System

The material in this section is standard and serves as a reminder of Curry-Howard isomorphism in three areas: between formulas and types, between derivation and terms, and between reductions on derivation and reductions on terms.

Consider a first-order language $\mathcal{L}$ which includes simply typed λ-calculus. Atomic types may include, for example, nat, int, bool and $\mathsf{listnat}$. Terms of $\mathcal{L}$ occur in formulas and are called *object terms*, as opposed to *derivation terms* which encode derivations via Curry-Howard isomorphism. Object terms do not contain any control operators.

Because the main interest of this paper lies in derivation terms, we choose not to specify the exact syntax and typing rules of $\mathcal{L}$. The typing judgments are assumed to have the form $x_1 \colon \sigma_1, \ldots, x_n \colon \sigma_n \vdash_{\text{obj}} s \colon \sigma$ where x_i are variables, σ_i and σ are types, and s is an object term. Also, we do not list reductions on object terms but rather postulate that these reductions are computable, confluent, and strongly normalizing. This ensures that the congruence $\simeq$ generated by these reductions is decidable (by normalizing terms and comparing their normal forms). Besides β-reduction $(\lambda x.s)s' \succ_\beta s[s'/x]$, there may be so called δ-reductions, for example, $+(2,3) \succ_\delta 5$, which simulate

$$\Gamma, u\colon D \vdash u\colon D$$

$$\frac{\Gamma, u\colon C \vdash t\colon D}{\Gamma \vdash \lambda u.\, t\colon C \to D}\ (\to I) \qquad \frac{\Gamma \vdash t_1\colon C \to D \quad \Gamma \vdash t_2\colon C}{\Gamma \vdash t_1 t_2\colon D}\ (\to E)$$

$$\frac{\Gamma, x\colon \sigma \vdash t\colon D}{\Gamma \vdash \lambda x.\, t\colon \forall x^{\sigma}\, D}\ (\forall I) \qquad \frac{\Gamma \vdash t\colon \forall x^{\sigma}\, D \quad \Gamma \vdash_{\mathrm{obj}} s\colon \sigma}{\Gamma \vdash ts\colon D[s/x]}\ (\forall E)$$

$$\frac{\Gamma \vdash t_1\colon D_1 \quad \Gamma \vdash t_2\colon D_2}{\Gamma \vdash \langle t_1, t_2\rangle\colon D_1 \wedge D_2}\ (\wedge I) \qquad \frac{\Gamma \vdash t\colon D_1 \wedge D_2}{\Gamma \vdash \pi_i\, t\colon D_i}\ (\wedge E)$$

$$\frac{\Gamma \vdash t\colon D_i}{\Gamma \vdash \mathsf{in}_i\, t\colon D_1 \vee D_2}\ (\vee I)$$

$$\frac{\Gamma \vdash t\colon C_1 \vee C_2 \quad \Gamma, u_1\colon C_1 \vdash t_1\colon D \quad \Gamma, u_2\colon C_2 \vdash t_2\colon D}{\Gamma \vdash \mathsf{decide}(t; u_1.t_1; u_2.t_2)\colon D}\ (\vee E)$$

$$\frac{\Gamma \vdash t\colon D[s/x] \quad \Gamma \vdash_{\mathrm{obj}} s\colon \sigma}{\Gamma \vdash \langle s, t\rangle\colon \exists x^{\sigma}\, D}\ (\exists I)$$

$$\frac{\Gamma \vdash t_1\colon \exists x^{\sigma}\, C \quad \Gamma, x\colon \sigma, u\colon C \vdash t_2\colon D}{\Gamma \vdash \mathsf{spread}(t_1; x, u.\, t_2)\colon D}\ (\exists E)$$

Fig. 1. Inference rules of minimal logic. Terms in formulas, such as x and s, are *object* terms, and terms representing derivations, such as u and t, are *derivation* terms

the effect of built-in functions of a programming language. We identify object terms s' and s'' for which $s' \simeq s''$ holds. By dealing with these reductions on the metalevel we simplify formal proofs.

For the rest of the paper, $\equiv$ denotes syntactic identity, C, D, G range over arbitrary formulas of $\mathcal{L}$, s ranges over object terms and t ranges over arbitrary terms. The formula $\neg D$ is a contraction for $D \to \bot$. If M is a syntactic object then $\mathrm{FV}(M)$ denotes the set of free variables in M.

The derivation judgments of minimal logic have the form $\Gamma \vdash t\colon D$. Here Γ is a combination of type environment $x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n$ and formula environment $u_1\colon C_1, \ldots, u_k\colon C_k$ (the order of variables does not matter). If θ is a substitution then $\Gamma\theta$ is obtained from Γ by applying θ to each formula in Γ. We make a convention that derivation variables u_i and object variables x_i come from different namespaces. Thus if Γ is as above then $\Gamma[t/u_i] \equiv \Gamma$.

Inference rules of minimal logic in natural deduction style together with derivation terms are shown in Figure 1. (In the typing judgments $\Gamma \vdash_{\mathrm{obj}} s\colon \sigma$ the formula part of Γ must be discarded.) Each rule except for the first axiom serves either to introduce or to eliminate a certain connective. We write $\Gamma \vdash D$ if $\Gamma \vdash t\colon D$ is derivable for some t. (In the next sections we add more inference rules, and the definition of $\Gamma \vdash D$ is correspondingly expanded.)

The most general form of β-reductions on derivation term is the following.

$$\begin{array}{c} (\lambda x.t)s \succ_\beta t[s/x] \\ (\lambda u.t_1)t_2 \succ_\beta t_1[t_2/u] \\ \pi_i\langle t_1, t_2\rangle \succ_\beta t_i \end{array} \qquad \begin{array}{c} \mathsf{spread}(\langle t_1, t_2\rangle; x, u.t) \succ_\beta t[t_1/x, t_2/u] \\ \mathsf{decide}(\mathsf{in}_i\, t; u_1.t_1; u_2.t_2) \succ_\beta t_i[t/u_i] \end{array} \tag{4}$$

Each reduction on terms corresponds to a *detour* conversion on derivations, when an introduction rule is immediately followed by an elimination one (see [17], Section 6.1.4–7). The reductions for spread and decide correspond to

detour reductions for $\exists$ and $\vee$, respectively. [2]

Define $\rightarrow_\beta$ to be the least compatible (i.e., respecting term formation) relation containing $\succ_\beta$, and $\twoheadrightarrow_\beta$ to be the reflexive-transitive closure of $\rightarrow_\beta$. The following lemma is needed for proving subject reduction.

Lemma 2.1 (Substitution)

(i) *If* $\Gamma, u\colon C \vdash t_1\colon D$ *and* $\Gamma \vdash t_2\colon C$ *then* $\Gamma \vdash t_1[t_2/u]\colon D$.

(ii) *If* $\Gamma, x\colon \sigma \vdash t\colon D$ *and* $\Gamma \vdash_{\text{obj}} s\colon \sigma$ *then* $\Gamma[s/x] \vdash t[s/x]\colon D[s/x]$.

Proof. By induction on the first derivation. □

Theorem 2.2 (Subject reduction) *If* $\Gamma \vdash t\colon D$ *and* $t \rightarrow_\beta t'$ *then* $\Gamma \vdash t'\colon D$.

3 Intuitionistic and Classical Logics

Intuitionistic logic is obtained from minimal one by adding the following inference rule (where D is an arbitrary formula).

$$\frac{\Gamma \vdash t\colon \bot}{\Gamma \vdash \mathcal{A}t\colon D}$$

Classical logic is obtained from intuitionistic one by adding any of the following three rules

$$\frac{\Gamma \vdash t\colon (D \to \bot) \to \bot}{\Gamma \vdash \mathcal{F}t\colon D} \qquad \frac{\Gamma \vdash t\colon (D \to C) \to \bot}{\Gamma \vdash \mathcal{C}t\colon D} \qquad \frac{\Gamma \vdash t\colon (D \to C) \to D}{\Gamma \vdash \mathcal{K}t\colon D}$$

(where again C and D are arbitrary). We refer to $\mathcal{A}$, $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{K}$ as *control operators*. Felleisen et al. introduced operators $\mathcal{A}$ and $\mathcal{C}$ in [9] and operator $\mathcal{F}$ in [8]. $\mathcal{K}$ is the standard call-with-current-continuation (call/cc) construct from Scheme.

3.1 Operational Semantics

We would like to define operational semantics in such a way so that our results can be formulated for any evaluation strategy. So we define a strategy S as a pair consisting of a set of *evaluation contexts* and a *reduction rule*. The former determines exactly which subterm of a given term changes while the latter determines how it changes.

[2] The names spread and decide are taken from NuPRL theorem prover.

An evaluation context, or just a context, is a term with a hole. For example, in a call-by-value strategy, contexts are defined by the following grammar.

$$E ::= [] \mid Et \mid vE \mid \langle E, t\rangle \mid \langle v, E\rangle \mid \pi_i\, E \mid \mathsf{in}_i\, E \mid \\ \mathsf{spread}(E; x, u.t) \mid \mathsf{decide}(E; u_1.t; u_2.t)$$

where v ranges over the set of *values* defined as follows.

$$v ::= x \mid u \mid s \text{ (object term)} \mid \lambda x.\, t \mid \lambda u.\, t \mid \langle v, v\rangle \mid \mathsf{in}_i\, v$$

If E is a context then $E[t]$ denotes a term that is obtained by putting t in the hole of E. Note that in the definition above the hole is not in the scope of any abstraction because the definition lacks variants like $\lambda x.E$ or $\mathsf{spread}(t; x, u.E)$. This implies that $\mathrm{FV}(t) \subseteq \mathrm{FV}(E[t])$ for any context E and term t.

Reduction rule for a strategy S, denoted by $\succ_\beta^S$, is some restriction of the general reduction rule $\succ_\beta$. For example, call-by-value reduction rule is the following.

$$(\lambda x.t)s \succ_\beta^{\mathrm{CBV}} t[s/x] \qquad \mathsf{spread}(\langle v_1, v_2\rangle; x, u.t) \succ_\beta^{\mathrm{CBV}} t[v_1/x, v_2/u]$$
$$(\lambda u.t)v \succ_\beta^{\mathrm{CBV}} t[v/u] \qquad \mathsf{decide}(\mathsf{in}_i\, v; u_1.t_1; u_2.t_2) \succ_\beta^{\mathrm{CBV}} t_i[v/u_i]$$
$$\pi_i\langle v_1, v_2\rangle \succ_\beta^{\mathrm{CBV}} v_i$$

A term on the left-hand side of $\succ_\beta^S$ is called a *β-redex*. *Operational β-rule* for S acts on the entire term and applies β-reduction to the subterm in the hole of the context: $E[t] \mapsto_\beta^S E[t']$ if $t \succ_\beta^S t'$. *Operational control rule* is also applied to the entire term and has the following definition (recall that the definition of E depends on S).

$$E[\mathcal{A}t] \mapsto_c^S t \qquad E[\mathcal{F}t] \mapsto_c^S t(\lambda u.E[u])$$
$$E[\mathcal{C}t] \mapsto_c^S t(\lambda u.\mathcal{A}E[u]) \qquad E[\mathcal{K}\,t] \mapsto_c^S E[t(\lambda u.\mathcal{A}E[u])]$$

Some explanation of these rules is in order. Operator $\mathcal{A}$ just throws away the current context. Similarly, operator $\mathcal{C}$ discards the current context and applies its argument t to the current continuation $\lambda u.\,\mathcal{A}E[u]$. If t does not use this continuation then the net effect is the same as from the reduction for $\mathcal{A}$. In fact, $\mathcal{A}t$ can be defined as $\mathcal{C}(\lambda d.t)$ where $d \notin \mathrm{FV}(t)$. On the other hand, if the continuation is invoked inside t, the context existing at that point is replaced by E.

Similarly to $\mathcal{C}$, operator $\mathcal{F}$ discards the current context and applies its argument to the current continuation. But unlike the case of $\mathcal{C}$, this continuation

$\lambda u.E[u]$ does not have $\mathcal{A}$ in front of E, so when it is invoked, it is *composed* with the continuation existing at that point. Finally, $\mathcal{K}$ is like $\mathcal{C}$ in that it has $\mathcal{A}$ in the continuation (thus the continuation is not composable), but unlike $\mathcal{C}$ and $\mathcal{F}$, $\mathcal{K}$ does not discard the current context when it is invoked.

Given an evaluation strategy S, define $\mapsto_S = \mapsto_\beta^S \cup \mapsto_c^S$ and let $\mapsto\!\!\!\!\rightarrow_S$ be the reflexive-transitive closure of $\mapsto_S$. A term t is called *normal w.r.t. S* if there is no t' such that $t \mapsto_S t'$.

We require that regardless of the strategy the final result of the whole program is fully evaluated, except for the bodies of functions. More precisely, the final result of evaluation must be a *computed answer* defined as follows.

$$a ::= s \mid \lambda x.\, t \mid \lambda u.\, t \mid \langle a, a \rangle \mid \mathsf{in}_i\, a$$

One can think of computed answers as values returned to read-evaluate-print loop of an interpreter, and so this is a requirement on implementation rather than on the evaluation strategy of a particular language.

We postulate two restrictions on strategies. The first one says that the contexts and the reduction rule must be defined in such a way that evaluation cannot get "stuck."

Proposition 3.1 *If $\vdash t\colon D$ and t is normal w.r.t. S then t is a computed answer.*

This means that if t is not a computed answer then t can be represented as $E[t']$ where t' is a β-redex or $t' \equiv \mathcal{J}t''$ and $\mathcal{J}$ is a control operator. We do not require uniqueness here which makes a strategy deterministic because this is not essential for the rest of the paper.

The second restriction requires that a context's hole is not under abstraction.

Proposition 3.2 *If $\Gamma \vdash E[t]\colon D$ then $\Gamma \vdash t\colon C$ and $\Gamma, u\colon C \vdash E[u]\colon D$ for some formula C and for any fresh u.*

In other words, t does not have more free variables then $E[t]$ since all of them are declared in Γ. This is needed to ensure the restriction on reductions (2) and (3) given in the Introduction.

Properties 3.1 and 3.2 can in particular be proved for call-by-value and call-by-name strategies. For the rest of the paper, we assume that some strategy has been fixed, so we omit the index S in $\mapsto$.

3.2 Subject Reduction

Subject reduction property does not hold for the control rule. For example, suppose that $\Gamma \vdash E[\mathcal{A}t] \colon D$. By Property 3.2, we have $\Gamma \vdash \mathcal{A}t \colon C$ for some C which must have been derived from $\Gamma \vdash t \colon \bot$. Thus $E[\mathcal{A}t] \mapsto t$ but we don't have $\Gamma \vdash t \colon D$ unless $D \equiv \bot$.

As explained in the Introduction, the solution is to replace $\bot$ by the final formula F of the whole derivation (F is also called the *answer type* elsewhere in the literature). Let us introduce another judgment $\Gamma \vdash_F t \colon D$ where F is a closed formula not containing $\bot$. Inference rules are obtained from Figure 1 with the only change that $\vdash$ is replaced by $\vdash_F$, together with the following rules.

$$\frac{\Gamma \vdash_F t \colon F}{\Gamma \vdash_F \mathcal{A}t \colon D} \qquad \frac{\Gamma \vdash_F t \colon (D \to C) \to F}{\Gamma \vdash_F \mathcal{C}t \colon D}$$

$$\frac{\Gamma \vdash_F t \colon (D \to F) \to F}{\Gamma \vdash_F \mathcal{F}t \colon D} \qquad \frac{\Gamma \vdash_F t \colon (D \to C) \to D}{\Gamma \vdash_F \mathcal{K}t \colon D}$$

Substitution Lemma 2.1 can easily be extended to $\vdash_F$. Now we can prove Subject Reduction Theorem.

Theorem 3.3 (Subject reduction)

(i) *If* $\Gamma \vdash_F t \colon D$ *and* $t \mapsto_\beta t'$ *then* $\Gamma \vdash_F t' \colon D$. *In fact, the conclusion holds even if* $t \to_\beta t'$, *i.e., β-reduction is done anywhere in* t.

(ii) *If* $\Gamma \vdash_F t \colon F$ *and* $t \mapsto_c t'$ *then* $\Gamma \vdash_F t' \colon F$.

Proof. 1. This is proved in the same way as Theorem 2.2.

2. Suppose $\Gamma \vdash_F E[\mathcal{J}t] \colon F$ where $\mathcal{J}$ is $\mathcal{A}$, $\mathcal{F}$, $\mathcal{C}$, or $\mathcal{K}$. By Property 3.2, $\Gamma \vdash_F \mathcal{J}t \colon D$ and $\Gamma, u \colon D \vdash_F E[u] \colon F$ for some formula D.

Case $\mathcal{J} = \mathcal{A}$. Then $\Gamma \vdash_F \mathcal{A}t \colon D$ must have been derived from $\Gamma \vdash_F t \colon F$, and since $E[\mathcal{A}t] \mapsto_c t$, the claim holds.

Case $\mathcal{J} = \mathcal{F}$. Then $\Gamma \vdash_F \mathcal{F}t \colon D$ must have been derived from $\Gamma \vdash_F t \colon (D \to F) \to F$ and $E[\mathcal{F}t] \mapsto_c t(\lambda u.E[u])$. The following is the required derivation.

$$\frac{\Gamma \vdash_F t \colon (D \to F) \to F \qquad \dfrac{\Gamma, u \colon D \vdash_F E[u] \colon F}{\Gamma \vdash_F \lambda u.\, E[u] \colon D \to F}}{\Gamma \vdash_F t(\lambda u.\, E[u]) \colon F}$$

Case $\mathcal{J} = \mathcal{C}$. Then $\Gamma \vdash_F \mathcal{C}t \colon D$ must have been derived from $\Gamma \vdash_F t \colon (D \to$

$C) \to F$ and $E[\mathcal{C}t] \mapsto_c t(\lambda u.\mathcal{A}E[u])$. The following is the required derivation.

$$\dfrac{\Gamma \vdash_F t\colon (D \to C) \to F \quad \dfrac{\dfrac{\Gamma, u\colon D \vdash_F E[u]\colon F}{\Gamma, u\colon D \vdash_F \mathcal{A}E[u]\colon C}}{\Gamma \vdash_F \lambda u.\,\mathcal{A}E[u]\colon D \to C}}{\Gamma \vdash_F t(\lambda u.\,\mathcal{A}E[u])\colon F}$$

Case $\mathcal{J} = \mathcal{K}$. Then $\Gamma \vdash_F \mathcal{K}t\colon D$ must have been derived from $\Gamma \vdash_F t\colon (D \to C) \to D$ and $E[\mathcal{K}\,t] \mapsto_c E[t(\lambda u.\mathcal{A}E[u])]$. We have

$$\dfrac{\Gamma \vdash_F t\colon (D \to C) \to D \quad \dfrac{\dfrac{\Gamma, u\colon D \vdash_F E[u]\colon F}{\Gamma, u\colon D \vdash_F \mathcal{A}E[u]\colon C}}{\Gamma \vdash_F \lambda u.\,\mathcal{A}E[u]\colon D \to C}}{\Gamma \vdash_F t(\lambda u.\,\mathcal{A}E[u])\colon D}$$

and by Substitution Lemma $\Gamma \vdash_F E[t(\lambda u.\,\mathcal{A}E[u])]\colon F$. □

3.3 Pseudo-Classical Nature of Type System

The transition from regular classical provability $\vdash$ to $\vdash_F$ is straightforward.

Theorem 3.4 *If* $\Gamma \vdash t\colon D$ *then* $\Gamma[F/\bot] \vdash_F t\colon D[F/\bot]$.

Proof. Easy induction on derivations. □

The converse of this theorem holds if t does not contain control operators.

Theorem 3.5 *If* $\Gamma \vdash_F t\colon D$ *and* t *does not contain control operators, then* $\Gamma \vdash D$ *in intuitionistic logic.*

Proof. The inference rules for $\vdash$ and $\vdash_F$ coincide for other term constructors. □

Theorem 3.5 does not hold in general for terms with control operators. A representative example is the following. Assume that formulas C and D do not contain $\bot$ and let $F \equiv C \vee (C \to D)$. Consider the following term.

$$t \equiv \mathcal{F}\lambda k^{(C\vee(C\to D))\to F}.k\ \mathsf{in}_2\,\lambda u^C.\,\mathcal{A}(k\ \mathsf{in}_1\,u)$$

It is easy to see that $\vdash_F t\colon F$ (where $\mathcal{A}$ converts the type from F to D) and

$$t \mapsto_c (\lambda k.\,k\ \mathsf{in}_2(\lambda u.\,\mathcal{A}(k\ \mathsf{in}_1\,u)))\lambda w.\,w \twoheadrightarrow_\beta \mathsf{in}_2(\lambda u.\,\mathcal{A}(\mathsf{in}_1\,u)).$$

By Subject Reduction Theorem, $\vdash_F \mathsf{in}_2(\lambda u.\, \mathcal{A}(\mathsf{in}_1\, u))\colon F$, which must have been derived from $\vdash_F \lambda u.\, \mathcal{A}(\mathsf{in}_1\, u)\colon C \to D$. Clearly this does not mean that $C \to D$ is in general derivable in classical logic! The type derivation relies on the fact that $\mathcal{A}$ converts the type from F to D. Note also that $\lambda u.\, \mathcal{A}(\mathsf{in}_1\, u)$ is normal (the $\mathcal{A}$-reduction cannot be performed because otherwise the variable u would be unbound).

4 Towards More Interesting Derivations

In this section, we augment our logic by introducing some axioms and inference rules, together with their derivation terms, which make it possible to prove more interesting theorems. We describe new inference rules for $\vdash$ and also show rules for $\vdash_F$ in case they are different. Thus we simultaneously expand our logic and the type system for $\vdash_F$, preserving Theorems 3.4 and 3.5. For each new rule and each new derivation term we must adjust, if necessary, the definition of computed answer as well as of the context and of the reduction rule for the chosen strategy (we do it for call-by-value, left-to-right strategy as an example). The proofs of the statements in the previous sections, such as Substitution Lemma and Subject Reduction Theorem, must also be correspondingly expanded.

4.1 Atomic Axioms

$$\vdash \varepsilon\colon A \ (A \text{ is an atomic axiom}) \qquad \frac{\vdash A}{\Gamma \vdash_F \varepsilon\colon A}\ (A \text{ is any atom})$$

$$a ::= \ldots \mid \varepsilon \qquad v ::= \ldots \mid \varepsilon$$

The rules above indicate that if an atom A has a *closed* derivation (in particular, when A is an axiom), then we posit that the extracted term is just a fixed constant ε, since such derivation does not carry computational meaning. The presence of Γ in the derived judgment is intended to ensure weakening for $\vdash_F$.

For the rest of the paper, A and B will denote atomic formulas.

4.2 Atomic Rules

$$\frac{\Gamma \vdash t_1\colon A_1 \quad \ldots \quad \Gamma \vdash t_n\colon A_n}{\Gamma \vdash \mathsf{simplify}(t_1, \ldots, t_n)\colon B}\ n \geq 1 \qquad \frac{\Gamma \vdash t\colon A}{\Gamma \vdash t\colon B} \qquad \text{and similarly for } \vdash_F$$

$$\mathsf{simplify}(\boldsymbol{\varepsilon}) \succ_\beta \varepsilon$$

$$E ::= \ldots \mid \mathsf{simplify}(\boldsymbol{\varepsilon}, E, \boldsymbol{t})$$

Often one needs to consider rules which infer one atom from several others (transitivity of $\leq$ is an example). We may add some specific instances of the rules shown above to our system.

Operator simplify evaluates its arguments and returns ε. The order of evaluation is not important and may depend on the strategy. The idea is the following. According to Theorem 3.5, if $\Gamma \vdash_F t\colon D$ and t contains no control operators then D is derivable from the formulas in Γ. Thus to get a real witness of an existential formula we must expunge control operators from our program. Therefore, above we cannot just assign ε to B and ignore terms t_i corresponding to premises A_i of our inference rule. Instead, each t_i must be evaluated to ε (this can be done in a closed program by Property 3.1), and then we will have $\Gamma \vdash_F \varepsilon\colon A_i$ which must have been derived from $\vdash A_i$. Then $\vdash B$ will hold and we can derive $\Gamma \vdash_F \varepsilon\colon B$.

4.3 Equality Rules

$$\frac{\Gamma \vdash t_2\colon D[s] \quad \Gamma \vdash t_1\colon s = s'}{\Gamma \vdash \mathsf{second}(t_1, t_2)\colon D[s']} \qquad \frac{\Gamma \vdash t\colon D[s] \quad \vdash s = s'}{\Gamma \vdash t\colon D[s']} \text{ and similarly for } \vdash_F$$

$$\mathsf{second}(\varepsilon, v) \succ_\beta v$$

$$E ::= \ldots \mid \mathsf{second}(E, t) \mid \mathsf{second}(\varepsilon, E)$$

Operator second is similar to simplify in that it completely evaluates its arguments, but it returns the value of the second one.

In the presence of an axiom $x = x$ these rules are enough to prove that equality is an equivalence relation.

4.4 Dual Decidable Atoms

In this subsection we consider a subset of atoms which are decidable, such as $=$ and $\leq$ on natural numbers or null? on lists. We assume that for each such predicate symbol P there is another predicate symbol $\bar{P}$ which represents the complement of P. Moreover, we make a syntactic convention that $\bar{\bar{P}}$ is the same as P. Similarly, if $A \equiv P(\boldsymbol{s})$ then $\bar{A}$ denotes $\bar{P}(\boldsymbol{s})$.

We assume that for each closed decidable atom A, either $\vdash A$ or $\vdash \bar{A}$ holds.

The additions are the following.

$$\frac{\Gamma \vdash_{\text{obj}} \boldsymbol{s} : \boldsymbol{\sigma} \quad \Gamma, u_1 : P(\boldsymbol{s}) \vdash t_1 : D \quad \Gamma, u_2 : \bar{P}(\boldsymbol{s}) \vdash t_2 : D}{\Gamma \vdash \mathsf{if}\ P(\boldsymbol{s})\ \mathsf{then}\ t_1[\varepsilon/u_1]\ \mathsf{else}\ t_2[\varepsilon/u_2] : D} \text{ and similarly for } \vdash_F$$

$$\frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : \bar{A}}{\Gamma \vdash \mathsf{false}(t_1, t_2) : \bot} \qquad \frac{\Gamma \vdash_F t_1 : A \quad \Gamma \vdash_F t_2 : \bar{A}}{\Gamma \vdash_F \mathsf{false}(t_1, t_2) : F}$$

$$\begin{array}{ll} \mathsf{if}\ P(\boldsymbol{v})\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \succ_\beta t_1 & \text{if} \vdash P(\boldsymbol{v}) \\ \mathsf{if}\ P(\boldsymbol{v})\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \succ_\beta t_2 & \text{if} \vdash \bar{P}(\boldsymbol{v}) \end{array} \text{ for closed values } \boldsymbol{v}$$

$$E ::= \ldots \mid \mathsf{false}(E, t) \mid \mathsf{false}(v, E)$$

When a term $\mathsf{if}\ P(\boldsymbol{s})\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$ is evaluated, the object terms $\boldsymbol{s}$ are evaluated to values $\boldsymbol{v}$ and then the term reduces to either t_1 or t_2 depending on whether $P(\boldsymbol{v})$ or $\bar{P}(\boldsymbol{v})$ is true. However, we do not provide new context variants for if because evaluation of object terms is invisible on the metalevel.

Subject reduction for the new rules is straightforward. Suppose, for example, that $\Gamma \vdash_F \mathsf{if}\ P(\boldsymbol{s})\ \mathsf{then}\ t_1[\varepsilon/u_1]\ \mathsf{else}\ t_2[\varepsilon/u_2] : D$ and $\vdash P(\boldsymbol{s})$. Then $\Gamma, u_1 : P(\boldsymbol{s}) \vdash_F t_1 : D$ and $\Gamma \vdash_F \varepsilon : P(\boldsymbol{s})$, so by Substitution Lemma, $\Gamma \vdash_F t_1[\varepsilon/u_1] : D$.

The idea behind false is similar to that of $\mathsf{simplify}$: both arguments need to be completely evaluated. Since there are no reductions for false, Property 3.1 must be amended:

Proposition 3.1′ *If $\vdash t : D$ and t is normal w.r.t. the strategy in question then t is a computed answer or $t \equiv E[\mathsf{false}(\varepsilon, \varepsilon)]$.*

However, if our theory is consistent then the last variant is impossible. Indeed, if $\vdash E[\mathsf{false}(\varepsilon, \varepsilon)] : D$ then by Theorem 3.4, $\vdash_F E[\mathsf{false}(\varepsilon, \varepsilon)] : D[F/\bot]$. Then by Property 3.2, $\vdash_F \mathsf{false}(\varepsilon, \varepsilon) : F$. This judgment must have been derived from $\vdash_F \varepsilon : A$ and $\vdash_F \varepsilon : \bar{A}$ which in turn were derived from $\vdash A$ and $\vdash \bar{A}$, which contradicts consistency. Therefore, a control reduction must occur during evaluation of the arguments of false, which will abolish the current context.

4.5 Induction

The contents of this subsection is completely standard. We assume that nat is one of the base types of our logic. The additions are the following.

$$\frac{\Gamma \vdash t_1 \colon D[\mathsf{0}/x] \quad \Gamma \vdash t_2 \colon \forall x^{\mathsf{nat}}(D \to D[\mathsf{s}x/x]) \quad \Gamma \vdash_{\mathrm{obj}} t_3 \colon \mathsf{nat}}{\Gamma \vdash \mathsf{nrec}(t_1, t_2, t_3) \colon D[t_3/x]} \quad \text{and similarly for } \vdash_F$$

$$\mathsf{nrec}(v_1, v_2, \mathsf{0}) \succ_\beta v_1$$

$$\mathsf{nrec}(v_1, v_2, (\mathsf{s}v_3)) \succ_\beta v_2 v_3\, \mathsf{nrec}(v_1, v_2, v_3)$$

$$E ::= \ldots \mid \mathsf{nrec}(E, t, t) \mid \mathsf{nrec}(v, E, t) \mid \mathsf{nrec}(v, v, E)$$

The rules introduced so far are enough, for example, to formalize Peano Arithmetic, where one separation axioms is formulated as a rule $\dfrac{\mathsf{s}t = \mathsf{s}t'}{t = t'}$ and the other as an atomic axiom $\mathsf{s}t \neq \mathsf{0}$.

4.6 Main Result

As was described in Section 4.2, to produce a term which represents a valid derivation of a formula, we must eliminate control operators. In a computed answer, control operators may occur only under λ-abstraction.

Lemma 4.1 *If $\vdash_F a \colon D$ where a is a computed answer and D does not contain $\to$ or $\forall$, then a does not contain control operators.*

Proof. By induction on D. We use the fact that object terms do not contain control operators. □

Theorem 4.2 (Program extraction) *Suppose that our logic is consistent and suppose $\vdash t \colon \exists y\, G(y)$ where G does not contain $\bot$, $\to$ or $\forall$. Fix a strategy S and suppose that $t \mapsto\!\!\!\!\to_S t'$ where t' is normal w.r.t. S. Then t' is a computed answer, $t' \equiv \langle a_1, a_2 \rangle$ and $\vdash G(a_1)$.*

Proof. By Theorem 3.4, $\vdash_F t \colon \exists y\, G(y)$ where $F \equiv \exists y\, G(y)$. By Subject Reduction Theorem, $\vdash_F t' \colon \exists y\, G(y)$. Since t' is normal w.r.t. S, by Property 3.1′, either t' is a computed answer or $t' \equiv E[\mathsf{false}(\varepsilon, \varepsilon)]$. Since the logic is consistent, the last variant is impossible as explained in Section 4.4. From the definition of computed answers and the inference rules we see that t' must be of the form $\langle a_1, a_2 \rangle$ and the judgment $\vdash_F t' \colon \exists y\, G(y)$ must have been derived from $\vdash_F a_2 \colon G(a_1)$. By Lemma 4.1, a_2 does not contain control operators, therefore, by Theorem 3.5, $\vdash G(a_1)$. □

5 Example

We now consider in detail the Integer Root example introduced in Section 1. As is the case with every classically provable Π^0_2 formula, this theorem has also a constructive proof. The question of interest, however, is whether non-constructive inferences yield gain in size or complexity of the extraced program. Our present example is relevant because the conversion of the classical proof into an intuitionistic one is not completely trivial.

After we extract a term from a derivation it is straightforward to adapt it to Scheme or SML/NJ. As we do not yet have a software implementsation of our method, this example has been worked out by hand.

Though classical logic can be obtained from intuitionistic one by adding either double-negation elimination rule $\neg\neg D \to D$ or Pierce's law $((D \to C) \to D) \to D$, the former is a more commonly used rule of reasoning, therefore, we use it in our proof. One can see from inference rules in the beginning of Section 3 that operator $\mathcal{F}$ corresponds exactly to $\neg\neg D \to D$ while $\mathcal{C}$ corresponds to a more general schema $\neg(D \to C) \to D$ (due to the presence of $\mathcal{A}$ in the right-hand side of the reduction which transforms $\bot$ into an arbitrary type C; see the proof of Theorem 3.3). However, since operator $\mathcal{C}$ makes a program at least as efficient as $\mathcal{F}$ (because it produces continuations that throw away the current context) and since composable continuations are not available in standard Scheme, we express operator $\mathcal{C}$ through $\mathcal{K}$: $\mathcal{C}t = \mathcal{K}\lambda k.\mathcal{A}(tk)$. Because `abort` is a predefined procedure in Scheme, we used `return` as a synonym for $\mathcal{A}$, which we define by issuing the command

```
> (call/cc (lambda (k) (set! return k)))
```

at the top-level prompt.

The formal derivation is shown in Figure 2. For the sake of brevity, we exhibit derivations in "flag notation", as described for example in in [2, Section 3.1.5], which also visualizes the scoping of a natural deduction derivation. A formula in the "flag" (for example, $f(x) \leq y$) is an open assumption, and formulas to the right of the "flagpole" constitute a subderivation with that formula as an assumption. Once the assumption is closed, the corresponding indententation is withdrawn. We write the extracted terms in the right column. When these become large we use an ellipsis to refer to the (sub)term on the previous line.

$\neg\exists x\, f(x) \le y \wedge y < f(x+1)$	k
$f(x) \le y$	p
$f(x+1) > y$	u_1
$y < f(x+1)$	$u1$
$f(x) \le y \wedge y < f(x+1)$	$\langle p, u_1\rangle$
$\exists x f(x) \le y \wedge y < f(x+1)$	$\langle x, \langle p, u_1\rangle\rangle$
$\bot$	$k\langle x, \langle p, u_1\rangle\rangle$
$f(x+1) \le y$	$\mathcal{A}(k\langle x, \langle p, u_1\rangle\rangle)$
$f(x+1) \le y$	u_2
$f(x+1) \le y$	u_2
$f(x+1) \le y$	$\mathsf{if}\ f(x+1) > y\ \mathsf{then}\ \mathcal{A}(k\langle x, \langle p, \varepsilon\rangle\rangle)\ \mathsf{else}\ \varepsilon$
$f(x) \le y \to f(x+1) \le y$	$\lambda p.\mathsf{if}\ f(x+1) > y\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots$
$\forall x\, f(x) \le y \to f(x+1) \le y$	$\lambda x \lambda p.\mathsf{if}\ f(x+1) > y\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots$
$f(0) \le y$	ε
$f(g(y)) \le y$	$\mathsf{nrec}(\varepsilon, (\lambda x \lambda p.\,\mathsf{if}\ldots), g(y))$
$f(g(y)) > y$	ε
$\bot$	$\mathsf{false}(\mathsf{nrec}(\ldots), \varepsilon)$
$\neg\neg\exists x\, f(x) \le y \wedge y < f(x+1)$	$\lambda k.\,\mathsf{false}(\mathsf{nrec}(\ldots), \varepsilon)$
$\exists x\, f(x) \le y \wedge y < f(x+1)$	$\mathcal{C}\lambda k.\,\mathsf{false}(\mathsf{nrec}(\ldots), \varepsilon)$

Fig. 2. Derivation of Integer Root example

The complete extracted term is the following.

$$\mathcal{C}\lambda k.\,\mathsf{false}(\mathsf{nrec}(\varepsilon,\ \lambda x \lambda p.\,\mathsf{if}\ f(x+1) > y\ \mathsf{then}\ \mathcal{A}(k\langle x, \langle p, \varepsilon\rangle\rangle)\ \mathsf{else}\ \varepsilon,\ g(y)),\ \varepsilon)$$

The variables f, g and y are free in the derivation, as well as in the extracted term. By the Substitution Lemma, they can be instantiated by concrete terms. The derivation uses axioms $f(g(y)) > y$ and $f(0) \le y$. Therefore, the extracted term, with f, g and y instantiated, reduces to a correct result whenever the terms instantiating f, g and y satisfy these axioms.

The complete translation of the program above into Scheme is displayed in Figure 3. For false, we take advantage of Scheme's call-by-value strategy, which evaluates completely the arguments `args` (the order is not specified) before evaluating the body.

It is clear that the procedure `introot` in Figure 3 can be simplified. First, `false` can be replaced by its first argument since the second argument is already evaluated. Also, `return` is applied to the result of a continuation call. Since such calls abort the current context, `return` can be removed as well. Finally, since recursion is applied to a singleton type, the variable `p` (which gets bound to the value of the recursive call) always equals `'epsilon`. Therefore, we can replace `(k (cons x (cons p 'epsilon)))` by `(k x)` to make the procedure return a number n instead of a tuple `(`n `epsilon . epsilon)`. Note that we cannot eliminate the recursive call altogether, because it may have side effect via a continuation call.

```
(define false
  (lambda args
    (error 'false "Inconsistent theory")))

(define nrec
  (lambda (a f n)
    (if (zero? n) a
        (let ([m (sub1 n)])
          ((f m) (nrec a f m))))))

(define control
  (lambda (t)
    (call/cc (lambda (k) (return (t k))))))

(define introot
  (lambda (f g y)
    (control
      (lambda (k)
        (false (nrec 'epsilon
                  (lambda (x)
                    (lambda (p)
                      (if (> (f (+ x 1)) y)
                             (return (k (cons x
                                              (cons p 'epsilon))))
                             'epsilon)))
                  (g y))
               'epsilon)))))
```

Fig. 3. Translation of the program extracted from Integer Root example into Scheme

Based on these observations, we can informally rewrite `introot`, expanding `nrec` and removing unnecessary `epsilon`'s to obtain the code in Figure 4.

Thus, the algorithm starts with $x = 0$, where by assumption $f(x) \leq y$, and increments x until $f(x+1) > y$ for the first time. This x is immediately thrown to the top level as the final answer. This algorithm is akin to the

```
(define introot
  (lambda (f g y)
    (control
      (lambda (k)
        (letrec ([loop (lambda (x)
                         (if (zero? x)
                             'epsilon
                             (begin
                               (loop (sub1 x))
                               (if (> (f (+ x 1)) y)
                                   (k x)
                                   'epsilon))))])
          (loop (g y)))))))
```

Fig. 4. Another version of `introot`

following (naive) algorithm written in C.

```
int introot(int y) {
  int x;
  for (x = 0; x < g(y); x++)
    if (f(x+1) > y) return x;
}
```

The main difference is that Scheme program makes $g(y)$ nontail recursive calls.

The program extracted from a similar proof in [5] is the following.

$$\mathsf{nrec}(0, (\lambda x \lambda p.\, \mathsf{if}\ f(x) > y\ \mathsf{then}\ p\ \mathsf{else}\ x), g(y))$$

It corresponds to the following C code.

```
int introot(int y) {
  int x,p;
  for (x = 0,p = 0; x < g(y); x++)
    if ( !(f(x) > y) ) p = x;
  return p;
}
```

This program searches through every $0 \leq x < g(y)$ and returns the largest x such that $f(x) \leq y < f(x+1)$.

Our method scales well to larger proofs. For example, following [4] we extracted a program from a proof of a special case of Dickson's lemma, with the same ease as for the example above.

6 Comparisons and Future Work

Program extraction from classical proofs is a well-studied area, where interest has shifted from simple theories like Peano Arithmetic to more complicated ones like Classical Analysis with the Axiom of Countable Choice (see e.g. [3]). However, there were few attempts to date to develop program extraction methods that are readily implementable.

The most successful approach to program extraction was probably by Schwichtenberg [4], implemented in a Minlog system. That method works as follows. Suppose we are given a derivation in minimal logic of a formula

$$\forall \boldsymbol{x}.\boldsymbol{D} \to \exists^{\mathsf{cl}} y\, G, \tag{5}$$

where $\boldsymbol{D}$ and G are instances of syntactically defined *definite* and *goal* formulas, respectively, containing only connectives $\wedge$, $\to$, $\forall$, and $\bot$, and where $\exists^{\mathsf{cl}}$ is taken as an abbreviation for $\neg\forall\neg$. Such a derivation is converted into an intuitionistic derivation of

$$\forall \boldsymbol{x}.\boldsymbol{D} \to \exists y\, G,$$

where $\exists$ is the true existential quantifier. This is done by replacing $\bot$ by $\exists y\, G$, preserving the validity of the derivation but changing the premises to $\boldsymbol{D}[\exists y\, G/\bot]$ and the conclusion to $(\forall y.\, G[\exists y\, G/\bot] \to \exists y\, G) \to \exists y\, G$. One then proves that $\boldsymbol{D} \to D_i[\exists y\, G/\bot]$ and $G[\exists y\, G/\bot] \to \exists y\, G$ are derivable in intuitionistic logic. A purely functional program is extracted from the latter derivation using modified realizability.

A regular way to convert a classical proof to a minimal one is using Gödel-Gentzen's translation, which in this language amounts to inserting double negation in front of every atom. An advantage of Schwichtenberg's method is that to turn arbitrary formulas in this language to definite and goal ones, double negations need to be inserted in front of only certain atoms, considerably shortening the extracted program.

Another advantage of that method is that it allows the use of assumptions $\boldsymbol{D}$. However, we were able to extract programs from most examples considered by Schwichtenberg's group by using atomic axioms and inference rules.

The difficulty in using Schwichtenberg's method lies in its starting point. Aside from replacing $\vee$ and $\exists$ by their classical equivalents, one has to provide a derivation of (5) in *minimal* logic. Thus the only classical feature in this

derivation is the classical existential quantifier. While classical logic is theoretically embedded into this system, in practice one may encounter instances of DNE in the derivation. For example, the derived rule for elimination of $\exists^{\mathsf{cl}}$

$$\frac{\Gamma \vdash \exists^{\mathsf{cl}} x^{\sigma}\, C \quad \Gamma, x\colon \sigma, u\colon C \vdash D}{\Gamma \vdash D}$$

requires DNE for the conclusion D. In such cases one has to manually insert double negations to ensure that instances of DNE are constructively derivable.

In contrast, our approach handles arbitrary derivations of Π^0_2 formulas in full first-order classical logic. The extraction algorithm is very simple; indeed, the program extracted from a proof is almost the image of a derivation under the Curry-Howard isomorphism (with some closed proofs of atoms ignored). Also, having dual atoms makes proof more natural.

An important advantage of Schwichtenberg's account of modified realizability is that proof segments without computational meaning are excluded from the extracted program. This is done by ignoring subderivations of formulas that do not contain $\exists$ or $\vee$, in particular proofs of atoms. In our setting, it is not always possible to ignore subderivations of atoms. For example, in Figure 2, the significant segment, where the branching occurs, is a subderivation of the atomic formula $f(x+1) \leq y$.

An interesting method of program extraction was developed and implemented in a proof assistant PhoX by C. Raffalli [16], which uses second-order formulas to define datatypes. For example, if $N(x)$ is a second-order definition of the natural numbers, then a normal term extracted from a (constructive) proof of $N(x)$ is a Church numeral for x. Suppose that $I(x)$ and $O(x)$ are formulas for input and output datatypes, respectively. One considers a decidable specification $S(x, y)$:

$$\vdash_{\mathrm{Int}} D\colon \forall x\,(I(x) \to \forall y\,(O(y) \to S(x,y) \vee \neg S(x,y)))$$

and a classical proof of totality.

$$\vdash_{\mathrm{Cl}} P\colon \forall x\,(I(x) \to \exists y\,(O(y) \wedge S(x,y)))$$

Then from P and D a term in Parigot's $\lambda\mu$-calculus [15] is extracted which converts a λ-representation of the input into a representation of output that satisfies the specification. This method is similar to ours, in that extracted terms contain control operators, but the proof of correctness uses Krivine's realizability rather than subject reduction. Our method does not require a separate proof of the decidability of a specification, and it is insensitive to the representation of datatypes.

Yet another method, which is also based on evaluating classical derivations using control reductions, was developed by Barbanera and Berardi [1]. They describe a control operator $\mathcal{C}$ that corresponds to DNE and is akin to Felleisen's $\mathcal{C}$, but with reduction rules depending on the type of the term to which it is applied. The purpose is to ensure strong normalization of extracted terms. However, it is not clear from [1] how to translate programs with $\mathcal{C}$ into existing programming languages.

The results described here suggest several directions for future development. The development of software that implements our method is under way. There are also several extensions of interest for the method itself, notably bounded quantifiers and second-order logic. An important issue is the simplification of extracted programs along the lines describes in Section 5. There is evidence that it may significantly reduce the size of extracted programs, and make them more amenable to human inspection.

Acknowledgement

I am grateful to Daniel Leivant and Amr Sabry for discussions and guidance. I am also grateful to referees for helpful comments.

References

[1] Barbanera, F. and S. Berardi, *Extracting constructive content from classical logic via control-like reductions*, in: M. Bezem and J. Groote, editors, *Typed lambda calculi and applications (Utrecht, 1993)*, Lecture Notes in Computer Science **664** (1993), pp. 45–59.

[2] Barendregt, H., *Lambda calculi with types*, **II**, Oxford University Press, 1992 pp. 117–309.

[3] Berardi, S., M. Bezem and T. Coquand, *On the computational content of the axiom of choice*, The Journal of Symbolic Logic **63** (1998), pp. 600–622.

[4] Berger, U., W. Buchholz and H. Schwichtenberg, *Refined program extraction from classical proofs*, Annals of Pure and Applied Logic **114** (2002), pp. 3–25.

[5] Berger, U. and H. Schwichtenberg, *Program extraction from classical proofs*, in: D. Leivant, editor, *Logic and Computational Complexity, International Workshop LCC '94*, Lecture Notes in Computer Science **960** (1995), pp. 177–194.

[6] Berger, U., H. Schwichtenberg and M. Seisenberger, *The Warshall algorithm and Dickson's lemma: Two examples of realistic program extraction*, Journal of Automated Reasoning **26** (2001), pp. 205–221.

[7] Denney, E., *Logic-based program synthesis via program extraction*, **SS-02-05**, 2002, pp. 53–54.

[8] Felleisen, M., D. Friedman, B. Duba and J. Merrill, *Beyond continuations*, Technical Report 87-216, Indiana University Computer Science Department (1987).

[9] Felleisen, M., D. Friedman, E. Kohlbecker and B. Duba, *A syntactic theory of sequential control*, Theoretical Computer Science **52** (1987), pp. 205–237.

[10] Friedman, H., *Classically and intuitionistically provably recursive functions*, in: D. Scott and G.H.Müller, editors, *Higher set theory*, Lecture Notes in Mathematics **669**, Springer-Verlag, 1978 pp. 21–27.

[11] Griffin, T. G., *Formulae-as-types notion of control*, in: *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990, pp. 47–58.

[12] Kohlenbach, U., *Proof interpretations and the computational content of proofs* (2005), book draft.

[13] Murthy, C. R., "Extracting Constructive Content from Classical Proofs," Ph.D. thesis, Cornell University, Department of Computer Science (1990).

[14] Murthy, C. R., *Classical proofs as programs: How, what and why*, Technical Report TR 91-1215, Cornell University, Department of Computer Science (1991).

[15] Parigot, M., *$\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction*, in: *International Conference on Logic Programming and Automated Resoning*, Lecture Notes in Computer Science **624** (1992), pp. 190–201.

[16] Raffalli, C., *Getting results from programs extracted from classical proofs*, Theoretical Computer Science **323** (2004), pp. 49–70.

[17] Troelstra, A. S. and H. Schwichtenberg, "Basic Proof Theory," Cambridge University Press, 2000, 2nd edition.