Simplifying Programs Extracted from Classical Proofs

Yevgeniy Makarov

Indiana University, Bloomington, IN 47405, USA
emakarov@cs.indiana.edu

Abstract. Curry-Howard isomorphism extended to classical logic by associating the rule of double-negation elimination with the control operator $\mathcal{C}$ allows to view a natural deduction derivation of a Π^0_2 formula as a program. However, programs produced by this method are often hard to understand, in part because of the presence of control operators. In this paper, we show how to simplify such programs and make them more readable. Simplification consists of two parts: normalizing with respect to a certain set of reductions, and removing subterms of singleton type, whose origin is subderivations of atomic formulas. The latter part can in fact be applied to arbitrary functional program with side effects.

1 Introduction

Research about classical logic and computation received a powerful stimulus from the discovery made by Griffin and Murthy some 15 years ago, but the question of how best to extract demonstrably correct programs from classical proofs of Π^0_2 formulas still awaits a satisfactory answer. The facts that classical proofs often do not contain an obvious algorithm, and programs extracted from them are difficult to understand, does not make things easier. In this paper, we show how programs produced from classical proofs using Griffin-Murthy connection between control operators and double negation elimination (DNE) can be dramatically simplified so that they become amenable to human inspection.

Our contribution consists of two independent parts. The first describes rewriting rules for simplifying programs with control operators and free variables. Normalizing a program with respect to these rules eliminates most of the control operators and makes the program much clearer. The second part describes how to remove program fragments which have singleton type. These fragments inevitably occur in programs obtained from proofs and arise from subderivations of atomic formulas. In the case of programs obtained from constructive proofs, singleton subterms are useless because they always evaluate to the same constant, and their removal is achieved by an easy recursive mapping. However, in the presence of control operators the task is much trickier because even subterms of singleton type can produce side effects, which cannot be ignored.

We illustrate our method by two examples.[1]

[1] Thierry Coquand attributes the first example to Gabriel Stolzenberg in [1]. We learned it from Hugo Herbelin. The second example is taken from [2].

Example 1. For every function $f\colon \mathsf{nat} \to \mathsf{bool}$, there are two numbers x_1, x_2 such that $x_1 < x_2$ and $f(x_1) = f(x_2)$.

Proof. This claim follows from the following much more general statement.

$$\forall f\, \exists b\, \forall x\, \exists y.\, y \geq x \wedge f(y) = b \tag{1}$$

Having fixed b, one can find any number of points which are mapped to b by f.

To prove (1), we claim that $\forall x \exists y.\, y \geq x \wedge f(y) = \top$ or $\forall x \exists y.\, y \geq x \wedge f(y) = \bot$ (obviously, (1) follows from this disjunction). Suppose the contrary. Then we have $\exists x_1 \neg \exists y.\, y \geq x_0 \wedge f(y) = \top$ and $\exists x_2 \neg \exists y.\, y \geq x_1 \wedge f(y) = \bot$. Consider $y = \max(x_1, x_2)$. Clearly $y \geq x_1$, $y \geq x_2$ and either $f(y) = \top$ or $f(y) = \bot$. In both cases we have a contradiction with the previous claims.

Example 2 (Integer root). 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 $f(0) \leq y$ then there exists an "integer root" for y, i.e., there exists such x that $f(x) \leq y < f(x+1)$.

Proof. Toward 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 $f(x+1) \leq y$, which completes induction step. Putting $x = g(y)$ we get $f(g(y)) \leq y$, contradicting the assumption that $f(g(y)) > y$.

The plan of the paper is as follows. Section 2 presents the pseudo-classical type system which allows viewing derivations in classical logic as λ-terms with control operators. It also contains programs obtained from the proofs of the two examples shown above. Section 3 discusses reduction rules which simplify extracted programs. Section 4 is devoted to eliminating as many subterms of singleton type as possible. This is done by introducing a type-and-effect system and identifying subterms which produce side effects. After that, singleton terms without side effects can be erased. Finally, Section 5 describes related work and concludes.

2 Viewing classical derivations as programs

2.1 Type system

The details about viewing classical proofs of Π^0_2 formulas as programs using Griffin's connection between DNE and control operators are contained in our paper [3]. Here we briefly remind the main ideas.

It is well-known that Curry-Howard isomorphism establishes correspondence between three pairs of concepts: formulas and types, (natural deduction) derivation and terms, and reduction on derivations and reduction on terms. The last property is called *subject reduction*: reductions on terms turn derivations into derivations of the same formula.

Griffin extended Curry-Howard correspondence to classical logic by observing a connection between DNE and the control operators $\mathcal{C}$, a relative of call/cc

from the programming language Scheme. However, with this connection, subject reduction no longer holds. To restore it, one has to replace all occurrences of falsehood $\bot$ in the derivation by the formula F being proved by this derivation. This, of course, deforms DNE and *ex-falso-quodlibet* (EFQ) inferences, which after substitution would take the shapes

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

However, with the newly recovered subject reduction, all of these inferences can be expunged by normalizing the derivation, provided F does not contain $\to$ or $\forall$. In the end, we obtain a valid derivation in minimal logic, from which one can extract a witness for F.

Consider a first-order language $\mathcal{L}$ which includes simply typed λ-calculus. Base 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.

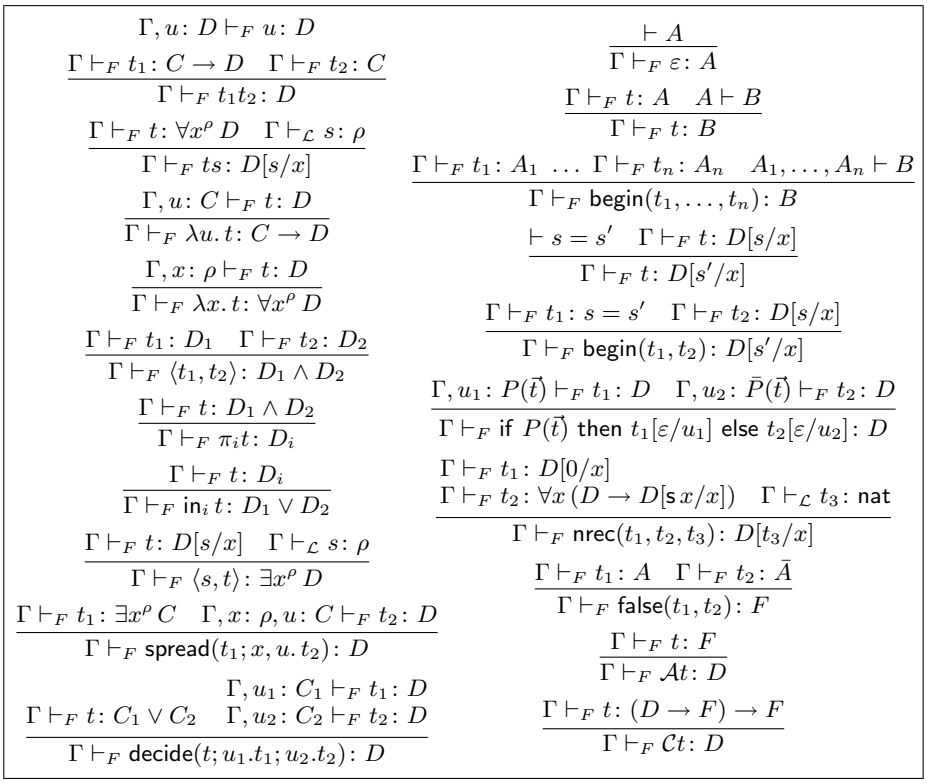
Because the main interest of this paper lies in derivation terms, we choose not to specify the exact syntax and typing rules of $\mathcal{L}$. The typing judgments are assumed to have the form $x_1 \colon \rho_1, \ldots, x_n \colon \rho_n \vdash_{\mathcal{L}} s \colon \rho$ where x_i are variables, ρ_i and ρ 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 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 range over arbitrary formulas of $\mathcal{L}$ while A, B range over atomic formulas (or *atoms*), 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.

We assume that there is a consistent classical theory T whose judgments are written as $\Gamma \vdash D$. It is also assumed that for each predicate symbol P there is another predicate symbol $\bar{P}$ which represents the complement of P, and if $A \equiv P(\vec{s})$ then $\bar{A}$ denotes $\bar{P}(\vec{s})$. Moreover, we make a syntactic convention that $\bar{\bar{P}}$ is the same as P. Finally, we assume that for each closed decidable atom A, either $\vdash A$ or $\vdash \bar{A}$ holds.

We now define a pseudo-classical 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." The type system is shown in Fig. 1.

The first column in Fig. 1 contains inference rules of minimal logic. The second column contains different extensions which are necessary for real-world proofs as well as classical rules with $\bot$ replaced by F. As an example of extension, if an atom A is derivable in our theory T, the corresponding term is



$$\Gamma, u\colon D \vdash_F u\colon D$$

$$\frac{\Gamma \vdash_F t_1\colon C \to D \quad \Gamma \vdash_F t_2\colon C}{\Gamma \vdash_F t_1 t_2\colon D}$$

$$\frac{\Gamma \vdash_F t\colon \forall x^\rho\, D \quad \Gamma \vdash_{\mathcal{L}} s\colon \rho}{\Gamma \vdash_F ts\colon D[s/x]}$$

$$\frac{\Gamma, u\colon C \vdash_F t\colon D}{\Gamma \vdash_F \lambda u.\, t\colon C \to D}$$

$$\frac{\Gamma, x\colon \rho \vdash_F t\colon D}{\Gamma \vdash_F \lambda x.\, t\colon \forall x^\rho\, D}$$

$$\frac{\Gamma \vdash_F t_1\colon D_1 \quad \Gamma \vdash_F t_2\colon D_2}{\Gamma \vdash_F \langle t_1, t_2\rangle\colon D_1 \wedge D_2}$$

$$\frac{\Gamma \vdash_F t\colon D_1 \wedge D_2}{\Gamma \vdash_F \pi_i t\colon D_i}$$

$$\frac{\Gamma \vdash_F t\colon D_i}{\Gamma \vdash_F \mathsf{in}_i\, t\colon D_1 \vee D_2}$$

$$\frac{\Gamma \vdash_F t\colon D[s/x] \quad \Gamma \vdash_{\mathcal{L}} s\colon \rho}{\Gamma \vdash_F \langle s, t\rangle\colon \exists x^\rho\, D}$$

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

$$\frac{\Gamma \vdash_F t\colon C_1 \vee C_2 \quad \begin{array}{c}\Gamma, u_1\colon C_1 \vdash_F t_1\colon D \\ \Gamma, u_2\colon C_2 \vdash_F t_2\colon D\end{array}}{\Gamma \vdash_F \mathsf{decide}(t; u_1.t_1; u_2.t_2)\colon D}$$

$$\frac{\vdash A}{\Gamma \vdash_F \varepsilon\colon A}$$

$$\frac{\Gamma \vdash_F t\colon A \quad A \vdash B}{\Gamma \vdash_F t\colon B}$$

$$\frac{\Gamma \vdash_F t_1\colon A_1 \;\ldots\; \Gamma \vdash_F t_n\colon A_n \quad A_1, \ldots, A_n \vdash B}{\Gamma \vdash_F \mathsf{begin}(t_1, \ldots, t_n)\colon B}$$

$$\frac{\vdash s = s' \quad \Gamma \vdash_F t\colon D[s/x]}{\Gamma \vdash_F t\colon D[s'/x]}$$

$$\frac{\Gamma \vdash_F t_1\colon s = s' \quad \Gamma \vdash_F t_2\colon D[s/x]}{\Gamma \vdash_F \mathsf{begin}(t_1, t_2)\colon D[s'/x]}$$

$$\frac{\Gamma, u_1\colon P(\vec{t}) \vdash_F t_1\colon D \quad \Gamma, u_2\colon \bar{P}(\vec{t}) \vdash_F t_2\colon D}{\Gamma \vdash_F \mathsf{if}\ P(\vec{t})\ \mathsf{then}\ t_1[\varepsilon/u_1]\ \mathsf{else}\ t_2[\varepsilon/u_2]\colon D}$$

$$\frac{\Gamma \vdash_F t_1\colon D[0/x] \quad \Gamma \vdash_F t_2\colon \forall x\,(D \to D[\mathsf{s}\,x/x]) \quad \Gamma \vdash_{\mathcal{L}} t_3\colon \mathsf{nat}}{\Gamma \vdash_F \mathsf{nrec}(t_1, t_2, t_3)\colon D[t_3/x]}$$

$$\frac{\Gamma \vdash_F t_1\colon A \quad \Gamma \vdash_F t_2\colon \bar{A}}{\Gamma \vdash_F \mathsf{false}(t_1, t_2)\colon F}$$

$$\frac{\Gamma \vdash_F t\colon F}{\Gamma \vdash_F \mathcal{A}t\colon D}$$

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

Fig. 1. Pseudo-classical type system

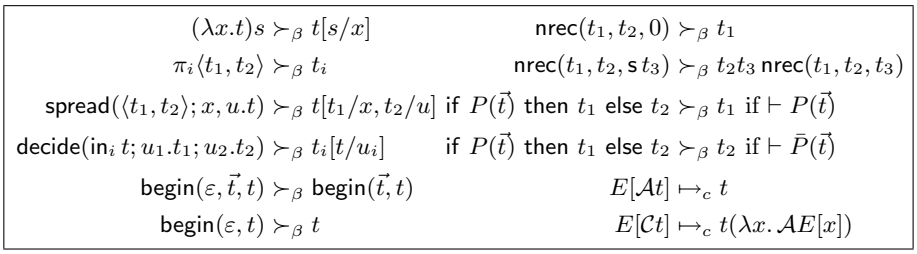
ε (regardless of whether A is an axiom or has a nontrivial derivation). If several atoms $A_1, \ldots, A_n$ with proof terms $t_1, \ldots, t_n$ imply another atom B, the proof term for B is $\mathsf{begin}(t_1, \ldots, t_n)$. As one can see from reductions shown in Fig. 2, $\mathsf{begin}(t_1, \ldots, t_n)$ just evaluates all its arguments and returns the last one of them. Another addition is a term $\mathsf{false}(t_1, t_2)$ which proves F from two complementary atoms A and $\bar{A}$ (in T, before we replaced $\bot$ by F, A and $\bar{A}$ implied $\bot$). There are no reduction rules involving false, but this term cannot come up during evaluation if T is consistent. For more details, see [3].

In Fig. 2, E denotes an evaluation context defined by the following grammar:

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

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

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



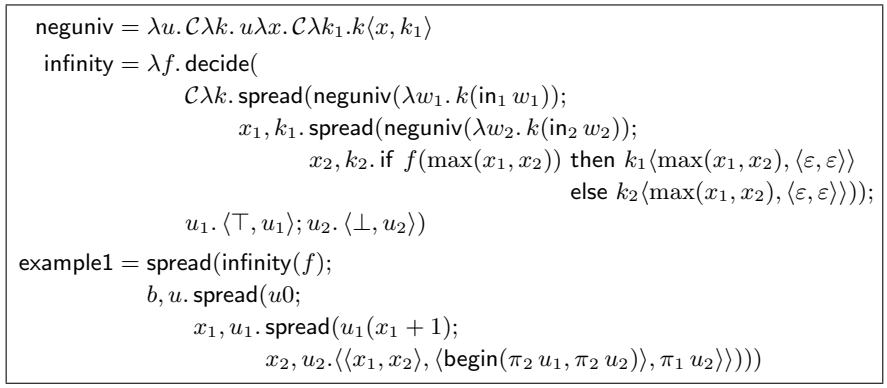
$$
\begin{array}{rl}
(\lambda x.t)s \succ_\beta t[s/x] & \mathsf{nrec}(t_1,t_2,0) \succ_\beta t_1 \\
\pi_i\langle t_1,t_2\rangle \succ_\beta t_i & \mathsf{nrec}(t_1,t_2,\mathsf{s}\,t_3) \succ_\beta t_2 t_3\,\mathsf{nrec}(t_1,t_2,t_3) \\
\mathsf{spread}(\langle t_1,t_2\rangle;x,u.t) \succ_\beta t[t_1/x,t_2/u] & \text{if } P(\vec{t}) \text{ then } t_1 \text{ else } t_2 \succ_\beta t_1 \text{ if} \vdash P(\vec{t}) \\
\mathsf{decide}(\mathsf{in}_i\,t;u_1.t_1;u_2.t_2) \succ_\beta t_i[t/u_i] & \text{if } P(\vec{t}) \text{ then } t_1 \text{ else } t_2 \succ_\beta t_2 \text{ if} \vdash \bar{P}(\vec{t}) \\
\mathsf{begin}(\varepsilon,\vec{t},t) \succ_\beta \mathsf{begin}(\vec{t},t) & E[\mathcal{A}t] \mapsto_c t \\
\mathsf{begin}(\varepsilon,t) \succ_\beta t & E[\mathcal{C}t] \mapsto_c t(\lambda x.\,\mathcal{A}E[x])
\end{array}
$$

Fig. 2. Reductions

Concerning reductions, the following syntactic conventions are adopted. Local reductions (i.e., the ones which can be performed anywhere in the term) are denoted by $\succ$ with possible indexes. Their compatible closures are denoted by $\to$. The reflexive-transitive closure of $\to$ is written as $\twoheadrightarrow$. Next, $\mapsto$ denotes *computational rule*, which can be applied only to the term as a whole. Several indexes indicate union of reductions. If $\to_{\vec{\imath}}$ is a reduction with some indexes $\vec{\imath}$ then $=_{\vec{\imath}}$ is the corresponding convertibility relation.

2.2 Programs produced from examples

The program produced from proof of Theorem 1 via typing rules in Fig. 1 is shown in Fig. 3. It consists of three parts: neguniv is a proof of $\neg(\forall x.\,D) \to \exists x.\,\neg D$, infinity is a proof of (1), and example1 combines the parts into the final proof. The program obtained from the proof of Example 2 is shown in Fig. 4.



$$
\begin{array}{l}
\mathsf{neguniv} = \lambda u.\,\mathcal{C}\lambda k.\,u\lambda x.\,\mathcal{C}\lambda k_1.k\langle x,k_1\rangle \\
\mathsf{infinity} = \lambda f.\,\mathsf{decide}(\\
\qquad \mathcal{C}\lambda k.\,\mathsf{spread}(\mathsf{neguniv}(\lambda w_1.\,k(\mathsf{in}_1\,w_1)); \\
\qquad\quad x_1,k_1.\,\mathsf{spread}(\mathsf{neguniv}(\lambda w_2.\,k(\mathsf{in}_2\,w_2)); \\
\qquad\qquad x_2,k_2.\,\mathsf{if}\ f(\max(x_1,x_2))\ \mathsf{then}\ k_1\langle\max(x_1,x_2),\langle\varepsilon,\varepsilon\rangle\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ k_2\langle\max(x_1,x_2),\langle\varepsilon,\varepsilon\rangle\rangle)); \\
\qquad u_1.\,\langle\top,u_1\rangle;u_2.\,\langle\bot,u_2\rangle) \\
\mathsf{example1} = \mathsf{spread}(\mathsf{infinity}(f); \\
\qquad b,u.\,\mathsf{spread}(u0; \\
\qquad\quad x_1,u_1.\,\mathsf{spread}(u_1(x_1+1); \\
\qquad\qquad x_2,u_2.\langle\langle x_1,x_2\rangle,\langle\mathsf{begin}(\pi_2\,u_1,\pi_2\,u_2)\rangle,\pi_1\,u_2\rangle\rangle))))
\end{array}
$$

Fig. 3. Program extracted from the proof of Example 1

```
example2 = Cλk. false(nrec(ε,
                           λxλz. if f(x + 1) > y then A(k⟨x, ⟨z, ε⟩⟩) else ε,
                           g(y)),
                      ε)
```

Fig. 4. Program extracted from the proof of Example 2

It is possible to discern the structure of the proof in the program in Fig. 3. For example, in infinity, the last step uses the disjunction $(\forall x \exists y.\, y \geq x \wedge f(y) = \top) \vee (\forall x \exists y.\, y \geq x \wedge f(y) = \bot)$ (decide), which is proved by contradiction ($\mathcal{C}\lambda k \ldots$). However, it is hard to understand the algorithm behind this program, in particular because the term with expanded definitions contains at least three nested control operators.

The program in Fig. 4 is much clearer. One can see that it is a loop which starts at $x = 0$ (recall that by assumption $f(0) \leq y$) and goes until $x = g(y) - 1$. The first time when $f(x + 1) > y$, x turns out to be the answer, so the loop is terminated and this x is returned. However, this program contains some unnecessary fragments, like $\mathsf{false}(\cdot, \varepsilon)$. Also, both programs have many occurrences of ε which came from subderivations of atomic formulas. As explained in the Introduction, subterms of singleton type without side effects are computationally useless and should be removed.

3 Simplifying extracted programs

Programs in Figs. 3 and 4 have free variables, namely f, g and y; therefore, they cannot be normalized to a value. Simplifying programs with free variables seems to require reductions which depend on the class of the programs and the final form we are willing to accept. One set of such rules is suggested in Fig. 5. Before applying these rules, operator $\mathcal{A}$ is converted to $\mathcal{C}$ according to the definition $\mathcal{A}t \equiv \mathcal{C}\lambda k.\, t$ where $k \notin \mathrm{FV}(t)$.

The rule (simp2) is a local rule for $\mathcal{C}$ which, in contrast to $\mapsto_c$, can be applied anywhere in the term. Its intention is to lift control operators as much as possible and then contract multiple occurrences of $\mathcal{C}$ using rule (simp3). The idea of the latter rule is that after reducing the first operator $\mathcal{C}$, the term $\mathcal{C}\lambda k_2.t$ is going to be on the top level, therefore, the continuation stored into k_2 is going to be $\lambda x.\, \mathcal{A}x$. However, (simp3) lacks $\mathcal{A}$ in front of x. In fact, if one adds operators $\mathcal{A}$ in front of k' in (simp2) and x in (simp3), one would get the rules from Felleisen and Hieb's calculus $\lambda_{\mathcal{C}}$ (see [4] and [5]). We will return to this momentarily.

The rule (simp4) is also standard and is found, for example, in $\lambda_{\mathcal{C}^-}$ calculus in [5]. It is used when t represents a derivation made in constructive logic and ended by a superfluous DNE.

$$\frac{\dfrac{k\colon D \to \bot \quad t\colon D}{kt\colon \bot}}{\mathcal{C}\lambda k^{D\to\bot}.\, kt\colon D}$$

$$
\begin{array}{rl}
\text{Arbitrary } \beta\text{-reductions} & \text{(simp1)} \\
E[\mathcal{C}\lambda k.t] \succ_{\text{simp}} \mathcal{C}\lambda k'.\, t[\lambda x.\, k' E[x]/k] & \text{(simp2)} \\
\mathcal{C}\lambda k_1.\, \mathcal{C}\lambda k_2.\, t \succ_{\text{simp}} \mathcal{C}\lambda k_1.\, t[\lambda x.\, x/k_2] & \text{(simp3)} \\
\mathcal{C}\lambda k.\, kt \succ_{\text{simp}} t \qquad k \notin \text{FV}(t) & \text{(simp4)} \\
\mathcal{C}\lambda k.\, t \mapsto_{\text{simp}} t \qquad k \notin \text{FV}(t) & \text{(simp5)} \\
\mathsf{if}\ A\ \mathsf{then}\ \mathcal{C}\lambda k_1.\, t_1\ \mathsf{else}\ \mathcal{C}\lambda k_2.\, t_2 \succ_{\text{simp}} \mathcal{C}\lambda k.\, \mathsf{if}\ A\ \mathsf{then}\ t_1[k/k_1]\ \mathsf{else}\ t_2[k/k_2] & \text{(simp6)} \\
\mathsf{if}\ A\ \mathsf{then}\ \mathcal{C}\lambda k_1.\, t_1\ \mathsf{else}\ t_2 \succ_{\text{simp}} \mathcal{C}\lambda k.\, \mathsf{if}\ A\ \mathsf{then}\ t_1[k/k_1]\ \mathsf{else}\ kt_2 & \text{(simp7)} \\
\mathsf{if}\ A\ \mathsf{then}\ t_1\ \mathsf{else}\ \mathcal{C}\lambda k_2.\, t_2 \succ_{\text{simp}} \mathcal{C}\lambda k.\, \mathsf{if}\ A\ \mathsf{then}\ kt_1\ \mathsf{else}\ t_2[k/k_2] & \text{(simp8)} \\
kt^F \succ_{\text{simp}} t & \text{(simp9)}
\end{array}
$$

Fig. 5. Reductions for simplifying extracted programs

The rule (simp5) is only applied on the top level and corresponds to removing the top-level $\mathcal{A}$.

The rules (simp6–8) are introduced for pragmatic reasons. Without them, many control operators are stuck under if and never get eliminated by (simp3). However, sometimes it is better not to lift control operator over if. For example, if lifting is applied to the program from Example 2 in Fig. 4, then the if statement is converted into

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

(and $\mathcal{C}$ cannot be lifted further because of λ-abstraction), which is arguable less clear.

The rule (simp9) means that an application of a continuation variable to a term t which proves the final formula F can be replaced by t (this requires having a separate syntactic class for continuation variables). This rule is special because all other rules (when $\mathcal{A}$ is inserted in the right-hand sides of (simp2) and (simp3), as described above) preserve semantics and are validated by CPS translation, in the sense that the CPS translations of the left- and right-hand sides are convertible into each other. The rule (simp9) does not necessarily preserve semantics; however, it satisfies subject reduction w.r.t. type system in Fig. 1 because in this case k has type $F \to F$ (the range of continuations is always F) and can therefore be replaced by the identity function. Therefore, even if after applying (simp9) a program may return a different answer, this answer will still be a witness for F.

The reason for introducing (simp9) is the following. During simplification one often encounters terms of the following form.

$$\mathcal{C}\lambda k.\, \ldots kt_1 \ldots kt_2 \ldots$$

Moreover, it turns out that at least some of the subterms kt_i are going to be executed on the top level (i.e., in the empty context). The presence of the continuations k prevents removing the top-level $\mathcal{C}$. It is clear that k is going to be

bound to $\lambda x.\, \mathcal{A}x$ and that invoking k on the top level is useless. Therefore, it would be possible to remove k from such subterms. However, it seems difficult to ascertain which parts of the program are going to be executed in the empty context. For example, one can see that in Fig. 4, the if construct is *not* going to be executed on the top level. However, in the case of the normalized CPS transformation of that program shown in Fig. 6, if statement *is* going to be executed on the top level. (The CPS translation we used is described in [6].)

$$\begin{array}{l}\mathsf{nrec}(\lambda k.\, k,\\ \quad \lambda x \lambda f.\, \lambda k.\, f \lambda z.\, \mathsf{if}\ f(x+1) > y\ \mathsf{then}\ \langle x, \langle z, \varepsilon\rangle\rangle\ \mathsf{else}\ k\varepsilon,\\ \quad g(y))(\lambda z.\, z)\end{array}$$

Fig. 6. The result of CPS transformation of the program from Example 2

Therefore, instead of trying to figure out if a particular subterm is going to be executed in the empty context, one can note that the subterms for which this is true must necessarily have type F. Thus by replacing kt with t when t has type F, we will remove the application of the continuation from all subterms of the form kt which were supposed to be executed on the top level (and possibly some others, which is no harm due to preservation of subject reduction).

Returning to the rule (simp3), one observes that, similar to the case of (simp9), it preserves subject reduction even if it does not preserve the semantics. However, replacing $\lambda x.\, x$ by $\lambda x.\, \mathcal{A}x$ in the right-hand side leads to nontermination in the following example.

$$\begin{array}{lll}\mathcal{C}\lambda k_1.\, \mathcal{C}\lambda k_2.\, \mathsf{if}\ A\ \mathsf{then}\ k_2 0\ \mathsf{else}\ \varepsilon & & \\ \quad \to \mathcal{C}\lambda k_1.\, \mathsf{if}\ A\ \mathsf{then}\ (\lambda x.\, \mathcal{C}\lambda k'.\, x)0\ \mathsf{else}\ \varepsilon & & (\text{simp3})\\ \quad \to \mathcal{C}\lambda k_1.\, \mathsf{if}\ A\ \mathsf{then}\ \mathcal{C}\lambda k'.\, 0\ \mathsf{else}\ \varepsilon & & (\beta)\\ \quad \to \mathcal{C}\lambda k_1.\, \mathcal{C}\lambda k_2.\, \mathsf{if}\ A\ \mathsf{then}\ 0\ \mathsf{else}\ k_2\varepsilon & & (\text{simp7})\\ \quad \twoheadrightarrow \mathcal{C}\lambda k_1.\, \mathcal{C}\lambda k_2.\, \mathsf{if}\ A\ \mathsf{then}\ k_2 0\ \mathsf{else}\ \varepsilon & & (\text{simp3,8})(\beta)\end{array}$$

As for adding $\mathcal{A}$ to (simp2), [5] points out that it is not necessary for evaluation but is important for the correspondence between the operational and reduction semantics. If $k'E[x]$ occurs inside some context, it should be possible to erase this context because k' is an abortive continuation. In our examples, though, we have not encountered this situation. In any case, it is easy to add a rule

$$E[kt] \succ_{\text{simp}} kt$$

which will take care of this.

We created a program for simplifying extracted terms, which used the first applicable rule (simp1–9) during the recursive traversal of the term. At this point, we don't know if the simplifying rules are strongly normalizing on well-typed programs.

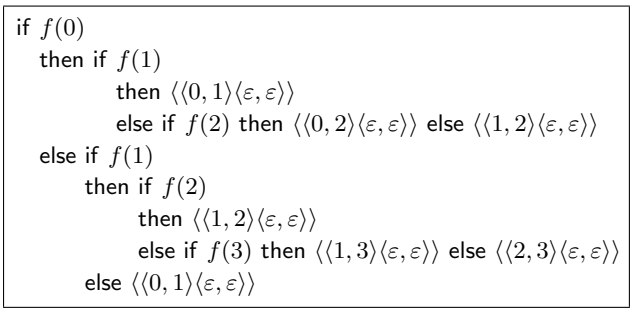


Fig. 7. Simplified program from Example 1

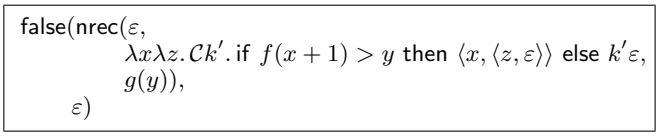


Fig. 8. Simplified program from Example 2

After performing simplifying reductions 153 times, the program from Example 1 in Fig. 3 is turned into the one in Fig. 7. The program from Example 2 is normalized after only three reductions: the first removes the application of the continuation k according to (simp9), the second lifts $\mathcal{A}$ over if, and the third removes the top-level control operator which functions as $\mathcal{A}$. The result is shown in Fig. 8.

It is clear that false can also be removed, but it functions as a type cast since recursion is done on singleton type I while the type of the whole program is $\mathsf{nat} \times (\mathsf{I} \times \mathsf{I})$. As explained before, lifting $\mathcal{A}$ over if does not necessarily make the program more readable.

A natural question is how normalizing a program using rules from Fig. 5 compares with first applying CPS translation and then normalizing the obtained purely functional program. The second method produces exactly the same result in the case of Example 1. In the case of Example 2, the result, shown in Fig. 6, involves recursion over a functional type, unlike the one in Fig. 8. If a language has an operator similar to call/cc then we believe that the latter program is preferable for human inspection.

One can see, especially from the first example, that simplifying rules make a big difference in readability of extracted programs. However, both resulting programs still contain many unnecessary occurrences of the constant ε of type I. In fact, one can freely remove them from the program of Example 1, but this is an exception rather than a rule. As was pointed out, the whole recursion operator in the second program has a singleton type, so it cannot be removed. Distinguishing which subterms of singleton type can be removed is the goal of the next section.

4 Removing terms of singleton type

In this section, in contrast to the previous one, we develop semantics-preserving means of manipulating programs which are used in removing subterms of singleton type wherever possible. This means that this technique is applicable not only to programs extracted from proofs, but to those written by hand as well. On the other hand, this approach relies on having a definite evaluation strategy. Since both Scheme and SML/NJ, languages which have call/cc construct, use call-by-value (CBV) strategy, we adopt it here as well.

4.1 Simple types

Since the techniques of this section are applicable to arbitrary well-typed programs, we'd like to move from the type system which associates terms with formulas to the one which associates terms with simple types. The latter are given by the following grammar.

$$\rho ::= \mathsf{I} \mid \mathsf{nat} \mid \mathsf{bool} \mid \ldots \text{ (other base types)} \mid \rho_1 \times \rho_2 \mid \rho_1 + \rho_2 \mid \rho \to \rho$$

The mapping from formulas to types is the following.

$$\begin{aligned} \kappa(A) &= \mathsf{I} \quad A \text{ atomic} & \kappa(D_1 \vee D_2) &= \kappa(D_1) + \kappa(D_2) \\ \kappa(C \to D) &= \kappa(C) \to \kappa(D) & \kappa(\forall x^\rho.\, D) &= \rho \to \kappa(D) \\ \kappa(D_1 \wedge D_2) &= \kappa(D_1) \times \kappa(D_2) & \kappa(\exists x^\rho.\, D) &= \rho \times \kappa(D) \end{aligned}$$

Let us denote $\varphi \equiv \kappa(F)$.

Along with the shift to simple types, we can introduce a new construct

$$\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2\ .$$

Define $\mathsf{let}\ x_1 = t_1; \ldots; x_n = t_n\ \mathsf{in}\ t$ to be an abbreviation for $\mathsf{let}\ x_1 = t_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = t_n\ \mathsf{in}\ t$. Now we can dispense with some other constructs according to the following definitions.

$$\begin{aligned} \mathsf{spread}(t_1; x_1, x_2.t_2) &\equiv \mathsf{let}\ z = t_1\ \mathsf{in}\ \mathsf{let}\ x_1 = \pi_1\, z; x_2 = \pi_2\, z\ \mathsf{in}\ t_2 \\ \mathsf{false}(t_1, t_2) &\equiv \mathsf{let}\ x_1 = t_1\ \mathsf{in}\ \mathsf{let}\ x_2 = t_2\ \mathsf{in}\ c_\varphi \\ \mathsf{begin}(t_1, t_2) &\equiv \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \qquad x \notin \mathrm{FV}(t_2) \end{aligned}$$

Here c_φ is some constant of type φ.

The typing rules are shown in Fig. 9. For the rest of this section, we assume that all programs considered here are well-typed.

Call-by-value reductions for this calculus are shown in Fig. 10.

Define $t_1 \mapsto_{\beta c} t_2$ if either $t_1 \mapsto_c t_2$ or $t_1 \equiv E[t_1']$, $t_2 \equiv E[t_2']$ and $t_1' \succ_\beta t_2'$. Also let $\mathsf{eval}(t_1) = t_2$ if $t_1 \mapsto\!\!\!\!\to_{\beta c} t_2$ and $\mapsto_{\beta c}$ is not applicable to t_2. It is straightforward to show that the output of eval on closed terms is a value. In his thesis ([7, Theorem 9.10.3]), Murthy proved that eval is defined on all well-typed terms in our language.

Let us define $t_1 \sim t_2$ to mean that $\mathsf{eval}(t_1) = \mathsf{eval}(t_2)$. This is not contextual equivalence because $\mapsto_c$ is a computational rule (i.e., it can be applied only to the whole term), but it is sufficient for our purposes.

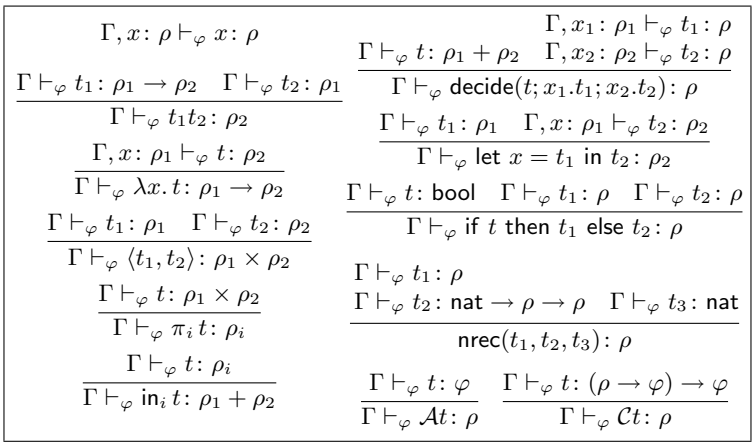


Fig. 9. Typing rules for λ-calculus with control operators

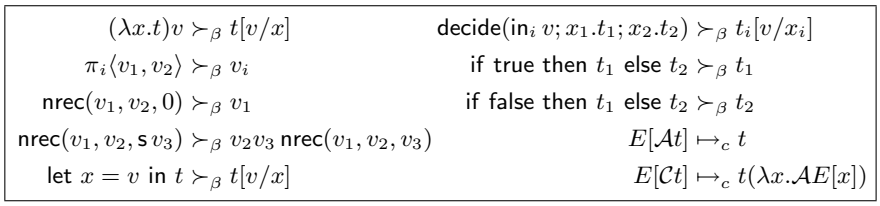


Fig. 10. CBV reductions

4.2 Rationale for extended reductions

For the purpose of pruning terms of singleton type from programs we need to consider more general reductions than CBV ones, where, for example, procedure arguments are not necessarily values. The reason for this is the following.

We define a singleton type-erasing mapping erase both on types and on terms. In the simplest case, a function from a singleton type to some other type ρ should be mapped into a constant of type $\mathsf{erase}(\rho)$. However, this in general cannot be done when the body of the function uses control operators. The simplest counterexample is given by the term

$$(\lambda f^{\mathsf{l}\to\mathsf{nat}}.\, 0)(\lambda x^{\mathsf{l}}.\, \mathcal{A}1) \tag{2}$$

If we put $\mathsf{erase}(\mathsf{l} \to \mathsf{nat}) = \mathsf{nat}$ and change the terms correspondingly, we get

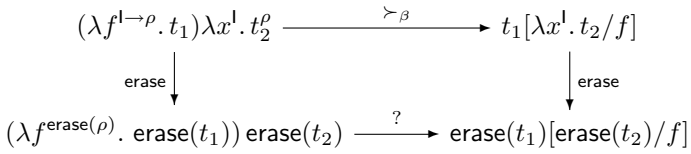
$$\mathsf{erase}((\lambda f^{\mathsf{l}\to\mathsf{nat}}.\, 0)(\lambda x^{\mathsf{l}}.\, \mathcal{A}1)) \equiv (\lambda f^{\mathsf{nat}}.\, 0)(\mathcal{A}1) \tag{3}$$

Now under CBV strategy, (2) evaluates to 0 while (3) evaluates to 1.

The problem here is that even though the argument $\lambda x^{\mathsf{l}}.\, \mathcal{A}1$ is a value, the result of simplification $\mathcal{A}1$ is not. Nevertheless, the idea of turning functions from

singleton type into constants works provided evaluation of the function's body does not invoke control operators (this is the case, for example, with programs extracted from proofs in minimal logic).

To clarify our task, consider the following diagram.



In order to explain how erase preserves the semantics, we need to relate the bottom terms, $t' \equiv (\lambda f^{\mathsf{erase}(\rho)}.\ \mathsf{erase}(t_1))\,\mathsf{erase}(t_2)$ and $t'' \equiv \mathsf{erase}(t_1)[\mathsf{erase}(t_2)/f]$. It is clear that t' does not in general evaluate to t'' according to CBV strategy because $\mathsf{erase}(t_2)$ does not have to be a value, and when it is not, it is going to be evaluated first. However, if evaluation of $\mathsf{erase}(t_2)$ does not invoke control operators, it does not matter when to evaluate $\mathsf{erase}(t_2)$—when it is in the argument position or when it is substituted for f in $\mathsf{erase}(t_1)$. The result of complete evaluation of t' and t'' should be the same.[2] (Of course, the number of times $\mathsf{erase}(t_2)$ is evaluated may be different in the two cases.) Therefore, this is the conjecture which we are going to justify.

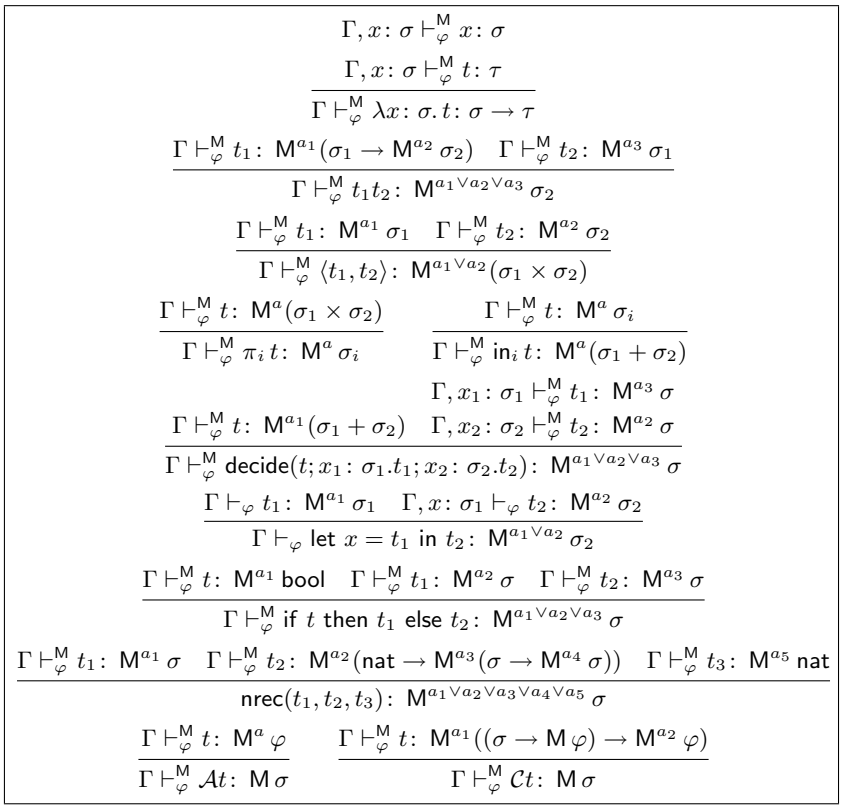
Terms which (even though they may contain control operators) do not invoke control reductions during their evaluation are called *pure*. From the standpoint of evaluation pure terms are equivalent to values. Therefore, we are going to to extend CBV β-reductions systematically replacing the word "value" by the phrase "pure term."

4.3 Extending β-reductions

The idea is to take typing information into account. Inspired by Moggi's monadic metalanguage (MML), we introduce a distinction between computation and values. Namely, a new unary connective M on types is introduced, and types of the form $\mathsf{M}\,\sigma$ are called *computational* types while those which do not start with M are called *pure* types. Similarly, terms of computation types are called *computational* terms while terms of pure types are called *pure* terms. Evaluating pure terms never invokes a control reduction while evaluating computation terms may do so. In the future, p with possible indexes will range over pure terms.

While our types are similar to those in MML, we do not consider a translation of our language into MML. This would require in particular proving correctness of such translation. It turns out that it is possible to justify the extension of β-reductions in our language with extended type system.

[2] To remind, we only consider terms whose evaluation terminates. It is clear that the statement above is in general false when evaluation of $\mathsf{erase}(t_2)$ diverges. Potential nontermination, along with control reduction, should be considered a *side effect*.



$$\Gamma, x\colon \sigma \vdash^{\mathsf{M}}_{\varphi} x\colon \sigma$$

$$\frac{\Gamma, x\colon \sigma \vdash^{\mathsf{M}}_{\varphi} t\colon \tau}{\Gamma \vdash^{\mathsf{M}}_{\varphi} \lambda x\colon \sigma.\, t\colon \sigma \to \tau}$$

$$\frac{\Gamma \vdash^{\mathsf{M}}_{\varphi} t_1\colon \mathsf{M}^{a_1}(\sigma_1 \to \mathsf{M}^{a_2}\,\sigma_2) \quad \Gamma \vdash^{\mathsf{M}}_{\varphi} t_2\colon \mathsf{M}^{a_3}\,\sigma_1}{\Gamma \vdash^{\mathsf{M}}_{\varphi} t_1 t_2\colon \mathsf{M}^{a_1 \vee a_2 \vee a_3}\,\sigma_2}$$

$$\frac{\Gamma \vdash^{\mathsf{M}}_{\varphi} t_1\colon \mathsf{M}^{a_1}\,\sigma_1 \quad \Gamma \vdash^{\mathsf{M}}_{\varphi} t_2\colon \mathsf{M}^{a_2}\,\sigma_2}{\Gamma \vdash^{\mathsf{M}}_{\varphi} \langle t_1, t_2\rangle\colon \mathsf{M}^{a_1 \vee a_2}(\sigma_1 \times \sigma_2)}$$

$$\frac{\Gamma \vdash^{\mathsf{M}}_{\varphi} t\colon \mathsf{M}^{a}(\sigma_1 \times \sigma_2)}{\Gamma \vdash^{\mathsf{M}}_{\varphi} \pi_i\, t\colon \mathsf{M}^{a}\,\sigma_i} \qquad \frac{\Gamma \vdash^{\mathsf{M}}_{\varphi} t\colon \mathsf{M}^{a}\,\sigma_i}{\Gamma \vdash^{\mathsf{M}}_{\varphi} \mathsf{in}_i\, t\colon \mathsf{M}^{a}(\sigma_1 + \sigma_2)}$$

$$\frac{\Gamma \vdash^{\mathsf{M}}_{\varphi} t\colon \mathsf{M}^{a_1}(\sigma_1 + \sigma_2) \quad \begin{array}{l}\Gamma, x_1\colon \sigma_1 \vdash^{\mathsf{M}}_{\varphi} t_1\colon \mathsf{M}^{a_3}\,\sigma \\ \Gamma, x_2\colon \sigma_2 \vdash^{\mathsf{M}}_{\varphi} t_2\colon \mathsf{M}^{a_2}\,\sigma\end{array}}{\Gamma \vdash^{\mathsf{M}}_{\varphi} \mathsf{decide}(t; x_1\colon \sigma_1.t_1; x_2\colon \sigma_2.t_2)\colon \mathsf{M}^{a_1 \vee a_2 \vee a_3}\,\sigma}$$

$$\frac{\Gamma \vdash_{\varphi} t_1\colon \mathsf{M}^{a_1}\,\sigma_1 \quad \Gamma, x\colon \sigma_1 \vdash_{\varphi} t_2\colon \mathsf{M}^{a_2}\,\sigma_2}{\Gamma \vdash_{\varphi} \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2\colon \mathsf{M}^{a_1 \vee a_2}\,\sigma_2}$$

$$\frac{\Gamma \vdash^{\mathsf{M}}_{\varphi} t\colon \mathsf{M}^{a_1}\,\mathsf{bool} \quad \Gamma \vdash^{\mathsf{M}}_{\varphi} t_1\colon \mathsf{M}^{a_2}\,\sigma \quad \Gamma \vdash^{\mathsf{M}}_{\varphi} t_2\colon \mathsf{M}^{a_3}\,\sigma}{\Gamma \vdash^{\mathsf{M}}_{\varphi} \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\colon \mathsf{M}^{a_1 \vee a_2 \vee a_3}\,\sigma}$$

$$\frac{\Gamma \vdash^{\mathsf{M}}_{\varphi} t_1\colon \mathsf{M}^{a_1}\,\sigma \quad \Gamma \vdash^{\mathsf{M}}_{\varphi} t_2\colon \mathsf{M}^{a_2}(\mathsf{nat} \to \mathsf{M}^{a_3}(\sigma \to \mathsf{M}^{a_4}\,\sigma)) \quad \Gamma \vdash^{\mathsf{M}}_{\varphi} t_3\colon \mathsf{M}^{a_5}\,\mathsf{nat}}{\mathsf{nrec}(t_1, t_2, t_3)\colon \mathsf{M}^{a_1 \vee a_2 \vee a_3 \vee a_4 \vee a_5}\,\sigma}$$

$$\frac{\Gamma \vdash^{\mathsf{M}}_{\varphi} t\colon \mathsf{M}^{a}\,\varphi}{\Gamma \vdash^{\mathsf{M}}_{\varphi} \mathcal{A}t\colon \mathsf{M}\,\sigma} \qquad \frac{\Gamma \vdash^{\mathsf{M}}_{\varphi} t\colon \mathsf{M}^{a_1}((\sigma \to \mathsf{M}\,\varphi) \to \mathsf{M}^{a_2}\,\varphi)}{\Gamma \vdash^{\mathsf{M}}_{\varphi} \mathcal{C}t\colon \mathsf{M}\,\sigma}$$

Fig. 11. Type-and-effect system for CBV λ-calculus with control operators

Extended type system Consider types defined by the following grammar.

$$\sigma ::= \mathsf{I} \mid \mathsf{nat} \mid \mathsf{bool} \mid \ldots \text{ (other base types)} \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \mid \sigma \to \tau$$
$$\tau ::= \sigma \mid \mathsf{M}\,\sigma$$

A simple type σ which has no occurrences of M can be converted into τ by inserting M in certain places, namely, in the beginning of σ and after some or all of $\to$'s. We introduce a metalevel notation M^a where a is a proposition. If a is true then M^a must be replaced by M, and if a is false, then M^a must be removed. For example, $\mathsf{M}^{a_1}(\mathsf{I} \to \mathsf{M}^{a_2}\,\mathsf{nat}) \equiv \mathsf{I} \to \mathsf{M}\,\mathsf{nat}$ if a_1 is false and a_2 is true.

In order to reveal evaluation of which subterms in CBV language can cause a side effect, programs are converted into the type-and-effect system shown in Fig. 11 (in the context Γ, all types are from the class σ).

The elements of Fig. 11 are schemata which ranges not only over terms and types but also over truth values of variables $a, a_1, \ldots$. For example, the schema

for term application expands into eight variants based on the values of a_1, a_2 and a_3.

Theorem 3. *If* $\Gamma \vdash_\varphi t : \rho$ *then* $\Gamma \vdash^{\mathsf{M}}_\varphi t : \tau$ *for some* τ.

The proof relies on the fact that a term monotonically inherits computational type from its subterms. Therefore, it is possible to go through a typing derivation top to bottom and put M where they are necessary according to the typing rules. In this process we won't encounter a contradiction because no rule demands the absence of M.

Extended reduction rules As was said before, the idea of extended reductions is to replace the word "value" by the phrase "pure term." Extended β-reductions are defined in Fig. 12.

$$
\begin{array}{rr}
(\lambda x.\, t)p \succ^e_\beta t[p/x] & \\
\pi_1\langle t, p\rangle \succ^e_\beta t & \mathsf{nrec}(t, p, 0) \succ^e_\beta t \\
\pi_2\langle p, t\rangle \succ^e_\beta t & \mathsf{nrec}(p_1, p_2, \mathsf{s}\, p_3) \succ^e_\beta p_2 p_3\, \mathsf{nrec}(p_1, p_2, p_3) \\
\mathsf{let}\ x = p\ \mathsf{in}\ t \succ^e_\beta t[p/x] & \mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2 \succ^e_\beta t_1 \\
\mathsf{decide}(\mathsf{in}_i\, p; x_1.\, t_1; x_2.\, t_2) \succ^e_\beta t_i[p/x_i] & \mathsf{if\ false\ then}\ t_1\ \mathsf{else}\ t_2 \succ^e_\beta t_2
\end{array}
$$

Fig. 12. Extended β-reductions

Theorem 4. *The reduction* $\to^e_\beta \cup \mapsto_c$ *is confluent and enjoys subject reduction.*

Proof. Subject reduction relies on the fact that only pure subterms are substituted for variables, which are themselves always pure. Confluence of $\to^e_\beta$ is proved by regular Tait-Martin-Löf method, and then $\to^e_\beta$ is shown to have a *commuting diamond property* with $\mapsto_c$ (the latter relation is a partial function and is therefore trivially confluent).

Corollary 5. *If* t_1 *and* t_2 *are closed terms of arrow-free type and* $t_1 =^e_{\beta c} t_2$ *then* $t_1 \sim t_2$.

Proof. Since evaluation always terminates, denote $v_1 \equiv \mathsf{eval}(t_1)$, $v_2 \equiv \mathsf{eval}(t_2)$. By the definition of eval, $t_1 \mapsto\!\!\!\!\to_{\beta c} v_1$ and $t_2 \mapsto\!\!\!\!\to_{\beta c} v_2$; therefore, $v_1 =^e_{\beta c} v_2$. Since confluence implies Church-Rosser property, v_1 and v_2 must be reducible to a common term. However, it is easy to see by induction on type that a value of an arrow-free type is normal. Therefore, $v_1 \equiv v_2$.

Another reduction will be useful for simplifying terms.

$$\mathsf{let}\ x = t\ \mathsf{in}\ E[x] \succ_{\mathsf{let}} E[t]$$

Even though, unlike all reductions so far, it may substitute computational terms for variables, it is easy to show that it also enjoys subject reduction. Moreover, $\rightarrow_{\mathsf{let}} \cup \mapsto_{\beta c}$ is confluent. Therefore, as before, if t_1 and t_2 are closed terms of arrow-free type and $t_1 =_{\mathsf{let}} t_2$ then $t_1 \sim t_2$.

4.4 Removing singleton terms

Removal of subterms of singleton type is done in three stages. During the first stage the immediate subterms of a given term are lifted up to be evaluated by a newly introduced let construct. For example,

$$t_1 t_2 \qquad \text{is converted to} \qquad \mathsf{let}\ x_1 = t_1; x_2 = t_2\ \mathsf{in}\ x_1 x_2$$

This transformation makes immediate subterms of most term constructors pure, which significantly simplifies the second stage.

The second stage is the main part of the simplification, where the type of the term changes and subterms of singleton type are erased. For example, a function of a singleton argument is turned into a constant unless the body of the function uses side effects. (In the latter case the function is a *thunk* and delays execution of the body until it is invoked; in such case λ-abstraction must be preserved.) This stage is similar to the construction of pure λ-terms from derivations in the proof of soundness of modified realizability in [8], which was later streamlined by Schwichtenberg in [9].

The third stage is the reverse of the first one and consists of bringing down the terms evaluated in let, whenever possible. It is always possible to bring down pure terms using extended β-reductions. Computational terms can be brought down using let-reduction if the body of the let is an evaluation context. Otherwise, eliminating let is generally impossible. In particular, nothing can be done if the variable in the left-hand side of let is not used in the let's body. For an example of this situation, consider two terms $t_1 \colon \mathsf{M}(\mathsf{I} \to \sigma)$ and $t_2 \colon \mathsf{M}\,\mathsf{I}$. After the first stage, the application $t_1 t_2$ is converted into

$$\mathsf{let}\ x_1 = t_1; x_2 = t_2\ \mathsf{in}\ x_1 x_2$$

After the second stage x_1, which was a function from I to σ, becomes a constant and the argument x_2 of type I is erased; therefore, the term becomes

$$\mathsf{let}\ x_1 = t_1'; x_2 = t_2'\ \mathsf{in}\ x_1$$

where t_1' and t_2' are the results of transformation of t_1 and t_2, respectively. The third stage does not change the term because the body of let cannot be represented as $E[x_2]$ for an evaluation context E. Neither can t_1' be substituted for x_1 since this would change the order of execution (which involves side effects) of t_1' and t_2'.

$$
\begin{array}{rcl}
\mathsf{lift}(x) & = & x \\
\mathsf{lift}(t_1 t_2) & = & \mathsf{let}\ x_1 = \mathsf{lift}(t_1); x_2 = \mathsf{lift}(t_2)\ \mathsf{in}\ x_1 x_2 \\
\mathsf{lift}(\lambda x.\, t) & = & \lambda x.\ \mathsf{lift}(t) \\
\mathsf{lift}(\langle t_1, t_2 \rangle) & = & \mathsf{let}\ x_1 = \mathsf{lift}(t_1); x_2 = \mathsf{lift}(t_2)\ \mathsf{in}\ \langle x_1, x_2 \rangle \\
\mathsf{lift}(\pi_i\, t) & = & \mathsf{let}\ x = \mathsf{lift}(t)\ \mathsf{in}\ \ \pi_i\, x \\
\mathsf{lift}(\mathsf{in}_i\, t) & = & \mathsf{let}\ x = \mathsf{lift}(t)\ \mathsf{in}\ \ \mathsf{in}_i\, x \\
\mathsf{lift}(\mathsf{decide}(t; x_1.t_1; x_2.t_2)) & = & \mathsf{let}\ x = \mathsf{lift}(t)\ \mathsf{in} \\
 & & \quad \mathsf{decide}(x; x_1.\, \mathsf{lift}(t_1); x_2.\, \mathsf{lift}(t_2)) \\
\mathsf{lift}(\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2) & = & \mathsf{let}\ x = \mathsf{lift}(t_1)\ \mathsf{in}\ \ \mathsf{lift}(t_2) \\
\mathsf{lift}(\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) & = & \mathsf{let}\ x = \mathsf{lift}(t)\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ \ \mathsf{lift}(t_1)\ \mathsf{else}\ \ \mathsf{lift}(t_2) \\
\mathsf{lift}(\mathsf{nrec}(t_1, t_2, t_3)) & = & \mathsf{let}\ x_1 = \mathsf{lift}(t_1); x_2 = \mathsf{lift}(t_2); x_3 = \mathsf{lift}(t_3)\ \mathsf{in} \\
 & & \quad \mathsf{nrec}(x_1, x_2, x_3) \\
\mathsf{lift}(\mathcal{A} t) & = & \mathsf{let}\ x = \mathsf{lift}(t)\ \mathsf{in}\ \mathcal{A} x \\
\mathsf{lift}(\mathcal{C} t) & = & \mathsf{let}\ x = \mathsf{lift}(t)\ \mathsf{in}\ \mathcal{C} x
\end{array}
$$

Fig. 13. First stage of singleton-removing transformation

First stage: lifting subterms The first part of the transformation is shown in Fig. 13.

The mapping lift does not essentially change terms because it preserves the semantics.

Theorem 6. *For any term t, $\mathsf{lift}(t) \twoheadrightarrow_{\mathsf{let}} t$. Therefore, for closed t of arrow-free type, $\mathsf{lift}(t) \sim t$.*

Proof. By induction on t. Note that for every innermost let introduced by lift, its body is an evaluation context with respect to that let's variable.

Now that we have immediate subterms having pure type, the second stage can be significantly simplified.

Second stage: removing singleton subterms The mapping erase is first defined on types (see Fig. 14) and then extended to terms.

Let us call types σ for which $\mathsf{erase}(\sigma) = \mathsf{I}$, *trivial* types, and terms of trivial type, *trivial* terms.

Definition of erase on terms is shown in Fig. 15.

Some explanation of the definition may be helpful. As was said before, a function of a trivial argument, represented by λ-abstraction, is converted into a constant unless the body of the λ-abstraction is computational. In the latter case, removing λ-abstraction would result in execution of the body (and invocation of a side effect) when it is encountered and not when it is applied, which would change the semantics of the program. Therefore, λ-abstraction is preserved if

$$
\begin{aligned}
\mathsf{erase}(\sigma) &= \sigma && \sigma \text{ a base type}\\
\mathsf{erase}(\sigma_1 \times \sigma_2) &= \begin{cases} \mathsf{erase}(\sigma_i) & \mathsf{erase}(\sigma_{3-i}) = \mathsf{I} \\ \mathsf{erase}(\sigma_1) \times \mathsf{erase}(\sigma_2) & \text{otherwise} \end{cases}\\
\mathsf{erase}(\sigma_1 + \sigma_2) &= \mathsf{erase}(\sigma_1) + \mathsf{erase}(\sigma_2)\\
\mathsf{erase}(\sigma_1 \to \mathsf{M}^a\, \sigma_2) &= \begin{cases} \mathsf{erase}(\sigma_2) & a = \bot,\ (\mathsf{erase}(\sigma_1) = \mathsf{I} \text{ or } \mathsf{erase}(\sigma_2) = \mathsf{I}) \\ \mathsf{erase}(\sigma_1) \to \mathsf{erase}(\sigma_2) & \text{otherwise} \end{cases}\\
\mathsf{erase}(\mathsf{M}\,\sigma) &= \mathsf{M}\,\mathsf{erase}(\sigma)
\end{aligned}
$$

Fig. 14. Definition of erase on types

$$
\begin{aligned}
\mathsf{erase}(t^\sigma) &= \varepsilon && \text{if } \mathsf{erase}(\sigma) = \mathsf{I}\\
&\textbf{otherwise:}\\
\mathsf{erase}(x^\sigma) &= x^{\mathsf{erase}(\sigma)}\\
\mathsf{erase}(\lambda x^{\sigma_1}.\, t^{M^a \sigma_2}) &= \begin{cases} \mathsf{erase}(t) & a = \bot,\ \mathsf{erase}(\sigma_1) = \mathsf{I} \\ \lambda x^{\mathsf{erase}(\sigma_1)}.\ \mathsf{erase}(t) & \text{otherwise} \end{cases}\\
\mathsf{erase}(t_1^{\sigma_1 \to \mathsf{M}^a\ \sigma_2} t_2^{\sigma_1}) &= \begin{cases} \mathsf{erase}(t_1) & a = \bot,\ \mathsf{erase}(\sigma_1) = \mathsf{I} \\ \mathsf{erase}(t_1)\,\mathsf{erase}(t_2) & \text{otherwise} \end{cases}\\
\mathsf{erase}(\langle t_1^{\sigma_1}, t_2^{\sigma_2}\rangle) &= \begin{cases} \mathsf{erase}(t_i) & \mathsf{erase}(\sigma_{3-i}) = \mathsf{I} \\ \langle \mathsf{erase}(t_1), \mathsf{erase}(t_2)\rangle & \text{otherwise} \end{cases}\\
\mathsf{erase}(\pi_i\, t^{\sigma_1 \times \sigma_2}) &= \begin{cases} \mathsf{erase}(t) & \mathsf{erase}(\sigma_{3-i}) = \mathsf{I} \\ \pi_i\,\mathsf{erase}(t) & \text{otherwise} \end{cases}\\
\mathsf{erase}(\mathsf{in}_i\, t) &= \mathsf{in}_i\,\mathsf{erase}(t)\\
&\mathsf{erase}(\mathsf{decide}(t; x_1.t_1^{\mathsf{M}^{a_1}\ \sigma}; x_2.t_2^{\mathsf{M}^{a_2}\ \sigma}))\\
&\qquad = \mathsf{decide}(\mathsf{erase}(t); x_1.\,\mathsf{erase}(t_1); x_2.\,\mathsf{erase}(t_2))\\
&\mathsf{erase}(\mathsf{if}\ t\ \mathsf{then}\ t_1^{\mathsf{M}^{a_1}\ \sigma}\ \mathsf{else}\ t_2^{\mathsf{M}^{a_2}\ \sigma})\\
&\qquad = \mathsf{if}\ \mathsf{erase}(t)\ \mathsf{then}\ \mathsf{erase}(t_1)\ \mathsf{else}\ \mathsf{erase}(t_2)\\
&\mathsf{erase}(\mathsf{let}\ x = t_1^{\mathsf{M}^{a_1}\ \sigma_1}\ \mathsf{in}\ t_2^{\mathsf{M}^{a_2}\ \sigma_2})\\
&\qquad = \begin{cases} \mathsf{erase}(t_2) & a_1 = \bot,\ \mathsf{erase}(\sigma_1) = \mathsf{I} \\ \mathsf{let}\ x = \mathsf{erase}(t_1)\ \mathsf{in} \\ \qquad\qquad \mathsf{erase}(t_2) & \text{otherwise} \end{cases}\\
&\mathsf{erase}(\mathsf{nrec}(t_1^\sigma, t_2^{\mathsf{nat} \to \mathsf{M}^{a_1}(\sigma \to \mathsf{M}^{a_2}\ \sigma)}, t_3^{\mathsf{nat}}))\\
&\qquad = \mathsf{nrec}(\mathsf{erase}(t_1), \mathsf{erase}(t_2), \\
&\qquad\qquad \mathsf{erase}(t_3)) && a_1 = \bot\\
\mathsf{erase}(\mathcal{A}t) &= \mathcal{A}\,\mathsf{erase}(t)\\
\mathsf{erase}(\mathcal{C}t^{(\sigma \to \mathsf{M}\,\varphi) \to \mathsf{M}^a\ \varphi}) &= \mathcal{C}\,\mathsf{erase}(t) && \mathsf{erase}(\varphi) \neq \mathsf{I}
\end{aligned}
$$

Fig. 15. Definition of erase on terms

the body is computational. In the case of let, however, even though let can be expressed using λ, one does not have to check if the body is computational because the body is executed when it is encountered.

Theorem 7. *If* $t_1 \to_{\beta c} t_2$ *then* $\mathsf{erase}(t_1) =^{e}_{\beta c} \mathsf{erase}(t_2)$.

Third stage: bringing down subterms In the third stage, we apply let-reduction to bring down subterms whenever possible. These reduction must be applied "inside out," i.e., starting with the inmost let. After no more let-reductions can be applied, extended β-reductions for let may still be applicable. Let us slightly modify the example in the beginning of the subsection by assuming that $t_1 \colon \mathsf{I} \to \sigma$ and $t_2 \colon \mathsf{M}\,\mathsf{I}$. Then the first two stages convert the term $t_1 t_2$ into

$$\mathsf{let}\ x_1 = t_1'; x_2 = t_2'\ \mathsf{in}\ x_1$$

where $t_i' \equiv \mathsf{erase}(\mathsf{lift}(t_i))$, $i = 1, 2$ and t_1' of type $\mathsf{erase}(\sigma)$ is a pure term. The term t_2' cannot be brought down but t_1' can (using extended β-reduction) with the following result.

$$\mathsf{let}\ x_2 = t_2'\ \mathsf{in}\ t_1'$$

Let us denote the transformation of the third stage down, and let $\mathsf{prune}(t)$ denote $\mathsf{down}(\mathsf{erase}(\mathsf{lift}(t)))$. The following statement holds.

Theorem 8. *Suppose* t *is a closed term of an arrow-free type and let* $\mathsf{eval}(t) \equiv v$. *Then* $\mathsf{eval}(\mathsf{prune}(t)) \equiv \mathsf{prune}(v)$.

Proof. Suppose that $t \equiv t_1 \mapsto_{\beta c} \cdots \mapsto_{\beta c} t_n \equiv v$. By induction on n we show that $\mathsf{prune}(t) \sim \mathsf{prune}(v)$. Then it is easy to see that $\mathsf{prune}(v)$ is a value and therefore, $\mathsf{eval}(\mathsf{prune}(t)) \equiv \mathsf{prune}(v)$. As for the fact that $t_i \mapsto_{\beta c} t_{i+1}$ implies $\mathsf{prune}(t_i) \sim \mathsf{prune}(t_{i+1})$, it follows from Corollary 5 and Theorems 6 and 7.

4.5 Examples

The result of applying prune to programs extracted from our examples is shown in Figs. 16 and 17 (in the second program we removed false and left $\mathcal{A}$ inside if).

5 Conclusion

We described two ways to simplify programs extracted from classical proofs: normalizing with respect to a certain set of rules and removing trivial subterms. The latter transformation is applicable to arbitrary functional programs with side effects, not only to those obtained from proofs.

Perhaps the best-known project of classical proof extraction is Minlog which is done by Schwichtenberg's group in Munich [9]. This method produces purely functional programs. Its advantage over CPS translation is that the result is optimized with respect to type rank. For example, the result of program extraction

```
if f(0)
  then if f(1)
         then ⟨0, 1⟩
         else if f(2) then ⟨0, 2⟩ else ⟨1, 2⟩
  else if f(1)
       then if f(2)
              then ⟨1, 2⟩
              else if f(3) then ⟨1, 3⟩ else ⟨2, 3⟩
       else ⟨0, 1⟩
```

Fig. 16. Result of removing trivial subterms from the program from Example 1

$$\mathsf{nrec}(\varepsilon, \lambda x \lambda z.\, \mathsf{if}\ f(x+1) > y\ \mathsf{then}\ \mathcal{A}x\ \mathsf{else}\ \varepsilon, g(y))$$

Fig. 17. Result of removing trivial subterms from the program from Example 2

for Example 1 is the same as in Fig. 16, but the result for Example 2 is as follows (see [2]).

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

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

One can conclude that Schwichtenberg's method, in order to produce a program without control operators and with recursion on natural numbers instead of functions (as in Fig. 6), performs a nontrivial modification of the algorithm. (This is also supported by the fact that a program extracted in Minlog from a classical existence proof of Fibonacci numbers passes λ-expressions and not pairs, see [10], while our method produces a program which passes pairs of numbers, as one would expect.) However, one can argue that the original algorithm, which throws the first found solution to the top level, is more natural and is in general faster. Therefore, there appears to be some benefit in leaving some control operators in simplified programs.

In our type-and-effect system in Fig. 11, computational types of subterms are inherited by terms that contain them. More sophisticated type-and-effect systems, like the one used by Thielecke in [11], is able to *mask* side effects so that even terms which invoke control operators can be pure if side effects are not observable from the outside. Using this type-and-effect system would allow to do more thorough pruning of trivial terms.

Acknowledgments

I'd like to thank Amr Sabry and anonymous referees for useful feedback.

References

1. Coquand, T.: A semantics of evidence for classical arithmetic. The Journal of Symbolic Logic **60** (1995) 325–337
2. Berger, U., Schwichtenberg, H.: Program extraction from classical proofs. In Leivant, D., ed.: Logic and Computational Complexity, International Workshop LCC '94. Volume 960 of Lecture Notes in Computer Science., Springer-Verlag (1995) 177–194
3. Makarov, Y.: Practical program extraction from classical proofs. In Escardó, M., Jung, A., Mislove, M., eds.: Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI). Volume 155 of Electronic Notes in Theoretical Computer Science., Elsevier B.V. (2006) 521–542
4. Felleisen, M., Hieb, R.: A revised report on the syntactic theories of sequential control and state. Theoretical Computer Science **103** (1992) 235–271
5. Ariola, Z.M., Herbelin, H., Sabry, A.: A proof-theoretic foundation of abortive continuations. Higher-Order and Symbolic Computation **To appear** (2005)
6. Friedman, D.P., Wand, M., Haynes, C.T.: Essentials of Programming Languages. 2nd edn. MIT Press (2001)
7. Murthy, C.R.: Extracting Constructive Content from Classical Proofs. PhD thesis, Cornell University, Department of Computer Science (1990)
8. Troelstra, A.S., ed.: Mathematical Investigation of Intuitionistic Arithmetic and Analysis. Volume 344 of Lecture Notes in Mathematics. Springer-Verlag (1973)
9. Berger, U., Schwichtenberg, H., Seisenberger, M.: The Warshall algorithm and Dickson's lemma: Two examples of realistic program extraction. Journal of Automated Reasoning **26** (2001) 205–221
10. Berger, U., Buchholz, W., Schwichtenberg, H.: Refined program extraction from classical proofs. Annals of Pure and Applied Logic **114** (2002) 3–25
11. Thielecke, H.: From control effects to typed continuation passing. In: POPL'03 (The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages), ACM (2003)