Program Extraction in simply-typed Higher Order Logic

Stefan Berghofer*

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
http://www.in.tum.de/~berghofe/

Abstract. Based on a representation of primitive proof objects as λ-terms, which has been built into the theorem prover Isabelle recently, we propose a generic framework for program extraction. We show how this framework can be used to extract functional programs from proofs conducted in a constructive fragment of the object logic Isabelle/HOL. A characteristic feature of our implementation of program extraction is that it produces both a program and a correctness proof. Since the extracted program is available as a function within the logic, its correctness proof can be checked automatically inside Isabelle.

1 Introduction

One of the most fascinating properties of constructive logic is that a proof of a specification contains an algorithm which, by construction, satisfies this specification. This idea forms the basis for *program extraction* mechanisms, which can be found in theorem provers such as Coq [3] or Nuprl [11]. To date, program extraction has mainly been restricted to theorem provers based on expressive dependent type theories such as the Calculus of Constructions [12]. A notable exception is the Minlog System by Schwichtenberg [5], which is based on minimal first order logic. Although Isabelle is based on simply-typed minimal higher order logic, which is purely constructive, little effort has been devoted to the issue of program extraction in this system so far.

The aim of this paper is to demonstrate that Isabelle is indeed quite suitable as a basis for program extraction. It has already been demonstrated that proofs in Isabelle can be encoded as λ-terms [8]. Based on this encoding, we describe a mechanism that turns an Isabelle proof into a functional program. Since Isabelle is a generic theorem prover, this mechanism will be generic, too. In order to instantiate it for a particular object logic, one has to assign programs to each of its primitive inference rules. By induction on the structure of proof terms, one can then build programs from more complex proofs making use of these inference rules. Since the essence of program extraction is to systematically produce programs that are *correct* by construction, we also describe a transformation

* Supported by DFG Graduiertenkolleg *Logic in Computer Science*, and IST project 29001 *TYPES*

that turns a proof into a correctness proof of the program extracted from it. The precise definition of what is meant by *correctness* will be given by a so-called *realizability interpretation*, that relates programs to logical formulae. The overall architecture of the program extraction framework is shown in Fig. 1. It should

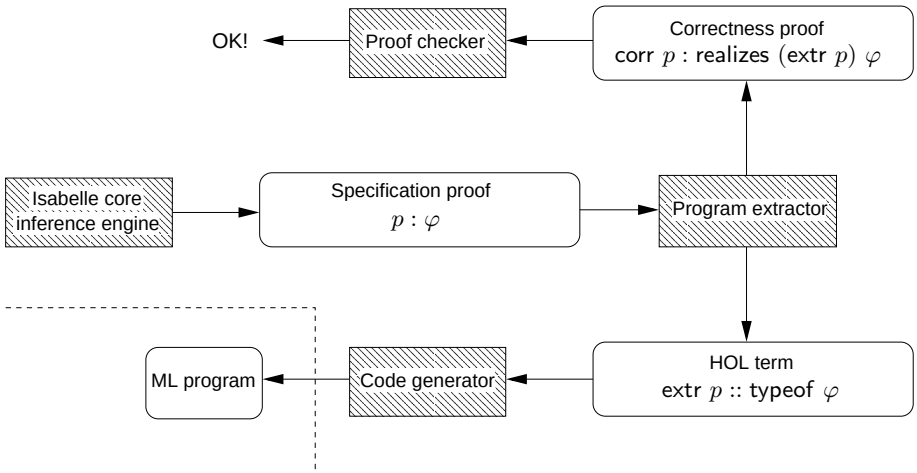

Fig. 1. Architecture of the Isabelle program extraction framework

be noted that the extracted program is actually available as a function in the object logic. Therefore, its proof of correctness can be checked inside Isabelle. The checking process turns the correctness proof into a genuine *theorem*, which may be used in other formalizations together with the extracted program. Finally, using Isabelle's code generator [9], the extracted function can be compiled into an efficiently executable ML program.

The rest of the paper is structured as follows: In §2, we give an overview of the logical system underlying Isabelle, as well as the object logic Isabelle/HOL. In §3, the generic program extraction mechanism will be introduced, whereas §4 describes its adaption to Isabelle/HOL. A case study is described in §5.

2 Preliminaries

2.1 The Isabelle/Pure logical framework

Isabelle offers a *logical framework* in which various different *object logics* can be formalized. Operators, inference rules and proofs of an object logic can be described using the *meta logic* Isabelle/Pure. Isabelle's meta logic essentially consists of three layers, which are summarized in Fig. 2.

Isabelle/Pure offers *simple types* according to Church, for which type inference is decidable. The set of type constructors includes the nullary type con-

$$\tau, \sigma \;=\; \alpha \mid (\tau_1, \ldots, \tau_n)tc \qquad\qquad \text{where } tc \in \{\mathsf{prop}, \Rightarrow, \ldots\}$$

TYPES

$$t, u, P, Q \;=\; x \mid c_{[\overline{\tau_n}/\overline{\alpha_n}]} \mid t\ u \mid \lambda x :: \tau.\ t \qquad\qquad \text{where } c \in \{\bigwedge, \Longrightarrow, \ldots\}$$

$$\frac{}{\Gamma, x :: \tau, \Gamma' \vdash x :: \tau} \qquad \frac{}{\Gamma \vdash c_{[\overline{\tau_n}/\overline{\alpha_n}]} : \Sigma(c)[\overline{\tau_n}/\overline{\alpha_n}]}$$

$$\frac{\Gamma \vdash t :: \tau \Rightarrow \sigma \quad \Gamma \vdash u :: \tau}{\Gamma \vdash t\ u :: \sigma} \qquad \frac{\Gamma, x :: \tau \vdash t :: \sigma}{\Gamma \vdash \lambda x :: \tau.\ t :: \tau \Rightarrow \sigma}$$

TERMS

$$p, q \;=\; h \mid c_{[\overline{\tau_n}/\overline{\alpha_n}]} \mid p \cdot t \mid p \bullet q \mid \boldsymbol{\lambda} x :: \tau.\ p \mid \boldsymbol{\lambda} h : P.\ p$$

$$\frac{}{\Gamma, h : t, \Gamma' \vdash h : t} \qquad \frac{}{\Gamma \vdash c_{[\overline{\tau_n}/\overline{\alpha_n}]} : \Sigma(c)[\overline{\tau_n}/\overline{\alpha_n}]}$$

$$\frac{\Gamma \vdash p : \bigwedge x :: \tau.\ P \quad \Gamma \vdash t :: \tau}{\Gamma \vdash p \cdot t : P[t/x]} \qquad \frac{\Gamma, x :: \tau \vdash p : P}{\Gamma \vdash \boldsymbol{\lambda} x :: \tau.\ p : \bigwedge x :: \tau.\ P}$$

$$\frac{\Gamma \vdash p : P \Longrightarrow Q \quad \Gamma \vdash q : P}{\Gamma \vdash p \bullet q : Q} \qquad \frac{\Gamma, h : P \vdash p : Q}{\Gamma \vdash \boldsymbol{\lambda} h : P.\ p : P \Longrightarrow Q}$$

PROOFS

Fig. 2. The Isabelle/Pure logical framework

structor prop for the type of meta level truth values as well as the binary type constructor $\Rightarrow$ for the function space.

The layer of *terms* is simply-typed λ-calculus, enriched with additional constants, with the usual typing rules. The connectives of the meta logic, namely universal quantification $\bigwedge$ and implication $\Longrightarrow$, are just specific constants. The signature Σ is a function mapping each constant to a type, possibly with free type variables. For example, $\Sigma(\Longrightarrow) = \mathsf{prop} \Rightarrow \mathsf{prop} \Rightarrow \mathsf{prop}$ and $\Sigma(\bigwedge) = (\alpha \Rightarrow \mathsf{prop}) \Rightarrow \mathsf{prop}$. Isabelle offers *schematic polymorphism*: when referring to a constant c, one may instantiate the type variables occurring in its declared type $\Sigma(c)$. Unlike in more expressive dependent type theories, there is no way to explicitly abstract over type variables.

The layer of *proofs* is built on top of the layers of terms and types. The central idea behind the proof layer is the *Curry-Howard isomorphism*, according to which proofs can be represented as λ-terms. Consequently, the proof layer looks quite similar to the term layer, with the difference that there are two kinds of abstractions and two kinds of applications, corresponding to introduc-

tion and elimination of universal quantifiers and implications, respectively. The proof checking rules for $\Longrightarrow$ can be seen as non-dependent variants of the rules for $\bigwedge$. The formulae P and Q in the proof checking rules are terms of type prop. Proof constants c are references to axioms or other theorems that have already been proved. Function Σ maps each proof constant to a term of type prop. Similar to term constants, one may give an instantiation for the free type variables occurring in the proposition corresponding to the proof constant. More background information on Isabelle's proof term calculus can be found in [8].

2.2 Formalizing object logics

When formalizing object logics, one usually introduces a new type of object level truth values, e.g. bool as well as object level logical connectives operating on terms of this type, e.g. $\longrightarrow ::$ bool $\Rightarrow$ bool $\Rightarrow$ bool. Inference rules can then be specified using the meta logic. Fig. 3 shows the inference rules for the constructive fragment of Isabelle/HOL. For these rules to be well-typed, one has to insert a

$$\begin{array}{ll}
\mathsf{impI}: \ (P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q & \mathsf{mp}: \ P \longrightarrow Q \Longrightarrow P \Longrightarrow Q \\
\mathsf{allI}: \ (\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x & \mathsf{spec}: \ \forall x.\ P\ x \Longrightarrow P\ x \\
\mathsf{exI}: \ P\ x \Longrightarrow \exists x.\ P\ x & \mathsf{exE}: \ \exists x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q) \Longrightarrow Q \\
\mathsf{conjI}: \ P \Longrightarrow Q \Longrightarrow P \wedge Q & \mathsf{conjunct}_1: \ P \wedge Q \Longrightarrow P \\
 & \mathsf{conjunct}_2: \ P \wedge Q \Longrightarrow Q \\
\mathsf{disjI}_1: \ P \Longrightarrow P \vee Q & \mathsf{disjE}: \ P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R \\
\mathsf{disjI}_2: \ Q \Longrightarrow P \vee Q & \\
\mathsf{notI}: \ (P \Longrightarrow \mathsf{False}) \Longrightarrow \neg P & \mathsf{notE}: \ \neg P \Longrightarrow P \Longrightarrow R \\
 & \mathsf{FalseE}: \ \mathsf{False} \Longrightarrow P
\end{array}$$

Fig. 3. Constructive inference rules of Isabelle/HOL

coercion function Trueprop :: bool $\Rightarrow$ prop in the right places. These coercion functions, as well as outermost quantifiers binding variables such as P and Q are usually omitted for the sake of readability. Hence, the rule impl actually reads

$$\bigwedge P\ Q.\ (\mathsf{Trueprop}\ P \Longrightarrow \mathsf{Trueprop}\ Q) \Longrightarrow \mathsf{Trueprop}\ (P \longrightarrow Q)$$

Using the proof term calculus introduced in §2.1 together with the rules from Fig. 3, a proof of $(\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)$ becomes:

$$\begin{array}{l}
\mathsf{impI} \cdot \exists x.\ \forall y.\ P\ x\ y \cdot \forall y.\ \exists x.\ P\ x\ y \cdot \\
\quad (\boldsymbol{\lambda} H : \exists x.\ \forall y.\ P\ x\ y.\ \mathsf{allI} \cdot (\lambda y.\ \exists x.\ P\ x\ y) \cdot \\
\quad\quad (\boldsymbol{\lambda} y.\ \mathsf{exE} \cdot (\lambda x.\ \forall y.\ P\ x\ y) \cdot \exists x.\ P\ x\ y \cdot H \cdot \\
\quad\quad\quad (\boldsymbol{\lambda} x\ H : \forall y.\ P\ x\ y.\ \mathsf{exI} \cdot (\lambda x.\ P\ x\ y) \cdot x \cdot (\mathsf{spec} \cdot P\ x \cdot y \cdot H))))
\end{array}$$

3 Program extraction

We now come to the definition of the generic program extraction framework. As described in Fig. 1, it consists of the following ingredients:

- A function typeof which maps a logical formula to the type of the term extracted from its proof
- The actual extraction function extr which extracts a term (i.e. a program) from a proof p with $\Gamma \vdash p : \varphi$, such that $\Gamma \vdash \mathsf{extr}\ \Gamma\ p :: \mathsf{typeof}\ \varphi$
- A function $\mathsf{realizes}$ which, given a term and a logical formula (the *specification*), returns a logical formula describing that the term in some sense satisfies (*"realizes"*) the specification
- A function corr which yields a proof that the program extracted from a proof p realizes the formula proved by p, i.e. $\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ p : \mathsf{realizes}\ (\mathsf{extr}\ \Gamma\ p)\ \varphi$

3.1 Extracting types

The function typeof is specified as a set of (conditional) rewrite rules. It can easily be adapted to specific object logics by adding new rules. Rewrite rules are formulated using Isabelle's term calculus introduced in §2.1. In order to explicitly encode *type* constraints on the level of *terms*, we use a technique due to Wenzel [22]. We introduce a new polymorphic type $\alpha\ \mathsf{itself}$ together with a constant $\mathsf{TYPE} :: \alpha\ \mathsf{itself}$. On top of this, we add a type Type together with a coercion function $\alpha\ \mathsf{itself} \Rightarrow \mathsf{Type}$. Then, typeof will be a function of type $\tau \Rightarrow \mathsf{Type}$, where τ is of the form $\overline{\sigma} \Rightarrow \beta$ with $\beta \in \mathbb{P}$. Here, $\mathbb{P}$ denotes the set of *propositional types*, i.e. $\mathbb{P} = \{\mathsf{prop}, \mathsf{bool}, \ldots\}$. We also introduce a dummy type Null which has the constant Null as its only element. It should be noted that the functions typeof and Type are not actually *defined* within Isabelle/Pure, since doing so would require a kind of *meta-logical framework* [20], but rather serve as syntax to formulate the rewrite rules below.

Intuitively, a program extracted from a proof of $P \Longrightarrow Q$ should be some function of type $\sigma \Rightarrow \tau$. However, not all parts of a formula actually have a computational content. For example, in a formula $x \neq 0 \Longrightarrow \cdots$, the premise $x \neq 0$ merely *verifies* that x has the right value. One possibility would be to simply assign the above formula a function type with a dummy argument type, e.g. $\mathsf{Null} \Rightarrow \tau$. Unfortunately, this would lead to a considerable amount of garbage appearing in the extracted program. Even worse, when recursively extracting programs from lemmas appearing in a proof, one would be forced to extract useless dummy programs for all lemmas involved, regardless of their actual computational content. To remedy this, the type corresponding to a formula will be specified by several rules, depending on the computational content of its subformulae. For example, if P is a formula without computational content, $P \Longrightarrow Q$ would simply correspond to the type τ instead of $\mathsf{Null} \Rightarrow \tau$. Neither the type nor the constant Null may actually occur in extracted programs.

The rules below specify the extracted type corresponding to formulas of Isabelle/Pure. They should be read like a functional program, i.e. earlier rules have precedence over rules appearing later.

$$\mathsf{typeof}\ P \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Type}\ (\mathsf{TYPE}(\tau)) \Longrightarrow \mathsf{typeof}\ (P \Longrightarrow Q) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\tau))$$

$$\mathsf{typeof}\ Q \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \mathsf{typeof}\ (P \Longrightarrow Q) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null}))$$

$$\mathsf{typeof}\ P \equiv \mathsf{Type}\ (\mathsf{TYPE}(\sigma)) \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Type}\ (\mathsf{TYPE}(\tau)) \Longrightarrow \mathsf{typeof}\ (P \Longrightarrow Q) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\sigma \Rightarrow \tau))$$

$$(\lambda x.\ \mathsf{typeof}\ (P\ x)) \equiv (\lambda x.\ \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null}))) \Longrightarrow \mathsf{typeof}\ (\bigwedge x.\ P\ x) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null}))$$

$$(\lambda x.\ \mathsf{typeof}\ (P\ x)) \equiv (\lambda x.\ \mathsf{Type}\ (\mathsf{TYPE}(\sigma))) \Longrightarrow \mathsf{typeof}\ (\bigwedge x :: \alpha.\ P\ x) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\alpha \Rightarrow \sigma))$$

$$(\lambda x.\ \mathsf{typeof}\ (f\ x)) \equiv (\lambda x.\ \mathsf{Type}\ (\mathsf{TYPE}(\tau))) \Longrightarrow \mathsf{typeof}\ f \equiv \mathsf{Type}\ (\mathsf{TYPE}(\tau))$$

We also need to deal with predicate variables occurring in a formula. It depends on the formula a predicate variable is instantiated with, whether or not it contributes to the computational content of the formula it occurs in. If the variable is instantiated with a formula having computational content, we call the variable *computationally relevant*, otherwise *computationally irrelevant*. A computationally relevant predicate variable corresponds to a type variable in the type of the extracted program. During extraction, each computationally relevant predicate variable P is assigned a specific type variable α_P, i.e. $\mathsf{typeof}\ (P\ \bar{t}) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\alpha_P))$. In contrast, $\mathsf{typeof}\ (Q\ \bar{t}) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null}))$ for each computationally irrelevant variable Q. For a theorem with n predicate variables, there are 2^n possibilities for variables being computationally relevant or irrelevant. Thus, we may need to extract up to 2^n different programs from this theorem, depending on the context it is used in. For example, the program extracted from a proof of $(P \Longrightarrow Q \Longrightarrow R) \Longrightarrow (P \Longrightarrow Q) \Longrightarrow P \Longrightarrow R$ will have type $(\alpha_P \Rightarrow \alpha_Q \Rightarrow \alpha_R) \Rightarrow (\alpha_P \Rightarrow \alpha_Q) \Rightarrow \alpha_P \Rightarrow \alpha_R$ if P, Q and R are computationally relevant, whereas it will have type $(\alpha_Q \Rightarrow \alpha_R) \Rightarrow \alpha_Q \Rightarrow \alpha_R$ if just Q and R are computationally relevant. Fortunately, only few of these variants are actually needed in practice, and our extraction mechanism can generate them on demand. Function RVars assigns to each theorem c with parameters $\bar{t}$ the set of its computationally relevant variables. Analogously, TInst yields a suitable type substitution for the type variables corresponding to computationally relevant predicate variables of c. Finally, we use PVars to denote the set of *all* predicate variables of a theorem c.

$$\begin{aligned}
\mathsf{RVars}\ c\ \bar{t} &= \{x_i \mid \Sigma(c) = (\bigwedge \overline{x} :: \overline{\tau}.\ \varphi),\ \tau_i = \overline{\sigma} \Rightarrow \beta,\ \beta \in \mathbb{P},\\
&\qquad \mathsf{typeof}\ t_i \neq \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null}))\}\\
\mathsf{TInst}\ c\ \bar{t} &= \{\alpha_{x_i}/\tau \mid \Sigma(c) = (\bigwedge \overline{x} :: \overline{\tau}.\ \varphi),\ \tau_i = \overline{\sigma} \Rightarrow \beta,\ \beta \in \mathbb{P},\\
&\qquad \mathsf{typeof}\ t_i = \mathsf{Type}\ (\mathsf{TYPE}(\tau)),\ \tau \neq \mathsf{Null}\}\\
\mathsf{PVars}\ c &= \{x_i \mid \Sigma(c) = (\bigwedge \overline{x} :: \overline{\tau}.\ \varphi),\ \tau_i = \overline{\sigma} \Rightarrow \beta,\ \beta \in \mathbb{P}\}
\end{aligned}$$

3.2 Extracting terms

We are now ready to give the definition of the extraction function extr. In addition to the actual proof, extr takes a context Γ as an argument, which associates term variables with types and proof variables with propositions. The extracted term is built up by recursion over the structure of a proof. The proof may refer to other theorems, for which we also need extracted programs. We therefore introduce a function $\mathcal{E}$ which maps a theorem name and a set of predicate variables to a term. We assume $\mathcal{E}$ to contain terms for both complex theorems, whose extracted term has already been computed by earlier invocations of extr, and

primitive inference rules such as exI, for which a corresponding term has been specified by the author of the object logic. In the former case, the result of $\mathcal{E}$ will usually just be some constant referring to a more complex program, which helps to keep the extracted program more modular. As mentioned in §3.1, for theorems with predicate variables, the type of the corresponding program depends on the set of *relevant predicate variables*, which is passed as an additional argument to $\mathcal{E}$. Which predicate variables of a particular occurrence of a theorem are relevant depends on its context, which is why extr takes an additional list of terms $\bar{t}$ describing the arguments of a theorem.

$$
\begin{aligned}
&\mathsf{extr}\ \bar{t}\ \Gamma\ h && = \hat{h} \\
&\mathsf{extr}\ \bar{t}\ \Gamma\ (\boldsymbol{\lambda} x :: \tau.\ p) && = \lambda x :: \tau.\ \mathsf{extr}\ []\ (\Gamma, x :: \tau)\ p \\
&\mathsf{extr}\ \bar{t}\ \Gamma\ (\boldsymbol{\lambda} h : P.\ p) && = \begin{cases} \mathsf{extr}\ []\ (\Gamma, h : P)\ p & \text{if } \tau = \mathsf{Null} \\ \boldsymbol{\lambda}\hat{h} :: \tau.\ \mathsf{extr}\ []\ (\Gamma, h : P)\ p & \text{otherwise} \end{cases} \\
& && \quad \text{where } \mathsf{typeof}\ P = \mathsf{Type}\ (\mathsf{TYPE}(\tau)) \\
&\mathsf{extr}\ \bar{t}\ \Gamma\ (p \cdot t) && = (\mathsf{extr}\ (t\bar{t})\ \Gamma\ p)\ t \\
&\mathsf{extr}\ \bar{t}\ \Gamma\ (p_1 \cdot p_2) && = \begin{cases} \mathsf{extr}\ []\ \Gamma\ p_1 & \text{if } \tau = \mathsf{Null} \\ (\mathsf{extr}\ []\ \Gamma\ p_1)\ (\mathsf{extr}\ []\ \Gamma\ p_2) & \text{otherwise} \end{cases} \\
& && \quad \text{where} \quad \Gamma \vdash p_2 : P \\
& && \qquad\quad \mathsf{typeof}\ P = \mathsf{Type}\ (\mathsf{TYPE}(\tau)) \\
&\mathsf{extr}\ \bar{t}\ \Gamma\ c_{[\overline{\tau_n}/\overline{\alpha_n}]} && = \mathcal{E}(c, \mathsf{RVars}\ c\ \bar{t})[\overline{\tau_n}/\overline{\alpha_n}, \mathsf{TInst}\ c\ \bar{t}]
\end{aligned}
$$

The first clause of extr says that proof variables become term variables in the extracted program. To avoid clashes with already existing term variables, we map each proof variable h to a term variable $\hat{h}$ that does not occur in the original proof. Abstractions on the proof level, i.e. introduction of $\bigwedge$ and $\Longrightarrow$, are turned into abstractions on the program level. In the case of a proof of $P \Longrightarrow Q$, where P has no computational content, the extracted program is a degenerate "function" with no arguments. Analogously, applications on the proof level, i.e. elimination of $\bigwedge$ and $\Longrightarrow$, are turned into applications on the program level. In the case of an elimination of $P \Longrightarrow Q$, where P has no computational content, the function argument is omitted. In the case of a $\bigwedge$-elimination, the term argument t is added to the list $\bar{t}$, since it could be the argument of a theorem constant. When encountering a theorem constant c, the corresponding program is chosen with respect to the current list of term arguments.

3.3 Correctness

It has already been mentioned in §1 that for each extracted program, one can obtain a correctness proof. For this correctness proof to make sense, we first have to make clear what is actually meant by correctness. The key for understanding the correctness of extracted programs is the notion of *realizability*. Realizability establishes a connection between a program and its specification. More precisely, we will specify a predicate $\mathsf{realizes}$ which relates terms (so-called *realizers*) with logical formulae. The notion of realizability was first introduced by Kleene [15]

to study the semantics of intuitionistic logic. In his original formulation, realizers were *Gödel numbers*, which were somewhat hard to work with. To improve on this, Kreisel introduced so-called *modified realizability*, where realizers were actual terms of a kind of programming language, namely Gödel's *system T*. Our characterization of realizability, as well as the one which is described by Schwichtenberg [5], is inspired by Kreisel's modified realizability.

The following set of conditional rewrite rules characterizes realizability for formulae of Isabelle/Pure. As before, earlier rules have higher priority.

$$\mathsf{typeof}\ P \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \\ \quad \mathsf{realizes}\ r\ (P \Longrightarrow Q) \equiv (\mathsf{realizes}\ \mathsf{Null}\ P \Longrightarrow \mathsf{realizes}\ r\ Q)$$

$$\mathsf{typeof}\ P \equiv \mathsf{Type}\ (\mathsf{TYPE}(\sigma)) \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \\ \quad \mathsf{realizes}\ r\ (P \Longrightarrow Q) \equiv (\bigwedge x :: \sigma.\ \mathsf{realizes}\ x\ P \Longrightarrow \mathsf{realizes}\ \mathsf{Null}\ Q)$$

$$\mathsf{realizes}\ r\ (P \Longrightarrow Q) \equiv (\bigwedge x.\ \mathsf{realizes}\ x\ P \Longrightarrow \mathsf{realizes}\ (r\ x)\ Q)$$

$$(\lambda x.\ \mathsf{typeof}\ (P\ x)) \equiv (\lambda x.\ \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null}))) \Longrightarrow \\ \quad \mathsf{realizes}\ r\ (\bigwedge x.\ P\ x) \equiv (\bigwedge x.\ \mathsf{realizes}\ \mathsf{Null}\ (P\ x))$$

$$\mathsf{realizes}\ r\ (\bigwedge x.\ P\ x) \equiv (\bigwedge x.\ \mathsf{realizes}\ (r\ x)\ (P\ x))$$

For example, in the third clause defining realizability for $P \Longrightarrow Q$, P can be thought of as a specification of the *input* of program r, whereas Q specifies its *output*. We can now give a specification of function corr, which produces a correctness proof for the program computed by extr. It has a similar structure as function extr and again works by recursion on the proof. Since a proof may refer to other theorems, we also need a function $\mathcal{C}$ which yields correctness proofs for the programs extracted from these theorems. Its parameters are the same as those for function $\mathcal{E}$ described in §3.2.

$$\begin{array}{lcl}
\mathsf{corr}\ \overline{t}\ \Gamma\ h & = & h \\
\mathsf{corr}\ \overline{t}\ \Gamma\ (\boldsymbol{\lambda} x :: \tau.\ p) & = & \boldsymbol{\lambda} x :: \tau.\ \mathsf{corr}\ []\ (\Gamma, x :: \tau)\ p \\
\mathsf{corr}\ \overline{t}\ \Gamma\ (\boldsymbol{\lambda} h : P.\ p) & = & \begin{cases} \boldsymbol{\lambda} h : \mathsf{realizes}\ \mathsf{Null}\ P.\ \mathsf{corr}\ []\ (\Gamma, h : P)\ p & \text{if } \tau = \mathsf{Null} \\ \boldsymbol{\lambda}(\hat{h} :: \tau)\ (h : \mathsf{realizes}\ \hat{h}\ P).\ \mathsf{corr}\ []\ (\Gamma, h : P)\ p & \text{otherwise} \end{cases} \\
 & & \text{where } \mathsf{typeof}\ P = \mathsf{Type}\ (\mathsf{TYPE}(\tau)) \\
\mathsf{corr}\ \overline{t}\ \Gamma\ (p \cdot t) & = & (\mathsf{corr}\ (t\overline{t})\ \Gamma\ p) \cdot t \\
\mathsf{corr}\ \overline{t}\ \Gamma\ (p_1 \cdot p_2) & = & \begin{cases} \mathsf{corr}\ []\ \Gamma\ p_1 \cdot \mathsf{corr}\ []\ \Gamma\ p_2 & \text{if } \tau = \mathsf{Null} \\ \mathsf{corr}\ []\ \Gamma\ p_1 \cdot \mathsf{extr}\ []\ \Gamma\ p_2 \cdot \mathsf{corr}\ []\ \Gamma\ p_2 & \text{otherwise} \end{cases} \\
 & & \text{where } \Gamma \vdash p_2 : P \\
 & & \quad \mathsf{typeof}\ P = \mathsf{Type}\ (\mathsf{TYPE}(\tau)) \\
\mathsf{corr}\ \overline{t}\ \Gamma\ c_{[\overline{\tau_n}/\overline{\alpha_n}]} & = & \mathcal{C}(c, \mathsf{RVars}\ c\ \overline{t})[\overline{\tau_n}/\overline{\alpha_n}, \mathsf{TInst}\ c\ \overline{t}]
\end{array}$$

The main correctness property relating functions extr and corr can now be stated as follows: Provided that

$$\vdash \mathcal{C}(c,\ V)[\overline{\tau}/\overline{\alpha}] : \mathsf{realizes}\ (\mathcal{E}(c, V)[\overline{\tau}/\overline{\alpha}])\ (\Sigma(c)[\overline{\tau}/\overline{\alpha}])$$

for all c, $\overline{\tau}$, $V \subseteq \mathsf{PVars}(c)$, we have

$$\begin{array}{ll}
\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ []\ \Gamma\ q : \mathsf{realizes}\ \mathsf{Null}\ \varphi & \text{if } \tau = \mathsf{Null} \\
\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ []\ \Gamma\ q : \mathsf{realizes}\ (\mathsf{extr}\ []\ \Gamma\ q)\ \varphi & \text{otherwise}
\end{array}$$

for all Γ, q, τ and φ with $\Gamma \vdash q : \varphi$ and $\mathsf{typeof}\ \varphi = \mathsf{Type}\ (\mathsf{TYPE}(\tau))$, where

$$\begin{array}{lcl}
\mathcal{R}\ [] & = & [] \\
\mathcal{R}\ (x :: \tau, \Gamma) & = & (x :: \tau, \mathcal{R}(\Gamma)) \\
\mathcal{R}\ (h : P, \Gamma) & = & \begin{cases} (h : \mathsf{realizes\ Null}\ P, \mathcal{R}(\Gamma)) & \text{if } \tau = \mathsf{Null} \\ (\hat{h} :: \tau, h : \mathsf{realizes}\ \hat{h}\ P, \mathcal{R}(\Gamma)) & \text{otherwise} \end{cases} \\
 & & \quad \text{where } \mathsf{typeof}\ P = \mathsf{Type}\ (\mathsf{TYPE}(\tau))
\end{array}$$

Function $\mathcal{R}$ is used to express that, when producing a correctness proof for q, one may already assume to have suitable realizers and correctness proofs for each assumption in Γ. Since corr depends on context information $\overline{t}$ for theorems, the above correctness theorem does not hold for arbitrary q, but only for those where each occurrence of a theorem is *fully applied*, i.e. each theorem has as many term arguments as it has outermost $\bigwedge$-quantifiers. The proof of the correctness theorem is by induction on the structure of the fully applied proof q. For lack of space, we only show two particularly interesting cases of the proof.

Case $q = h$ Since $h : \varphi \in \Gamma$, we have $\mathcal{R}(\Gamma) \vdash h : \mathsf{realizes\ Null}\ \varphi$ or $\mathcal{R}(\Gamma) \vdash h : \mathsf{realizes}\ \hat{h}\ \varphi$, as required.

Case $q = (\boldsymbol{\lambda} h : P.\ p)$ Let $\varphi = P \Longrightarrow Q$, $\mathsf{typeof}\ P = \mathsf{Type}\ (\mathsf{TYPE}(\tau))$ and $\mathsf{typeof}\ Q = \mathsf{Type}\ (\mathsf{TYPE}(\sigma))$. If $\tau \neq \mathsf{Null}$ and $\sigma \neq \mathsf{Null}$, then $(\mathcal{R}(\Gamma), \hat{h} :: \tau, h : \mathsf{realizes}\ \hat{h}\ P) \vdash \mathsf{corr}\ []\ (\Gamma, h : P)\ p : \mathsf{realizes}\ (\mathsf{extr}\ []\ (\Gamma, h : P)\ p)\ Q$ by induction hypothesis. Hence, $\mathcal{R}(\Gamma) \vdash \boldsymbol{\lambda}(\hat{h} :: \tau)\ (h : \mathsf{realizes}\ \hat{h}\ P).\ \mathsf{corr}\ []\ (\Gamma, h : P)\ p : \bigwedge \hat{h} :: \tau.\ \mathsf{realizes}\ \hat{h}\ P \Longrightarrow \mathsf{realizes}\ (\mathsf{extr}\ []\ (\Gamma, h : P)\ p)$, as required. The other three subcases are similar.

4 Program extraction for Isabelle/HOL

So far, we have presented a generic framework for program extraction. We will now show how to instantiate it to a specific object logic, namely Isabelle/HOL.

4.1 Type extraction

First of all, we need to assign types to logical formulae of HOL, i.e. add new equations characterizing typeof.

$$\mathsf{typeof}\ (\mathsf{Trueprop}\ P) \equiv \mathsf{typeof}\ P$$

$$(\lambda x.\ \mathsf{typeof}\ (P\ x)) \equiv (\lambda x.\ \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null}))) \Longrightarrow \mathsf{typeof}\ (\exists x :: \alpha.\ P\ x) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\alpha))$$

$$(\lambda x.\ \mathsf{typeof}\ (P\ x)) \equiv (\lambda x.\ \mathsf{Type}\ (\mathsf{TYPE}(\tau))) \Longrightarrow \mathsf{typeof}\ (\exists x :: \alpha.\ P\ x) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\alpha \times \tau))$$

$$\mathsf{typeof}\ P \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \mathsf{typeof}\ (P \vee Q) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{sumbool}))$$

$$\mathsf{typeof}\ P \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Type}\ (\mathsf{TYPE}(\tau)) \Longrightarrow \mathsf{typeof}\ (P \vee Q) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\tau\ \mathsf{option}))$$

$$\mathsf{typeof}\ P \equiv \mathsf{Type}\ (\mathsf{TYPE}(\sigma)) \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \mathsf{typeof}(P \vee Q) \equiv \mathsf{Type}(\mathsf{TYPE}(\sigma\ \mathsf{option}))$$

$$\mathsf{typeof}\ P \equiv \mathsf{Type}\ (\mathsf{TYPE}(\sigma)) \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Type}\ (\mathsf{TYPE}(\tau)) \Longrightarrow \mathsf{typeof}\ (P \vee Q) \equiv \mathsf{Type}\ (\mathsf{TYPE}(\sigma + \tau))$$

$$\mathsf{typeof}\ A \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \qquad \text{if } A \text{ atomic, i.e. } A \in \{x = y, \mathsf{True}, \mathsf{False}, \ldots\}$$

We only show the equations for $\exists$ and $\vee$. The equations for $\wedge$ are quite similar and those for $\forall$ and $\longrightarrow$ look almost the same as their meta level counterparts introduced in 3.1. The first equation states that typeof can simply be pushed through the coercion function Trueprop. The computational content of $\exists x.\ P\ x$ is either a pair consisting of the witness and the computational content of $P\ x$, if there is one, otherwise it is just the witness. If both P and Q have a computational content, then the computational content of $P \vee Q$ is a disjoint sum

$$\textbf{datatype}\ (\alpha + \beta)\ =\ \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta$$

If just one of P and Q has a computational content, the result is of type

$$\textbf{datatype}\ \alpha\ \mathit{option}\ =\ \mathit{None} \mid \mathit{Some}\ \alpha$$

i.e. a program satisfying this specification will either return a proper value or signal an error. If neither P nor Q has a computational content, the result is just a boolean value, i.e. an element of type

$$\textbf{datatype}\ \mathit{sumbool}\ =\ \mathit{Left} \mid \mathit{Right}$$

4.2 Realizability

In order to reason about correctness of programs extracted from HOL proofs, we also need to add equations for realizes.

$$\mathsf{realizes}\ t\ (\mathsf{Trueprop}\ P) \equiv \mathsf{Trueprop}\ (\mathsf{realizes}\ t\ P)$$

$$(\lambda x.\ \mathsf{typeof}\ (P\ x)) \equiv (\lambda x.\ \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null}))) \Longrightarrow \mathsf{realizes}\ t\ (\exists x.\ P\ x)) \equiv \mathsf{realizes\ Null}\ (P\ t)$$

$$\mathsf{realizes}\ t\ (\exists x.\ P\ x) \equiv \mathsf{realizes}\ (\mathit{snd}\ t)\ (P\ (\mathit{fst}\ t))$$

$$\mathsf{typeof}\ P \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \mathsf{realizes}\ t\ (P \vee Q)) \equiv (\mathit{case}\ t\ \mathit{of}\ \mathit{Left} \Rightarrow \mathsf{realizes\ Null}\ P \mid \mathit{Right} \Rightarrow \mathsf{realizes\ Null}\ Q)$$

$$\mathsf{typeof}\ P \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \mathsf{realizes}\ t\ (P \vee Q) \equiv (\mathit{case}\ t\ \mathit{of}\ \mathit{None} \Rightarrow \mathsf{realizes\ Null}\ P \mid \mathit{Some}\ q \Rightarrow \mathsf{realizes}\ q\ Q)$$

$$\mathsf{typeof}\ Q \equiv \mathsf{Type}\ (\mathsf{TYPE}(\mathsf{Null})) \Longrightarrow \mathsf{realizes}\ t\ (P \vee Q) \equiv (\mathit{case}\ t\ \mathit{of}\ \mathit{None} \Rightarrow \mathsf{realizes\ Null}\ Q \mid \mathit{Some}\ p \Rightarrow \mathsf{realizes}\ p\ P)$$

$$\mathsf{realizes}\ t\ (P \vee Q) \equiv (\mathit{case}\ t\ \mathit{of}\ \mathit{Inl}\ p \Rightarrow \mathsf{realizes}\ p\ P \mid \mathit{Inr}\ q \Rightarrow \mathsf{realizes}\ q\ Q)$$

Again, the equations for $\wedge$ are similar and those for $\forall$ and $\longrightarrow$ look almost the same as their meta level counterparts from 3.3. For atomic predicates A, we set $\mathsf{realizes\ Null}\ A = A$. The above characterization of realizes can be applied to $\neg$ as follows: Let $\mathsf{typeof}\ P = \mathsf{Type}\ (\mathsf{TYPE}(\tau))$ and $\tau \neq \mathsf{Null}$. Then

$$\begin{array}{lll}
& \mathsf{realizes\ Null}\ (\neg P) & \\
= & \mathsf{realizes\ Null}\ (P \longrightarrow \mathsf{False}) & \{\text{definition of } \neg\} \\
= & \forall x :: \tau.\ \mathsf{realizes}\ x\ P \longrightarrow \mathsf{realizes\ Null\ False} & \{\text{definition of realizes}\} \\
= & \forall x :: \tau.\ \mathsf{realizes}\ x\ P \longrightarrow \mathsf{False} & \{\text{definition of realizes}\} \\
= & \forall x :: \tau.\ \neg\mathsf{realizes}\ x\ P & \{\text{definition of } \neg\}
\end{array}$$

name	V	$\mathcal{E}(name,\ V)$
impI	$\{P,\ Q\}$ $\{Q\}$	$\lambda P\ Q\ pq.\ pq$ $\lambda P\ Q\ q.\ q$
mp	$\{P,\ Q\}$ $\{Q\}$	$\lambda P\ Q\ pq.\ pq$ $\lambda P\ Q\ q.\ q$
allI	$\{P\}$	$\lambda P\ p.\ p$
spec	$\{P\}$	$\lambda P\ x\ p.\ p\ x$
exI	$\{P\}$ $\{\}$	$\lambda P\ x\ p.\ (x,\ p)$ $\lambda P\ x.\ x$
exE	$\{P,\ Q\}$ $\{Q\}$	$\lambda P\ Q\ p\ pq.\ pq\ (fst\ p)\ (snd\ p)$ $\lambda P\ Q\ x\ pq.\ pq\ x$
disjI1	$\{P,\ Q\}$ $\{P\}$ $\{Q\}$ $\{\}$	$\lambda P\ Q.\ Inl$ $\lambda P\ Q.\ Some$ $\lambda P\ Q.\ None$ $\lambda P\ Q.\ Left$
disjI2	$\{P,\ Q\}$ $\{P\}$ $\{Q\}$ $\{\}$	$\lambda Q\ P.\ Inr$ $\lambda Q\ P.\ None$ $\lambda Q\ P.\ Some$ $\lambda Q\ P.\ Right$
disjE	$\{P,\ Q,\ R\}$ $\{Q,\ R\}$ $\{P,\ R\}$ $\{R\}$	$\lambda P\ Q\ R\ pq\ pr\ qr.\ case\ pq\ of\ Inl\ p \Rightarrow pr\ p \mid Inr\ q \Rightarrow qr\ q$ $\lambda P\ Q\ R\ pq\ pr\ qr.\ case\ pq\ of\ None \Rightarrow pr \mid Some\ q \Rightarrow qr\ q$ $\lambda P\ Q\ R\ pq\ pr\ qr.\ case\ pq\ of\ None \Rightarrow qr \mid Some\ p \Rightarrow pr\ p$ $\lambda P\ Q\ R\ pq\ pr\ qr.\ case\ pq\ of\ Left \Rightarrow pr \mid Right \Rightarrow qr$
FalseE	$\{P\}$	$\lambda P.\ arbitrary$

Fig. 4. Realizers for basic inference rules of Isabelle/HOL

If $\tau = \mathsf{Null}$, then $\mathsf{realizes}\ \mathsf{Null}\ (\neg P)$ is simply $\neg\mathsf{realizes}\ \mathsf{Null}\ P$. Note that for P without computational content, we do not necessarily have $\mathsf{realizes}\ \mathsf{Null}\ P = P$. For example, $\mathsf{realizes}\ \mathsf{Null}\ (\neg(\exists x.\ x = c)) = \forall x.\ \neg x = c$.

4.3 Realizing terms

What remains to do is to specify how the functions $\mathcal{E}$ and $\mathcal{C}$ introduced in 3.2 and 3.3 act on theorems of Isabelle/HOL. This means that for each basic inference rule of the logic, we have to give a realizing term and a correctness proof. As before, we only treat some particularly interesting cases. Figure 4 shows the realizing terms corresponding to some of the inference rules of HOL. As mentioned in §3.1, there may be more than one realizer for each inference rule. When proving the correctness of a realizer corresponding to an inference rule with predicate variables, such as $P\ \bar{t}$, we face a problem. Since P is not known beforehand, we do not know what $\mathsf{realizes}\ r\ (P\ \bar{t})$ actually means. Therefore, we set $\mathsf{realizes}\ r\ (P\ \bar{t}) = P'\ r\ \bar{t}$. Later on, when an instantiation for P is known, we can substitute $\lambda r\ \bar{t}.\ \mathsf{realizes}\ r\ (P\ \bar{t})$ for P'.

For example, the correctness theorem $\mathsf{disjE\_correctness\_P\_Q}$

$$sum\text{-}case\ P\ Q\ x \Longrightarrow (\bigwedge p.\ P\ p \Longrightarrow R\ (f\ p)) \Longrightarrow (\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)) \Longrightarrow R\ (sum\text{-}case\ f\ g\ x)$$

corresponding to program $\mathcal{E}(\mathsf{disjE}, \{P, Q, R\})$ is shown by case analysis on x and by applying the above rules for *sum-case*. When applying this correctness theorem, we have to instantiate P, Q and R appropriately:

$\mathcal{C}(\mathsf{disjE}, \{P, Q, R\}) = \boldsymbol{\lambda} P\ Q\ R\ pq\ (h1\colon \text{-})\ pr\ (h2\colon \text{-})\ qr.$
$\quad \mathsf{disjE\text{-}correctness\text{-}P\text{-}Q} \cdot (\lambda p.\ \mathsf{realizes}\ p\ P) \cdot (\lambda q.\ \mathsf{realizes}\ q\ Q) \cdot pq \cdot (\lambda r.\ \mathsf{realizes}\ r\ R) \cdot$
$\quad pr \cdot qr \cdot h1 \cdot h2$

The correctness of programs $\mathcal{E}(\mathsf{disjI1}, \{P, Q\})$ and $\mathcal{E}(\mathsf{disjI2}, \{P, Q\})$ follows directly from the rewrite rules *sum-case f1 f2* (*Inl a*) = *f1 a* and *sum-case f1 f2* (*Inr b*) = *f2 b*.

The induction principle $P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)) \Longrightarrow P\ n$ for natural numbers is realized by $\lambda P\ n\ p0\ ps.$ *nat-rec p0 ps n*, where *nat-rec f1 f2 0* = *f1* and *nat-rec f1 f2* (*Suc nat*) = *f2 nat* (*nat-rec f1 f2 nat*). The corresponding correctness theorem is proved by induction on *nat*:

$P\ f1\ 0 \Longrightarrow$
$(\bigwedge nat\ rnat.\ P\ rnat\ nat \Longrightarrow P\ (f2\ nat\ rnat)\ (Suc\ nat)) \Longrightarrow P\ (nat\text{-}rec\ f1\ f2\ nat)\ nat$

5 Example: Warshall's algorithm

As a larger example, we show how Warshall's algorithm for computing the transitive closure of a relation can be derived using program extraction. The formalization is inspired by Berger et al. [7]. It has also been treated in the Coq system [3] by Paulin-Mohring [18]. In the sequel, a relation will be a function mapping two elements of a type to a boolean value.

datatype $b = T \mid F$
types $'a\ rel = 'a \Rightarrow 'a \Rightarrow b$

To emphasize that the relation has to be *decidable*, we use the datatype b instead of the built-in type *bool* of HOL for this purpose.

In order to write down the specification of the algorithm, it will be useful to introduce a function *is-path'*, where *is-path' r x ys z* holds iff there is a path from x to z with intermediate nodes *ys* with respect to a relation r.

consts *is-path'* :: $'a\ rel \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow bool$
primrec
$\quad$ *is-path' r x* [] $z = (r\ x\ z = T)$
$\quad$ *is-path' r x* $(y \#\ ys)\ z = (r\ x\ y = T \wedge$ *is-path' r y ys z*$)$

Paths will be modeled as triples consisting of a source node, a list of intermediate nodes and a target node. In the sequel, nodes will be natural numbers. Using the auxiliary function *is-path'* we can now define a function *is-path*, where *is-path r p i j k* holds iff p is a path from j to k with intermediate nodes less than i. For brevity, a path with this property will be called an i-path. We also introduce a function *conc* for concatenating two paths.

constdefs
is-path :: *nat rel* ⇒ (*nat* × *nat list* × *nat*) ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*
is-path r p i j k ≡ *fst p* = *j* ∧ *snd* (*snd p*) = *k* ∧
list-all (λ*x*. *x* < *i*) (*fst* (*snd p*)) ∧ *is-path*′ *r* (*fst p*) (*fst* (*snd p*)) (*snd* (*snd p*))
conc :: (′*a* × ′*a list* × ′*a*) ⇒ (′*a* × ′*a list* × ′*a*) ⇒ (′*a* × ′*a list* × ′*a*)
conc p q ≡ (*fst p*, *fst* (*snd p*) @ *fst q* # *fst* (*snd q*), *snd* (*snd q*))

The main proof relies on several lemmas about properties of *is-path*. For example, if p is an i-path from j to k, then p is also a *Suc* i-path.

lemma *lemma1*: ⋀*p*. *is-path r p i j k* ⟹ *is-path r p* (*Suc i*) *j k*

If p is a *0*-path from j to k, then relation r has an edge connecting j and k.

lemma *lemma2*: ⋀*p*. *is-path r p 0 j k* ⟹ *r j k* = *T*

If p is an i-path from j to i, and q is an i-path from i to k, then concatenating these paths yields a *Suc* i-path from j to k.

lemma *lemma3*: ⋀*p q*. *is-path r p i j i* ⟹ *is-path r q i i k* ⟹
is-path r (*conc p q*) (*Suc i*) *j k*

The last lemma is central to the proof of the main theorem. It says that if there is a *Suc* i-path from j to k, but no i-path, then there must be i-paths from j to i and from i to k.

lemma *lemma4*: ⋀*p*. *is-path r p* (*Suc i*) *j k* ⟹ ¬ *is-path r p i j k* ⟹
(∃ *q*. *is-path r q i j i*) ∧ (∃ *q*. *is-path r q i i k*)

The first component of the conjunction can be proved by induction on the list of intermediate nodes of path p. The proof of the second component is symmetric to the proof of the first component, using "reverse induction". Although this lemma can be proved constructively, its computational content is not used in the main theorem. To emphasize this, we rephrase it, writing ¬ (∀ *x*. ¬ *P x*) instead of ∃ *x*. *P x*.

lemma *lemma4*′: ⋀*p*. *is-path r p* (*Suc i*) *j k* ⟹ ¬ *is-path r p i j k* ⟹
¬ (∀ *q*. ¬ *is-path r q i j i*) ∧ ¬ (∀ *q*. ¬ *is-path r q i i k*)

The main theorem can now be stated as follows: For a given relation r, for all i and for every two nodes j and k there either exists an i-path p from j to k, or no such path exists. Of course, this would be trivial to prove classically. However, a constructive proof of this statement actually yields a function that either returns *Some p* if there is a path or returns *None* otherwise.

The proof is by induction on i. In the base case, we have to find a *0*-path from j to k, which can only exist if r has an edge connecting these two nodes. Otherwise there can be no such path according to *lemma2*. In the step case, we are supposed to find a *Suc* i-path from j to k. By appeal to the induction hypothesis, we can decide if we already have an i-path from j to k. If this is the case, we can easily conclude by *lemma1* that this is also a *Suc* i-path. Otherwise, by appealing to the induction hypothesis two more times, we check whether we

have i-paths from j to i and from i to k. If there are such paths, we combine them to get a *Suc* i-path from j to k by *lemma3*. Otherwise, if there is no i-path from j to i or from i to k, there can be no *Suc* i-path from j to k either, because this would contradict *lemma4′*. In order to formalize the above proof in Isabelle in a readable way, we make use of the proof language *Isar* due to Wenzel [23].

```
theorem warshall: ⋀j k. ¬ (∃ p. is-path r p i j k) ∨ (∃ p. is-path r p i j k)
proof (induct i)
  case (0 j k) show ?case — induction basis
  proof (cases r j k)
    assume r j k = T
    hence is-path r (j, [], k) 0 j k by (simp add: is-path-def)
    hence ∃ p. is-path r p 0 j k .. thus ?thesis ..
  next
    assume r j k = F hence r j k ≠ T by simp
    hence ¬ (∃ p. is-path r p 0 j k) by (rules dest: lemma2) thus ?thesis ..
  qed
next
  case (Suc i j k) thus ?case — induction step
  proof
    assume ∃ p. is-path r p i j k
    hence ∃ p. is-path r p (Suc i) j k by (rules intro: lemma1) thus ?case ..
  next
    assume h1: ¬ (∃ p. is-path r p i j k)
    from Suc show ?case
    proof
      assume ¬ (∃ p. is-path r p i j i)
      with h1 have ¬ (∃ p. is-path r p (Suc i) j k) by (rules dest: lemma4′)
      thus ?case ..
    next
      assume ∃ p. is-path r p i j i
      then obtain p where h2: is-path r p i j i ..
      from Suc show ?case
      proof
        assume ¬ (∃ p. is-path r p i i k)
        with h1 have ¬ (∃ p. is-path r p (Suc i) j k) by (rules dest: lemma4′)
        thus ?case ..
      next
        assume ∃ q. is-path r q i i k
        then obtain q where is-path r q i i k ..
        with h2 have is-path r (conc p q) (Suc i) j k by (rule lemma3)
        hence ∃ pq. is-path r pq (Suc i) j k .. thus ?case ..
      qed
    qed
  qed
qed
```

From the above proof, the following program is extracted by Isabelle:

```
warshall ≡
λr i j k.
  nat-rec (λi j. case r i j of T ⇒ Some (i, [], j) | F ⇒ None)
   (λk H i j.
      case H i j of
      None ⇒
        case H i k of None ⇒ None
        | Some p ⇒ case H k j of None ⇒ None | Some q ⇒ Some (conc p q)
      | Some q ⇒ Some q)
   i j k
```

Applying the definition of realizability presented in §4 yields the following correctness theorem, which is automatically derived from the above proof:

$$case\ warshall\ r\ i\ j\ k\ of\ None \Rightarrow \forall x.\ \neg\ is\text{-}path\ r\ x\ i\ j\ k \mid Some\ q \Rightarrow is\text{-}path\ r\ q\ i\ j\ k$$

6 Related work

The first theorem provers to support program extraction were Constable's Nuprl system [11], which is based on Martin-Löf type theory, and the PX system by Hayashi [14]. The Coq system [3], which is based on the Calculus of Inductive Constructions (CIC), can extract programs to OCaml [19] and Haskell. Paulin-Mohring [18, 17] has given a realizability interpretation for the Calculus of Constructions and proved the correctness of extracted programs with respect to this realizability interpretation. Although it would be possible in principle to check the correctness proof corresponding to an extracted program inside Coq itself, this has not been implemented yet. Moreover, it is not completely obvious how to do this in practice, because Coq allows for the omission of termination arguments (e.g. wellordering types) in the extracted program, which may render the program untypable in CIC due to the occurrence of unguarded fixpoints. Instead of distinguishing between *relevant* and *irrelevant* predicate variables as described in §3, the Coq system has two universes Set and Prop, which are inhabited by computationally interesting and computationally noninteresting types, respectively. Recently, Fernández, Severi and Szasz [13, 21] have proposed an extension of the Calculus of Constructions called the *Theory of Specifications*, which internalizes program extraction and realizability. The built-in reduction relation of this calculus reflects the behaviour of the functions corr and extr defined in §3. A similar approach is taken in Burstall and McKinna's theory of *deliverables* [16]. A deliverable is a pair consisting of a program together with its correctness proof, which is modeled using strong Σ types. Anderson [1] describes the embedding of a first order logic with program extraction in Elf and proves several meta-theoretic properties of the extraction function, e.g. well-typedness of the extracted program. The Minlog system [5] by Schwichtenberg can extract Scheme programs from proofs in minimal first order logic, enriched with

inductive datatypes and predicates. It has recently been extended to produce correctness proofs for extracted programs as well. Moreover, it also supports program extraction from classical proofs [6]. Isabelle has already been used for implementing program extraction calculi in the past, too. Basin and Ayari [2] have shown how to simulate Manna and Waldinger's "Deductive Tableau" in Isabelle/HOL. Coen [10] formalized his own "Classical Computational Logic", which is tailored specifically towards program extraction, whereas our framework is applicable to common object logics such as HOL.

7 Conclusion

We have developed a program extraction framework for the theorem prover Isabelle and have shown its applicability to realistic examples. In the future, we would like to tackle some more advanced case studies. A good candidate seem to be algorithms from graph theory. For example, one could think of extracting a graph colouring algorithm from the proof of the *Five Colour Theorem* by Bauer and Nipkow [4].

Another important point to study is how our framework can be instantiated to other logics, such as constructive versions of Zermelo-Fränkel Set Theory (ZF). For the HOL instantiation described in §4, matters were particularly simple, since HOL and Isabelle's meta logic share the same type system. This is in contrast to ZF, which is essentially untyped and simulates the concept of type checking by explicit logical reasoning about set membership statements.

Finally, it would be interesting to examine how work on program extraction from *classical* proofs, e.g. along the lines of Berger et al. [6], can be applied to classical proofs in HOL.

Acknowledgement

Laura Crosilla, Tobias Nipkow, Martin Strecker and Markus Wenzel commented on a draft version and suggested improvements. I would also like to thank Helmut Schwichtenberg and Monika Seisenberger for numerous discussions on the subject of this paper.

References

[1] P. Anderson. Program extraction in a logical framework setting. In F. Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, volume 822 of *LNAI*, pages 144–158. Springer-Verlag, July 1994.

[2] A. Ayari and D. Basin. A higher-order interpretation of deductive tableau. *Journal of Symbolic Computation*, 31(5):487–520, May 2001.

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq proof assistant reference manual – version 7.2. Technical Report 0255, INRIA, February 2002.

[4] G. Bauer and T. Nipkow. The 5 colour theorem in Isabelle/Isar. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *LNCS*, pages 67–82. Springer-Verlag, 2002.

[5] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II: Systems and Implementation Techniques of *Applied Logic Series*, pages 41–71. Kluwer Academic Publishers, Dordrecht, 1998.

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

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

[8] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2000*, volume 1869 of *LNCS*. Springer-Verlag, 2000.

[9] S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs: TYPES'2000*, volume 2277 of *LNCS*. Springer-Verlag, 2002.

[10] M. D. Coen. *Interactive program derivation*. PhD thesis, Cambridge University, November 1992.

[11] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[12] T. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, January 1985.

[13] M. Fernández and P. Severi. An operational approach to program extraction in the Calculus of Constructions. In *International Workshop on Logic Based Program Development and Transformation (LOPSTR'02)*, LNCS. Springer, 2002.

[14] S. Hayashi and H. Nakano. *PX, a Computational Logic*. Foundations of Computing. MIT Press, 1988.

[15] S. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[16] J. McKinna and R. M. Burstall. Deliverables: A categorical approach to program development in type theory. In A. M. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium*, volume 711 of *lncs*, pages 32–67, Gdansk, Poland, 30 Aug.– 3 Sept. 1993. Springer.

[17] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, Jan. 1989. ACM.

[18] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, Jan. 1989.

[19] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[20] F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In H. Ganzinger, editor, *CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, 1999.

[21] P. Severi and N. Szasz. Studies of a theory of specifications with built-in program extraction. *Journal of Automated Reasoning*, 27 (1):61–87, July 2001.

[22] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: TPHOLs'97*, LNCS 1275, 1997.

[23] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, TU München, 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.