What Can we Gain by Integrating a Language Processor with a Theorem Prover?

Paul J. Voda

Institute of Informatics, Comenius University Bratislava,
Mlynská dolina, 842 15 Bratislava, Slovakia,
email: voda@fmph.uniba.sk
homepage: www.fmph.uniba.sk/∼voda

1 Introduction

The goal of this paper is to investigate the impact on the design of a programming language of tight integration of a language processor with a theorem prover (intelligent proof assistant). What improvements in syntax, semantics, and computation do we gain by this?

We assume as self-evident the obvious gain which is quite substantial. The language obtains a sound semantics by its interpretation into a formal logical theory. By proving theorems about our programs we can prove them correct. From the many language systems with theorem provers we mention just two: PVS and Isabelle/HOL [6, 5]. Both of them are more general theorem provers than programming languages. On the other hand, our system CL (Clausal Language), has been designed as a programming language from the start. It is not a language which can be used for industrial applications (yet, we hope), but neither is it a toy language. It has been used in our undergraduate teaching for seven years now. About four hundred students yearly actively use it in the four courses based on CL [2]. While, the formal basis for PVS and Isabelle is high order logic with typed functionals, CL is based on the simplest non-trivial formal theory: Peano Arithmetic (PA).

The less obvious gain, and the one which is the subject of this paper, is the surprising amount of flexibility and extensibility one obtains by coupling the definition of syntax, (denotational) semantics, and computations (operational semantics) with an integrated theorem prover. As every computer programmer knows, the syntax of a programming language must be readable, comfortable to use, and flexible even if the functions computed are just primitive recursive ones. The programmer's day is filled with the incredibly detailed design of his functions and increasingly in the future with the proofs of their correctness. Both tasks are extremely hard. The task of a programming language designer is to make it slightly more comfortable. This should be contrasted with the view of a logician. To him the five syntactic rules for the construction of primitive recursive functions just about exhaust the subject of syntax. Proofs of properties of concrete functions are mostly a trivial matter to him. Unless of course, he happens to sit in a car which crashed because of a bug in the program controlling the brakes.

It has been a folklore in the programming circles for many years that the only constructors of programs are case analysis and recursion. C. McBride and J. McKinna [1] discuss an independently conceived approach whose power is similar to ours. We think that our approach is simpler not only because PA is typeless but also because our programs stay entirely within the language of PA whereas they work in the type theory and introduce new syntactic constructs with the semantics given by translation.

Case analysis and recursion are like straightedge and compass and our task is to see how far our constructions with the two can go. We thus take a formal theory (PA in our case, but any reasonably rich theory preferred by the reader will do) and use the provability within the theory to admit a wide set of case analysis and recursion constructs. In PA the basic case analysis is the test $x = 0$ versus $x > 0$ and the basic tool for recursion is complete induction: from $\forall y{<}x\, P(y)$ infer $P(x)$. Everything else, and for comfortable programming we need a virtually unlimited supply of those, is provably reducible to the two.

The existing language CL has a fixed syntax for its definitions given by clauses which are a subset of formulas of PA. Also the pattern matching capabilities of CL, - although far more extensive than those of, say, Haskell - are fixed. This paper discusses extensions of CL to make both syntax and pattern matching extensible in an almost unlimited way although the resulting clauses remain just the formulas of PA and no new syntax is needed. The basic idea is to make the legality of the constructs subject to a formal proof in PA.

We reduce the clausal definitions of programs (i.e. functions) in CL to extensions by definitions of PA and so the defined functions automatically obtain denotations in the unique expansion of the standard model of PA. We also present not only effective but also efficient method of computing the functions from their definitions.

The outline of the paper is follows. Section 2 discussed the basic facts about PA. Section 3 describes (and defines) clausal definitions through a refinement process starting from a wholly undefined clause. We give examples of discriminations (corresponding to the case constructs from programming languages) and of clausal definitions in Sect. 4. Section 5 shows that the clauses of (legal) clausal definitions are theorems in extensions by definitions of PA. Section 6 discusses efficient evaluation of clausal definitions.

The author would like to acknowledge the great contribution of Ján Komara who came up with the suggestion of extensibility of syntax and pattern matching during our long discussions about the future of CL.

2 Peano Arithmetic

Because we expect our readers to be computer scientists we are slightly more detailed in the following outline of Peano Arithmetic than if this were addressed to logicians who are familiar with PA.

2.1 The language, axioms, and standard model of PA. The first-order language of PA consists of the symbols 0, S (successor), $+$, and $\cdot$ with the usual interpretation in the *standard model* $\mathcal{M}$ *of PA* with the domain $\mathbb{N}$ of natural numbers.

We use $\boldsymbol{A}$, $\boldsymbol{B}$, …and $\boldsymbol{s}$, $\boldsymbol{t}$, …as metavariables ranging over the formulas and terms of PA respectively. We write $\forall\overline{x}\boldsymbol{A}$ for $\forall x_1 \ldots \forall x_n \boldsymbol{A}$ where we do not exclude $n = 0$ in which case the first formula abbreviates $\boldsymbol{A}$. We use similar abbreviations for $\exists$.

The set of *closed* terms, i.e. without variables, generated from 0 by $S(\boldsymbol{s})$ consists of (monadic) *numerals*. We abbreviate by $\underline{n}$ the numeral denoting in $\mathcal{M}$ the number n. We also use the decimal numerals 0, 1, …as the obvious abbreviations.

PA is axiomatized by the six axioms:

$$
\begin{aligned}
S(x) &\neq 0 & \qquad S(x) = S(y) &\rightarrow x = y \\
0 + y &= y & S(x) + y &= S(x+y) \\
0 \cdot y &= 0 & S(x) \cdot y &= x \cdot y + y \;.
\end{aligned}
$$

plus the *induction axioms*

$$\boldsymbol{A}[0] \land \forall x(\boldsymbol{A}[x] \rightarrow \boldsymbol{A}[S(x)]) \rightarrow \boldsymbol{A}[x] \tag{1}$$

for each formula $\boldsymbol{A}[x]$ with x the *induction variable* and with possibly other variables free as *parameters*.

2.2 Extensions of PA. Programming in PA means a continual extension of PA with new function and predicate symbols together with new axioms concerning the symbols. When we say PA we mean the current extension of PA. When discussing concrete extensions we will designate by PA the theory before the extension and by PA$'$ the theory after the extension. The extensions will be *conservative*, i.e. if for a formula $\boldsymbol{A}$ of PA we have PA$' \vdash \boldsymbol{A}$ then already PA $\vdash \boldsymbol{A}$.

2.3 Extensions by definitions. *Extension by definition* (see [8]) of PA with a new n-ary function symbol f ($n \geq 0$) adds the *defining axiom*:

$$f(\overline{x}) = y \leftrightarrow \boldsymbol{A}[\overline{x}, y] \tag{1}$$

where the formula $\boldsymbol{A}[\overline{x}, y]$ with at most the indicated variables free is in the language of PA (so it does not apply f) and PA proves the *existence* and *uniqueness* conditions:

$$\text{PA} \vdash \exists y \boldsymbol{A}[\overline{x}, y] \tag{2}$$

$$\text{PA} \vdash \boldsymbol{A}[\overline{x}, y] \land \boldsymbol{A}[\overline{x}, z] \rightarrow y = z \;. \tag{3}$$

The extension is conservative and induction axioms containing f are provable in PA$'$ from the defining axiom as theorems.

We will abbreviate by $\boldsymbol{U}_{x_1,\ldots,x_n}\boldsymbol{B}[x_1,\ldots,x_n]$ the formula

$$\forall\overline{x}\forall\overline{y}(\boldsymbol{B}[x_1,\ldots,x_n]\wedge\boldsymbol{B}[y_1,\ldots,y_n]\rightarrow x_1=y_1\wedge\cdots\wedge x_n=y_n)$$

for new variables $\overline{y}$ which asserts the uniqueness of $\overline{x}$. (3) can thus be written as $\boldsymbol{U}_y\boldsymbol{A}[\overline{x},y]$. We also write $\exists!y\boldsymbol{A}[\overline{x},y]$ for $\exists y\boldsymbol{A}[\overline{x},y]\wedge\boldsymbol{U}_y\boldsymbol{A}[\overline{x},y]$.

It is well-known that PA$'$ proves the equivalence of (1) and $\boldsymbol{A}[\overline{x},f(\overline{x})]$. Thus the last formula can be alternatively viewed as the defining axiom for f. It is perhaps less known that (1) is also equivalent in PA$'$ to:

$$f(\overline{x})=y\leftarrow\boldsymbol{A}[\overline{x},y] \tag{4}$$

where by convention originating in the programming language Prolog we write the formula with inverted implication. The theory of clausal definitions is based on this equivalence.

The only non-trivial part of the proof in PA$'$ of the equivalence is that $\rightarrow$ of (1) follows from (4). We assume $f(\overline{x})=y$ and get $\boldsymbol{A}[\overline{x},z]$ for some z from (2). Then $f(\overline{x})=z$ by (4) and so $\boldsymbol{A}[\overline{x},y]$.

Explicit definitions $f(\overline{x})=\boldsymbol{s}[\overline{x}]$ are a special case of extensions by definitions: $f(\overline{x})=y\leftrightarrow\boldsymbol{s}[\overline{x}]=y$.

One can also extend PA by definitions with new predicate symbol P and the defining axiom $P(\overline{x})\leftrightarrow\boldsymbol{A}[\overline{x}]$ where $\boldsymbol{A}$ contains at most the indicated variables free and does not apply P. We will often use a special form with the defining axiom:

$$\mathrm{PA}'\vdash P(\overline{x})\leftrightarrow f(\overline{x})>0$$

where f is such that PA $\vdash f(\overline{x})\leq 1$ and it is called the *characteristic* function of P.

A property of an extension by definition is that the standard model of PA is uniquely *expanded* to a model for the new symbol satisfying its defining axiom.

2.4 Provably recursive functions. A special place among extensions by definitions 2.2(4) have those where the formula $\boldsymbol{A}$ is Σ_1 in PA (the definition follows). We then call the interpretation of the function symbol f in the standard model $\mathcal{M}$ a *provably recursive* function.

We call the quantifiers $\exists x{\leq}\boldsymbol{s}\,\boldsymbol{C}$ and $\forall x{\leq}\boldsymbol{s}\,\boldsymbol{C}$ *bounded* and treat them as abbreviations for the formulas $\exists x\exists y(x+y=\boldsymbol{s}\wedge\boldsymbol{C})$ and $\forall x\forall y(x+y=\boldsymbol{s}\rightarrow\boldsymbol{C})$ respectively. Her y is a new variable. The reader will note that when PA contains $\leq$ then the last two formulas are equivalent in PA to $\exists x(x\leq\boldsymbol{s}\wedge\boldsymbol{C})$ and $\forall x(x\leq\boldsymbol{s}\rightarrow\boldsymbol{C})$ respectively.

Σ_0-formulas are built up from the atomic formulas of the original language (see Par. 2.1) by propositional connectives and bounded quantification. Formulas of the form $\exists\overline{x}\boldsymbol{A}$ with $\boldsymbol{A}$ Σ_0 are *Σ_1-formulas.* The formula $\boldsymbol{A}$ is Σ_1 *in PA* if there is a Σ_1-formula $\boldsymbol{B}$ such that PA $\vdash\boldsymbol{A}\leftrightarrow\boldsymbol{B}$.

The reader will note that the formula $\boldsymbol{A}$ from Par. 2.3 may apply provably recursive functions and predicates with provably recursive characteristic functions. This is because

$$\mathrm{PA}'\vdash\boldsymbol{C}[f(\overline{\boldsymbol{s}})]\leftrightarrow\exists y(\boldsymbol{A}[\overline{\boldsymbol{s}},y]\wedge\boldsymbol{C}[y])$$

for f defined by 2.2(4) and so we can systematically eliminate all functions and predicates introduced by extensions by definition. This means that formulas $\exists \overline{x} \boldsymbol{A}$ with $\boldsymbol{A}$ like a Σ_0-formula except that it applies provably recursive functions are Σ_1 in PA.

2.5 Extensions with new function (predicate) symbols without defining axioms. Extensions designated by PA$[+f]$ (PA$[+P]$), extend (the current extension of) PA with a new n-ary function symbol f (predicate symbol P) ($n \geq 0$) with new induction axioms for formulas applying the new symbol but no axioms describing the new symbol. This is clearly conservative because we can replace in a proof in PA$'$ all applications of f (P) with 0 ($\top$) and obtain a proof in PA (after an adjustment for the identity axioms for the new symbol which are turned by the replacement into theorems).

Proofs of theorems $\boldsymbol{A}$ in PA$[+f]$ (PA$[+P]$) will serve as *schemes*. By replacing in such a proof of $\boldsymbol{A}$ all applications of f (P) by corresponding applications of a function (predicate) symbol in the language of PA we obtain a proof in PA of $\boldsymbol{A'}$ obtained from $\boldsymbol{A}$ by the same replacements.

The function symbol f in the above formula $\boldsymbol{A}$ can be replaced by a function symbol with additional arguments, called, parameters, but the parameters must be variables not occurring in $\boldsymbol{A}$ and must stay fixed in all replaced applications. The same holds for the replacement of predicate symbols P. A typical case is when we replace applications of the constant (nullary) predicate symbol P by applications $Q(\overline{x})$. We can even replace a predicate symbol by a formula $\boldsymbol{A}[\overline{x}]$ without introducing first the predicate symbol Q by $Q(\overline{x}) \leftrightarrow \boldsymbol{A}[\overline{x}]$.

Extensions PA$[+f]$ (PA$[+P]$) will play important role in the proofs of admissibility of clausal definitions (see Sect. 5). We plan to use a subsystem of second order theory of arithmetic in the next version of CL. In such a system the second order quantification over function and predicate variables takes over the role played by the extensions PA$[+f]$ (PA$[+P]$) (see [13]).

2.6 Course of values recursion. The hardest part of the bootstrap of PA is the introduction of some kind of coding of finite sequences. This is needed before one can show that PA is closed under *course of values recursion*, i.e. that there is an extension by definition of PA with f such that

$$\begin{aligned}
&\mathrm{PA}' \vdash f(0, \overline{y}) = \boldsymbol{s}[\overline{y}] \\
&\mathrm{PA}' \vdash f(x+1, \overline{y}) = \boldsymbol{t}[x, \overline{y}, f(\boldsymbol{r_1}, \overline{y}), \ldots, f(\boldsymbol{r_k}, \overline{y})]
\end{aligned}$$

for terms $\boldsymbol{s}[\overline{y}]$, $\boldsymbol{t}[x, \overline{y}, r_1, \ldots, r_k]$, $\boldsymbol{r_1}[\overline{x}, \overline{y}]$, $\ldots$, $\boldsymbol{r_k}[\overline{x}, \overline{y}]$ of PA with the free variables among the indicated ones. We require that PA proves $\boldsymbol{r_1} < x+1$, $\ldots$, $\boldsymbol{r_k} < x+1$.

For a traditional bootstrap via Gödel's β function see [8]. A modern direct development is in [12].

If the above terms apply only provably recursive functions then also f is provably recursive.

2.7 Pairing function. We wish to extend PA with a two-place *pairing* function designated by x, y and satisfying:

$$\mathrm{PA}' \vdash x, y = v, w \to x = v \wedge y = w \tag{1}$$

$$\mathrm{PA}' \vdash x < x, y \wedge y < x, y \tag{2}$$

$$\mathrm{PA}' \vdash x = 0 \vee \exists v \exists w\, x = v, w\ . \tag{3}$$

(1) is the usual *pairing property*, property (2) is needed so we can do induction over pairs. We also get from it that 0 is an *atom*, i.e. $0 \neq x, y$, because otherwise we would get $x < 0$. That 0 is the only atom follows from (3).

We can easily see that every every natural number is then uniquely denoted (in $\mathcal{M}$) by a *pair numeral* which is a term formed from 0 by pairing $(\boldsymbol{s}, \boldsymbol{t})$. If we are interested in the characterization of small classes of computational complexity (such as PTIME) we should require that the number of symbols needed to write down the pair numeral denoting x be on the order of $\log(x)$.

It is well-known that the Cantor's diagonal function $J(x, y)$ (see [3]) satisfies the pairing property but, since $0, 0 = 0$, there are no atoms. If we increase the function by one, i.e. if we extend PA with the *modified Cantor's function* $x, y = J(x, y) + 1$, by:

$$\mathrm{PA}' \vdash x, y = (x + y) \cdot (x + y + 1) \div 2 + x + 1\ , \tag{4}$$

then PA' proves the above three pairing properties. The function is provably recursive.

It was shown in [10] that the modified Cantor's function fails the size requirement. A suitable pairing function was introduced there. The function is used in CL and it is discussed at length in [11]. However, the modified Cantor's function can be used as long as we are interested in all provably recursive functions. Since this is the case in this paper, we fix the pairing function (4).

2.8 Pairing conventions. We let pairing group to the right where x, y, z abbreviates $x, (y, z)$. We assign to the pairing function the lowest precedence. Thus, for instance, $x, y+z$ stands for $x, (y+z)$. We adopt a pairing convention such that whenever in the arguments of applications the infix pairing comma in (x, y) cannot be mistaken for a comma separating arguments we may drop the parentheses around pairing. Thus, for instance, if f is a binary function then $f((v, w), x, y)$ abbreviates $f((v, w), (x, y))$. Notice that the first set of parentheses cannot be omitted.

2.9 Coding of finite sequences. Any pairing function satisfying the three pairing properties from Par. 2.7 establishes a one-to-one correspondence between finite sequences over $\mathbb{N}$ and $\mathbb{N}$. The empty sequence is coded by 0 and the sequence $a_1 \ldots a_n$ by $a_1, \ldots, a_n, 0$. Notice the use of 0 to recognize the end of the code. This is not possible without atoms. Codes of finite sequences are called *lists* in computer programming where they were introduced in the late nineteen fifties by J. McCarthy in his path-breaking programming language LISP. Our representation in $\mathbb{N}$ of lists reduces without loss of power the denumerable infinitude of LISP's atoms to the single one: 0. As a consequence every natural number is a unique list.

2.10 Provably well-founded relations. We say that the relation $x \prec y$ is *provably well-founded* if the *$\prec$-induction* is provable as follows:

$$\mathrm{PA}[+P] \vdash \forall x(\forall y(y \prec x \rightarrow P(y)) \rightarrow P(x)) \rightarrow P(x)\ . \tag{1}$$

Two examples of well-founded relations are $x \prec y \leftrightarrow x+1 = y$ and $x \prec y \leftrightarrow x < y$. The reader can easily convince himself that (1) follows in the first case from the induction axiom 2.1(1) where $\boldsymbol{A}[x] \equiv P(x)$ and in the second case (1) (with $\prec$ replaced by $<$) is a scheme of *complete* induction which follows from 2.1(1) with $\boldsymbol{A}[x] \equiv \forall y{\leq}x\, P(y)$.

For any formula $\boldsymbol{A}[x, \overline{z}]$ of PA with *parameters* $\overline{z}$ we can obtain an *instance* of $\prec$-induction from the proof of the scheme (1):

$$\mathrm{PA} \vdash \forall x(\forall y(y \prec x \rightarrow \boldsymbol{A}[y, \overline{z}]) \rightarrow \boldsymbol{A}[x, \overline{z}]) \rightarrow \boldsymbol{A}[x, \overline{z}]\ .$$

2.11 Intended use of provably well-founded relations in recursive definitions. We will use bellow provably well-founded relations $\prec$ for the uniform handling of arguments $\overline{x}$ of a function $f(\overline{x})$ defined by recursion. The recursion will be such that the recursive applications $f(\overline{\boldsymbol{s}})$ will *decrease* in the well-founded relation: $\overline{\boldsymbol{s}} \prec \overline{x}$. Note that this stands for $\boldsymbol{s_1}, \ldots, \boldsymbol{s_n} \prec x_1, \ldots, x_n$ with the commas designating the pairing function. We do not exclude *nested* recursion where the terms $\overline{\boldsymbol{s}}$ contain recursive applications of f.

The above is the most general case of recursion. Most of the time for a function $f(\overline{x}, \overline{v}, \overline{z})$ only the *recursion* variables $\overline{x}$ (not necessarily the first ones) will decrease in a provably well-founded relation $\prec_1$, the parameters $\overline{v}$ (possibly none) will be *substituted* for, and the parameters $\overline{z}$ (possibly none) will not change. Thus every recursive application will be of a form application $f(\boldsymbol{t}, \overline{\boldsymbol{s}}, \overline{z})$ where $\boldsymbol{t} \prec_1 \boldsymbol{x}$.

Consider the relation:

$$a \prec b \leftrightarrow \exists\overline{x}\exists\overline{y}\exists\overline{v}\exists\overline{w}\exists\overline{z}(a = \overline{y}, \overline{w}, \overline{z} \land x = \overline{x}, \overline{v}, \overline{z} \land \overline{y} \prec_1 \overline{x})\ . \tag{1}$$

For the recursive calls $f(\boldsymbol{t}, \overline{\boldsymbol{s}}, \overline{z})$ we now have $\boldsymbol{t}, \overline{\boldsymbol{s}}, \overline{z} \prec \overline{x}, \overline{v}, \overline{z}$ and so the recursion decreases in $\prec$. It remains to show that $\prec$ is provably well-founded. For that we take a new unary predicate symbol P and we wish to derive:

$$\mathrm{PA}[+P] \vdash \forall b(\forall a(a \prec b \rightarrow P(a)) \rightarrow P(b)) \rightarrow P(b)\ . \tag{2}$$

Because $\prec_1$ is provably well-founded we have:

$$\mathrm{PA}[+P] \vdash \forall\overline{x}(\forall\overline{y}(\overline{y} \prec_1 \overline{x} \rightarrow \forall\overline{v}P(\overline{y}, \overline{v}, \overline{z})) \rightarrow \forall\overline{v}P(\overline{x}, \overline{v}, \overline{z})) \rightarrow P(\overline{x}, \overline{v}, \overline{z})\ . \tag{3}$$

Designating the antecedent of (3) by $\boldsymbol{A_1}$ and the antecedent of (2) by $\boldsymbol{A}$ we first show $\mathrm{PA}[+P] \vdash \boldsymbol{A} \rightarrow \boldsymbol{A_1}$ which we designate by $\star_1$. We work in $\mathrm{PA}[+P]$

where we take any $\overline{z}$, assume $\boldsymbol{A}$, take any $\overline{x}$, assume $\forall\overline{y}(\overline{y} \prec_1 \overline{x} \to \forall\overline{v}P(\overline{y},\overline{v},\overline{z}))$ which we designate by $\star_2$, and take any $\overline{v}$. We wish to show $P(\overline{x},\overline{v},\overline{z})$. For that it suffices to show $\forall a(a \prec \overline{x},\overline{v},\overline{z} \to P(a))$ and use $\boldsymbol{A}$ with $b := \overline{x},\overline{v},\overline{z}$. In order to prove the last formula we take any a and assume $a \prec \overline{x},\overline{v},\overline{z}$. Then $a = \overline{y},\overline{w},\overline{z}$ for some $\overline{y}$, $\overline{w}$ s.t. $\overline{y} \prec_1 \overline{x}$ by (1), and we use $\star_2$ to obtain $P(a)$.

With $\star_1$ proved we now show (2) by assuming $\boldsymbol{A}$, taking any b, and by considering two cases. If $b \neq \overline{x},\overline{v},\overline{z}$ for all $\overline{x}$, $\overline{v}$, $\overline{z}$ then $\forall a(a \prec b \to P(a))$ because for no a we have $a \prec b$. Thus $P(b)$ follows from $\boldsymbol{A}$. If $b = \overline{x},\overline{v},\overline{z}$ for some $\overline{x}$, $\overline{v}$, $\overline{z}$ then, since $\boldsymbol{A_1}$ from $\star_1$, we obtain $P(b)$ from (3).

3 Clausal Definitions

In this section we define clausal definitions as finite collections of formulas of PA whose construction must be certified by proofs in PA. Clausal definitions are constructed by discrimination (case analysis) and recursion.

3.1 Provable discriminations. The basic building blocks of clausal definitions are *provable discriminations* for which we use the notation:

$$\textbf{discr}_{\overline{x}} \quad \boldsymbol{A}_1[\overline{x},\overline{y_1}] \mid \cdots \mid \boldsymbol{A}_j[\overline{x},\overline{y_j}] \mid \cdots \mid \boldsymbol{A}_p[\overline{x},\overline{y_p}] \qquad \leftarrow \boldsymbol{D}[\overline{x}]$$

where $p \geq 1$ and for $j = 1,\ldots,p$ the *discriminator* formula $\boldsymbol{A}_j[\overline{x},\overline{y_j}]$ is a formula with free variables *output* variables $\overline{y_j} \equiv y_1,\ldots,y_{k_j}$ and possibly also some of the *input* variables $\overline{x} \equiv x_1,\ldots,x_n$. We do not exclude $\overline{x}$ and some of $\overline{y_j}$ to be empty (i.e. $n = 0$ and/or $k_j = 0$). The formula $\boldsymbol{D}[\overline{x}]$ with free variables among the input ones is called the *condition* of the discrimination. Most often the condition will be $\top$ when the discrimination is *unconditional* and we usually omit it from the discrimination notation.

The above notation stands for $(p+1)$ theorems of PA. The first theorem is the *discrimination* condition:

$$\mathrm{PA} \vdash \boldsymbol{D}[\overline{x}] \to \exists! j(\underline{1} \leq j \leq \underline{p} \wedge (j{=}\underline{1} \wedge \exists\overline{y_1}\boldsymbol{A}_1[\overline{x},\overline{y_1}] \vee \cdots \vee j{=}\underline{p} \wedge \exists\overline{y_p}\boldsymbol{A}_p[\overline{x},\overline{y_p}]))$$

which says that for any input values $\overline{x}$ satisfying the condition exactly one discriminator holds for some values of its output variables. For every $j = 1,\ldots,p$ the remaining theorems express the *uniqueness* conditions:

$$\mathrm{PA} \vdash \boldsymbol{D}[\overline{x}] \to \boldsymbol{U}_{\overline{y_j}}\boldsymbol{A}_j[\overline{x},\overline{y_j}]$$

asserting that the local variables are uniquely determined if they exist.

The reader will note that the discrimination notation acts as a scheme because for the terms $\overline{\boldsymbol{t}}[\overline{z}]$ whose number is the same as that of $\overline{x}$ the following is also a provable discrimination with input variables $\overline{z}$:

$$\textbf{discr}_{\overline{z}} \quad \boldsymbol{A}_1[\overline{\boldsymbol{t}}[\overline{z}],\overline{y_1}] \mid \boldsymbol{A}_j[\overline{\boldsymbol{t}}[\overline{z}],\overline{y_j}] \mid \cdots \mid \boldsymbol{A}_p[\overline{\boldsymbol{t}}[\overline{z}],\overline{y_p}] \qquad \leftarrow \boldsymbol{D}[\overline{\boldsymbol{t}}[\overline{z}]]$$

We assume that $\overline{z}$ are pairwise disjoint from the output variables.

We can have a provable discriminator in an extension PA$[+\boldsymbol{p}]$ where $\boldsymbol{p}$ is a function or a predicate symbol and some of the discriminator formulas apply the symbol. It should be clear that we can systematically eliminate the symbol $\boldsymbol{p}$ as described in Par. 2.5 and then the new discrimination is provable in PA. See paragraphs 4.1 and 6.8 for examples.

3.2 Clausal definitions. A *clausal definition* of the function symbol $f(\overline{x})$ consists of *clauses* of the following form:

$$\begin{array}{ll}
f(\overline{x}) = \boldsymbol{s}_1[\overline{x},\overline{y_1}] \;\leftarrow \boldsymbol{B}_1[\overline{x},\overline{y_1}] & \leftarrow \boldsymbol{C}[\overline{x}] \\
\quad\ldots & \\
f(\overline{x}) = \boldsymbol{s}_i[\overline{x},\overline{y_i}] \;\leftarrow \boldsymbol{B}_i[\overline{x},\overline{y_i}] & \leftarrow \boldsymbol{C}[\overline{x}] \\
\quad\ldots & \\
f(\overline{x}) = \boldsymbol{s}_k[\overline{x},\overline{y_k}] \;\leftarrow \boldsymbol{B}_k[\overline{x},\overline{y_k}] & \leftarrow \boldsymbol{C}[\overline{x}]
\end{array}$$

The clausal definition of the predicate symbol $P(\overline{x})$ has the form:

$$\begin{array}{ll}
\hat{P}(\overline{x}) \leftarrow \boldsymbol{B}_1[\overline{x},\overline{y_1}] & \leftarrow \boldsymbol{C}[\overline{x}] \\
\quad\ldots & \\
\hat{P}(\overline{x}) \leftarrow \boldsymbol{B}_i[\overline{x},\overline{y_i}] & \leftarrow \boldsymbol{C}[\overline{x}] \\
\quad\ldots & \\
\hat{P}(\overline{x}) \leftarrow \boldsymbol{B}_k[\overline{x},\overline{y_k}] & \leftarrow \boldsymbol{C}[\overline{x}]
\end{array}$$

where $\hat{P}(\overline{\boldsymbol{a}})$ is $P(\overline{\boldsymbol{a}})$ or $\neg P(\overline{\boldsymbol{a}})$. We designate by $\boldsymbol{p}$ the symbol being defined, i.e. either f or P.

The optional formulas $\boldsymbol{C}[\overline{x}]$ in antecedents of clausal definitions are their *conditions* and they may not apply $\boldsymbol{p}$. If $\boldsymbol{C}$ is $\top$, which is usually the case, the clausal definitions are *unconditional* and the conditions are not given. We do not exclude the case when $\overline{x}$ is empty and the clausal definition defines a function or a predicate constant $\boldsymbol{p}$.

The formulas in the consequents of clauses are called the *heads* of the clauses and the formulas $\boldsymbol{B}_i[\overline{x},\overline{y_i}]$ are *bodies*. The variables $\overline{y_i}$, which may be empty, are called *local* variables. We do not exclude the case of empty bodies when there is no $\boldsymbol{B}_i$ and then the clause should be viewed as an abbreviation for $f(\overline{x}) = \boldsymbol{s}$ or $\hat{P}(\overline{x})$ depending on the type of the definition. We say that a function of a predicate symbol other than $\boldsymbol{p}$ is *used* in a clausal definition if it has an application in the head or body of some clause (applications in the condition $\boldsymbol{C}$ do not count as uses).

When the clausal definition is constructed by at least one recursion refinement (see Par. 3.5) then such a definition is *recursive* and a provably well-founded relation $\prec$ must be supplied in which recursive applications of $\boldsymbol{p}$ occurring in the bodies will descend. This means that the arity of $\boldsymbol{p}$ must be positive.

We wish the clauses of a clausal definition to be theorems of a suitable extension by definitions of PA. In the following we will perform various transformations on finite sets of clauses preseving their equivalence. We say that two sets of clauses are *provably equivalent* if PA $\vdash \boldsymbol{A}_1 \leftrightarrow \boldsymbol{A}_2$ where $\boldsymbol{A}_1$ is the conjunction of universal closures of clauses of one set of clauses and $\boldsymbol{A}_2$ is similarly formed from the clauses of the second set. The transformations will be often performed before the symbol $\boldsymbol{p}$ has been introduced into PA. In such a case the provable equivalence means $\mathrm{PA}[+\boldsymbol{p}] \vdash \boldsymbol{A}_1 \leftrightarrow \boldsymbol{A}_2$.

The clauses of a clausal definition must be *discriminated* which in the standard model means that for any $\overline{x}$ such that $\boldsymbol{C}[\overline{x}]$ holds exactly one of $\exists\overline{y_i}\boldsymbol{B}_i[\overline{x},\overline{y_i}]$ most hold. This is syntactically expressed as follows:

$$\mathrm{PA} \vdash \boldsymbol{C}[\overline{x}] \rightarrow \exists! i(\underline{1} \leq i \leq \underline{k} \wedge (i{=}\underline{1} \wedge \exists\overline{y_1}\boldsymbol{B}_1[\overline{x},\overline{y_1}] \vee \cdots \vee i{=}\underline{k} \wedge \exists\overline{y_k}\boldsymbol{B}_k[\overline{x},\overline{y_k}])) \ .$$

The local variables in every body of a clausal definition must satisfy the *uniqueness condition*:

$$\mathrm{PA} \vdash \boldsymbol{C}[\overline{x}] \rightarrow \boldsymbol{U}_{\overline{y_i}}\boldsymbol{B}_i[\overline{x},\overline{y_i}] \ .$$

3.3 Top down construction of clausal definitions. The construction of a clausal definition of a symbol $\boldsymbol{p}$ starts with an *initial* partially constructed clause:

$$\cdots \leftarrow \cdots \qquad\qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

and proceeds by a finite number of steps, called *refinements*.

At a certain stage during a construction we have a set of partially constructed clauses:

$$\begin{array}{ll}
\cdots \leftarrow \boldsymbol{B}_1[\overline{x},\overline{y_1}] \ \cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\quad \cdots & \\
\cdots \leftarrow \boldsymbol{B}_i[\overline{x},\overline{y_i}] \ \cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\quad \cdots & \\
\cdots \leftarrow \boldsymbol{B}_k[\overline{x},\overline{y_k}] \ \ldots & \leftarrow \boldsymbol{C}[\overline{x}]
\end{array}$$

which satisfy the discrimination and uniqueness conditions as in Par. 3.2. We let the initial partially constructed clause to satisfy both conditions trivially. There are three forms of refinements described in the three following paragraphs. Each refinement takes a set of partially constructed clauses satisfying the two conditions into a new set satisfying the conditions again.

3.4 Discrimination refinements. The first refinement step is a *discrimination* which when applied to the i-th partially constructed clause from Par. 3.3 is specified by taking a provable discrimination

$$\mathbf{discr}_{\overline{x},\overline{y_i}} \quad \boldsymbol{A}_1[\overline{x},\overline{y_i},\overline{z_1}] \mid \cdots \mid \boldsymbol{A}_j[\overline{x},\overline{y_i},\overline{z_j}] \mid \cdots \mid \boldsymbol{A}_k[\overline{x},\overline{y_i},\overline{z_k}] \qquad \leftarrow \boldsymbol{D}[\overline{x},\overline{y_i}]$$

with input variables $\overline{x},\overline{y_i}$ and output variables $\overline{z_j}$. The discriminators may not apply the symbol $\boldsymbol{p}$ and the following *discrimination* condition must hold:

$$\mathrm{PA}[+\boldsymbol{p}] \vdash \boldsymbol{C}[\overline{x}] \wedge \boldsymbol{B}_i[\overline{x},\overline{y_i}] \rightarrow \boldsymbol{D}[\overline{x},\overline{y_i}] \ .$$

Note that although the discriminators may not apply the symbol $\boldsymbol{p}$, the formula $\boldsymbol{B}_i[\overline{x},\overline{y_i}]$ may.

The new set of partially constructed clauses is:

$$\begin{array}{ll}
\cdots \leftarrow \boldsymbol{B}_1[\overline{x},\overline{y_1}] \;\cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\;\;\cdots & \\
\cdots \leftarrow \boldsymbol{B}_i[\overline{x},\overline{y_i}] \wedge \boldsymbol{A}_1[\overline{x},\overline{y_i},\overline{z_1}] \;\cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\;\;\cdots & \\
\cdots \leftarrow \boldsymbol{B}_i[\overline{x},\overline{y_i}] \wedge \boldsymbol{A}_j[\overline{x},\overline{y_i},\overline{z_j}] \;\cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\;\;\cdots & \\
\cdots \leftarrow \boldsymbol{B}_i[\overline{x},\overline{y_i}] \wedge \boldsymbol{A}_p[\overline{x},\overline{y_i},\overline{z_p}] \;\cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\;\;\cdots & \\
\cdots \leftarrow \boldsymbol{B}_k[\overline{x},\overline{y_k}] \;\cdots & \leftarrow \boldsymbol{C}[\overline{x}]
\end{array}$$

3.5 Recursion refinements. The second refinement step is a *recursion* refinement which when applied to the i-th partially constructed clause from Par. 3.3 is specified by taking a recursive application of the symbol $\boldsymbol{p}$ being defined which is $f(\overline{\boldsymbol{t}}[\overline{x},\overline{y_i}]) = y$ (with y a new local variable) or $P(\overline{\boldsymbol{t}}[\overline{x},\overline{y_i}])$. The terms $\overline{\boldsymbol{t}}$ may not apply the symbol $\boldsymbol{p}$ (if it is f).

Recursion refinement with the above recursive applications is legal only when the arity of $\boldsymbol{p}$ is positive and the conditions of *regularity* and *applicability* hold. The conditions are in that order as follows:

$$\begin{array}{l}
\mathrm{PA}[+\boldsymbol{p}] \vdash \boldsymbol{C}[\overline{x}] \wedge \boldsymbol{B}_i[\overline{x},\overline{y_i}] \rightarrow \overline{\boldsymbol{t}}[\overline{x},\overline{y_i}] \prec \overline{x} \\
\mathrm{PA}[+\boldsymbol{p}] \vdash \boldsymbol{C}[\overline{x}] \wedge \boldsymbol{B}_i[\overline{x},\overline{y_i}] \rightarrow \boldsymbol{C}[\overline{\boldsymbol{t}}[\overline{x},\overline{y_i}]] \ .
\end{array}$$

When $\boldsymbol{p} \equiv f$ then the new set of partially constructed clauses is

$$\begin{array}{ll}
\cdots \leftarrow \boldsymbol{B}_1[\overline{x},\overline{y_1}] \;\cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\;\;\cdots & \\
\cdots \leftarrow \boldsymbol{B}_i[\overline{x},\overline{y_i}] \wedge f(\overline{\boldsymbol{t}}[\overline{x},\overline{y_i}]) = y \;\cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\;\;\cdots & \\
\cdots \leftarrow \boldsymbol{B}_k[\overline{x},\overline{y_k}] \;\ldots & \leftarrow \boldsymbol{C}[\overline{x}]
\end{array}$$

When $\boldsymbol{p} \equiv P$ then the new set is:

$$\begin{array}{ll}
\cdots \leftarrow \boldsymbol{B}_1[\overline{x},\overline{y_1}] \;\cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\;\;\cdots & \\
\cdots \leftarrow \boldsymbol{B}_i[\overline{x},\overline{y_i}] \wedge \;\;\; P(\overline{\boldsymbol{t}}[\overline{x},\overline{y_i}]) \;\cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\cdots \leftarrow \boldsymbol{B}_i[\overline{x},\overline{y_i}] \wedge \neg P(\overline{\boldsymbol{t}}[\overline{x},\overline{y_i}]) \;\cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
\;\;\cdots & \\
\cdots \leftarrow \boldsymbol{B}_k[\overline{x},\overline{y_k}] \;\ldots & \leftarrow \boldsymbol{C}[\overline{x}]
\end{array}$$

3.6 Value refinements. The third refinement step is a *value* refinement when we finish the construction of a clause. Value refinement applied to the i-th partially constructed clause from Par. 3.3 is specified by constructing its head $\boldsymbol{H}$ which is $f(\overline{x}) = \boldsymbol{t}[\overline{x},\overline{y_i}]$ or $\hat{P}(\overline{x})$ depending on whether $\boldsymbol{p}$ is a function or predicate symbol. The term $\boldsymbol{t}$ may not contain recursive applications of $\boldsymbol{p}$ (when $\boldsymbol{p}$ is a function).

The new set of clauses is:

$$
\begin{array}{llr}
\cdots & \leftarrow \boldsymbol{B}_1[\overline{x},\overline{y_1}] \ \cdots & \leftarrow \boldsymbol{C}[\overline{x}] \\
 & \cdots & \\
\boldsymbol{H} & \leftarrow \boldsymbol{B}_i[\overline{x},\overline{y_i}] & \leftarrow \boldsymbol{C}[\overline{x}] \\
 & \cdots & \\
\cdots & \leftarrow \boldsymbol{B}_k[\overline{x},\overline{y_k}] \ \ldots & \leftarrow \boldsymbol{C}[\overline{x}]
\end{array}
$$

3.7 Explicit and course of values clausal definitions. A clausal definition of a symbol $\boldsymbol{p}$ is *explicit* if it is not recursive, i.e. there are no recursion refinements in the definition of $\boldsymbol{p}$.

A recursive clausal definition of $\boldsymbol{p}(\overline{u},x,\overline{v})$ is a *course of values* definition with the *recursion variable* x and possibly empty *parameters* $\overline{u}$, $\overline{v}$ when x, descends in the provably well-founded relation $<$ and the parameters do not change. This means that the recursive applications in the recursion refinements are $\boldsymbol{p}(\overline{u},\boldsymbol{s},\overline{v})$ with the regularity $\boldsymbol{s} < x$ provable under the assumption $\boldsymbol{C} \wedge \boldsymbol{B}_i$. We trust the reader with bringing the regularity condition into the form of Par. 3.5 by defining $\prec$ as in Par. 2.11.

We will in Thm. 5.4 reduce such definitions into course of values recursion from Par. 2.6. For that we need an additional condition that the recursion is not *nested* if the above clausal definition defines a function symbol f. Unnested definitions of functions are such that no local variable occurring in an argument of a recursive application *depends on recursion*.

A local variable w occurring free in a body of a clausal definition of $f(\overline{x})$ *depends on recursion* when the body contains a recursive application $f(\overline{\boldsymbol{s}}) = w$ or else w is among the output variables $\overline{z}$ of a discrimininator $\boldsymbol{A}[\overline{x},\overline{y},\overline{z}]$ and some of the input local variables $\overline{y}$ depend on recursion.

4 Examples of Clausal Definitions

We will now illustrate the discrimination and clausal definitions discussed in the preceding section with some examples of explicit and course of values clausal definitions. We hope that the reader will find the examples more or less self-evident because the justification of clausal definitions is the subject treated only in the following section.

4.1 Discrimination on negation. For any predicate constant P the theory PA[$+P$] proves the following scheme of unconditional *negation* discrimination:

discr $P \mid \neg P$

An example of an unconditional explicit clausal definition using negation discrimination is the *signum* function $sg(x)$:

$$
\begin{aligned}
&Sg(x) = 0 \leftarrow x = 0\\
&Sg(x) = 1 \leftarrow x \neq 0 \ .
\end{aligned}
$$

4.2 Discrimination on zero. A variation of discrimination on negation is provable *discrimination on zero*

$$\mathbf{discr}_x \quad x = 0 \mid x > 0$$

The signum function can be presented with discrimination on zero as follows:

$$\begin{aligned} &Sg(x) = 0 \leftarrow x = 0 \\ &Sg(x) = 1 \leftarrow x > 0 \ . \end{aligned}$$

4.3 Dichotomy and trichotomy discriminations. The *dichotomy* and *trichotomy* are in that order the following unconditional provable discriminations:

$$\begin{aligned} &\mathbf{discr}_{x,y} \quad x \leq y \mid x > y \\ &\mathbf{discr}_{x,y} \quad x < y \mid x = y \mid x > y \end{aligned}$$

The two argument *maximum* function has an explicit clausal definition by dichotomy discrimination:

$$\begin{aligned} &\max(x, y) = y \leftarrow x \leq y \\ &\max(x, y) = x \leftarrow x > y \quad . \end{aligned}$$

4.4 Monadic discrimination. Unconditional *monadic* discrimination is the first example of a provable discrimination with an output variable:

$$\mathbf{discr}_x \quad x = 0 \mid x = y + 1$$

The one-place *exponentiation* function 2^z has the following course of values clausal definition using a monadic discrimination followed by a recursion refinement in the second clause:

$$\begin{aligned} &2^z = 1 \quad\ \leftarrow z = 0 \\ &2^z = 2 \cdot y \leftarrow z = x + 1 \land 2^x = y \ . \end{aligned}$$

The regularity condition for the recursive application in the second clause is

$$\text{PA} \vdash z = x + 1 \rightarrow x < z \ .$$

The above definition is given in an *unabbreviated* form. It can be brought into a more readable form by an operation which provably equivalently simplifies the clauses:

$$\begin{aligned} &2^0 = 1 \\ &2^{x+1} = 2 \cdot 2^x \ . \end{aligned}$$

The simplification uses twice instances of the provable equivalence $\forall z(z = \boldsymbol{s} \rightarrow \boldsymbol{A}[z]) \leftrightarrow \boldsymbol{A}[\boldsymbol{s}]$ which holds when the term $\boldsymbol{s}$ does not contain the variable z.

4.5 Binary discrimination. The functions $S_0(x) = 2{\cdot}x$ and $S_1 = 2{\cdot}x = 1$ are called *binary successor* functions. We write them in postfix as $x\mathbf{0}$ and $x\mathbf{1}$ respectively.

We will need the *binary numerals* which are the least class of terms containing 0 and with every term $\boldsymbol{s}$ other than 0 also the terms $\boldsymbol{s}\mathbf{0}$ and $\boldsymbol{s}\mathbf{1}$. The reader will note that in order to escape the problem of leading zeros, the term $0\mathbf{0}$ is not a binary numeral. We designate by $\underline{i}_b$ the binary numeral denoting the number i in the standard model $\mathcal{M}$.

The provable *binary* discrimination is unconditional:

$$\mathbf{discr}_x \quad x = y\mathbf{0} \mid x = y\mathbf{1}$$

The *predecessor* function Pr has a course of values clausal definition with a binary discrimination as follows:

$$\begin{array}{ll} Pr(z) = 0 & \leftarrow z = x\mathbf{0} \land x = 0 \\ Pr(z) = y\mathbf{1} & \leftarrow z = x\mathbf{0} \land x > 0 \land Pr(x) = y \\ Pr(z) = x\mathbf{0} & \leftarrow z = x\mathbf{1}\ . \end{array}$$

The case $z = x\mathbf{0}$ is further discriminated on zero. The regularity condition $z = x\mathbf{0} \land x > 0 \to x < z$ is easily provable in PA. The same definition can be presented in an abbreviated form as follows:

$$\begin{array}{l} Pr(0\mathbf{0}) = 0 \\ Pr(x\mathbf{0}) = Pr(x)\mathbf{1} \leftarrow x > 0 \\ Pr(x\mathbf{1}) = x\mathbf{0}\ . \end{array}$$

4.6 Dyadic discrimination. The unconditional *dyadic* discrimination is provable from simple properties of divisibility by two:

$$\mathbf{discr}_x \quad x = 0 \mid x = 2{\cdot}q + r \land 1 \le r \land r \le 2$$

It is well-known that every positive number y can be presented in the *dyadic representation* as $\sum_{i<n} d_i{\cdot}2^i$ for a unique *dyadic length* $n = |y|_d$ and unique *dyadic digits* $d_0, \ldots d_{n-1}$ which are either 1 or 2. The two place *dyadic concatenation* function $x \star y = x{\cdot}2^{|y|_d} + y$ has the following clausal definition by course of values on the second argument which is dyadically discriminated:

$$\begin{array}{ll} x \star z = x & \leftarrow z = 0 \\ x \star z = u & \leftarrow z = 2{\cdot}y + d \land 1 \le d \land d \le 2 \land x \star y = u\ . \end{array}$$

The definition can be abbreviated as follows:

$$\begin{array}{l} x \star 0 = x \\ x \star (2{\cdot}y + d) = 2{\cdot}(x \star y) + d \leftarrow 1 \le d \land d \le 2\ . \end{array}$$

4.7 Pair discrimination. The pairing properties from Par. 2.7 prove the following unconditional *pair* discrimination:

$$\mathbf{discr}_x \quad x = 0 \mid x = v, w$$

The two-place *list concatenation* function $x \oplus y$ has a course of values clausal definition on the first argument which is discriminated on pairs:

$$\begin{array}{ll} u \oplus y = y & \leftarrow u = 0 \\ u \oplus y = v, z_1 & \leftarrow u = v, w \wedge w \oplus y = z_1 \quad . \end{array}$$

The same definition is in an abbreviated form as follows:

$$\begin{array}{l} 0 \oplus y = y \\ (v, w) \oplus y = v, w \oplus y \quad . \end{array}$$

The unary *projection* functions H (head) and T (tail) have explicit clausal definitions by pair discrimination:

$$\begin{array}{l} H(0) = 0 \\ H(x, y) = x \end{array}$$

$$\begin{array}{l} T(0) = 0 \\ T(x, y) = y \quad . \end{array}$$

The i-th element of the list x is accessed by the *list indexing* function $(x)_i$ which has an explicit clausal definition:

$$(x)_i = H\,T^i(x)$$

where the binary function $T^i(x)$ is defined by primitive recursion (a special case of course of values recursion) on i:

$$\begin{array}{l} T^0(x) = x \\ T^{i+1}(x) = T^i(x) \ . \end{array}$$

A direct clausal definition of the indexing function:

$$\begin{array}{l} (0)_i = 0 \\ (x, y)_0 = x \\ (x, y)_{i+1} = (y)_i \end{array}$$

is defined by pair and monadic discrimination. The recursion descends in both arguments but the clausal definition is not by course of values recursion because both arguments change in the recursion.

4.8 Let discrimination. Seemingly trivial is the following unnconditional *let* discrimination with one alternative:

$$\mathbf{discr}_x \quad x = y$$

Let discrimination is provable because $\exists y\, x = y$ is. The discrimination is so called because it plays the role of *let* constructs used in functional programming to prevent multiple computations. For instance, the function $f(x) = g(x+2)^2$ can be defined by a clausal definition:

$$f(x) = y{\cdot}y \leftarrow g(x+2) = y \ .$$

Computation with this definition saves the two-fold evaluation of $g(x+2)$ (see Sect. 6).

4.9 Example of a conditional clausal definition. The canonical example of a conditional definition is the definition of *integer* division $x \div y$, say, by repeated *modified subtraction* $x \dot{-} y$ which yields z such that $x = y + z$ or 0 otherwise. In order to save the repeated test for $y > 0$ we stipulate it as a condition to the definition:

$$\begin{array}{lll} x \div y = z + 1 & \leftarrow y \leq x \wedge (x \dot{-} y) \div y = z & \qquad \leftarrow y > 0 \\ x \div y = 0 & \leftarrow y > x & \qquad \leftarrow y > 0 \end{array}$$

This is a course of values clausal definition on x. Its regularity condition $y > 0 \wedge y \leq x \rightarrow x \dot{-} y < x$ is easily provable and so is the applicability condition $y > 0 \wedge y \leq x \rightarrow y > 0$. The same definition is abbreviated as follows:

$$\begin{array}{lll} x \div y = (x \dot{-} y) \div y + 1 & \leftarrow y \leq x & \qquad \leftarrow y > 0 \\ x \div y = 0 & \leftarrow y > x & \qquad \leftarrow y > 0 \end{array}$$

4.10 Arithmetization of binary trees. We wish to arithmetize (encode) the binary trees using the approach known from functional programming languages when one defines binary trees (having natural numbers as labels) by the following data-type definition:

$$Bt = 0 \mid Nd(N, Bt, Bt)$$

Toward that end we introduce the the three-place *constructor* $Nd(n,l,r) = n,l,r$ and use the pairing properties in the proof of the following unconditional discrimination with three alternatives:

$$\mathbf{discr}_a \quad a = 0 \mid a = n,0 \mid a = Nd(n,l,r)$$

Pascal-like data-types in CL are unary predicates. The predicate Bt has the following course of values clausal definition using the above discrimination:

$$\begin{array}{ll} Bt(a) & \leftarrow a = 0 \\ Bt(a) & \leftarrow a = Nd(n,l,r) \wedge Bt(l) \wedge Bt(r) \\ \neg Bt(a) & \leftarrow a = Nd(n,l,r) \wedge Bt(l) \wedge \neg Bt(r) \\ \neg Bt(a) & \leftarrow a = Nd(n,l,r) \wedge \neg Bt(l) \\ \neg Bt(a) & \leftarrow a = n,0 \end{array}$$

When defining a predicate it is customary to omit the *default* clauses which are the clauses with negative heads. When defining a function f the default clauses are the ones with zero values $f(\overline{x}) = 0$. With abbreviations and with defaults omitted we obtain the following clausal definition:

$$Bt(0)$$
$$Bt(Nd(n,l,t)) \leftarrow Bt(l) \wedge Bt(r)\ .$$

The following is the first example of a conditional discrimination on *binary trees*:

$$\mathbf{discr}_a \quad a = 0 \mid a = Nd(n,l,r) \qquad\qquad \leftarrow Bt(a)$$

The disrimination is provable because if for an a s.t. $Bt(a)$ we had the second alternative of the above three-alternative discrimination, i.e. if $a = n, 0$ for some n then we would get a contradiction from the fifth clause for Bt.

4.11 Membership in binary search trees. The number a is a *binary search tree* if $Bt(a)$ and the label of each of its node is greater than all labels to its left and less than all labels to its right. We trust the reader with the (not quite trivial) arithmetization of this definition in PA by a predicate $Bst(a)$. The main advantage of (balanced) binary search trees is that their labels can be looked up in time $\mathcal{O}(\log(n))$ where n is the number of labels in the tree whereas binary trees need time $\mathcal{O}(n)$.

The following is the course of values clausal definition on a of the two-place predicate $n \in a$ searching binary search trees. We make the definition conditional on $Bt(a)$:

$$n \in a \leftarrow a = Nd(m,l,r) \wedge n < m \wedge n \in l \qquad \leftarrow Bt(a)$$
$$n \in x \leftarrow a = Nd(m,l,r) \wedge n > m \wedge n \in r \qquad \leftarrow Bt(a)$$
$$n \in x \leftarrow a = Nd(m,l,r) \wedge n = m \qquad \leftarrow Bt(x)$$
$$n \notin a \leftarrow a = 0 \qquad \leftarrow Bt(a)$$

Because of the condition $Bt(a)$ we can use the discrimination on binary trees. Had we made the definition unconditional we would have to use the slightly slower to evaluate three-alternative discrimination from Par. 4.10.

We have to establish the applicability conditions whose consequents are $Bt(l)$ and $Bt(r)$ for the two recursive applications $n \in l$ and $n \in r$ respectively. The assumption of the first condition is $Bt(a) \wedge a = Nd(m,l,r) \wedge n < m$ from which can prove $Bt(l)$, It is instructive to sketch the proof which is not trivial (but can be performed without human intervention). For any a (exactly) one of the bodies of the definition of Bt holds for the uniquely determined values of local variables. With $a = Nd(m,l,r)$ it must be the second, third, or the fourth body. If it were not the case that $Bt(l)$ we would get a contradiction $\neg Bt(a)$ from the fourth clause. We similarly prove $Bt(r)$.

When we omit the default and substitute for a we obtain the following clausal definition:

$$n \in Nd(m,l,r) \leftarrow n < m \wedge n \in l \qquad \leftarrow Bt(Nd(m,l,r))$$
$$n \in Nd(m,l,r) \leftarrow n > m \wedge n \in r \qquad \leftarrow Bt(Nd(m,l,r))$$
$$n \in Nd(m,l,r) \leftarrow n = m \qquad \leftarrow Bt(Nd(m,l,r))$$

4.12 Least number discrimination. Conditional clausal definitions naturally call for conditional discrimination. The following are examples of provable conditional single alternative discriminations:

$$
\begin{array}{lll}
\mathbf{discr}_x & P(x,y) & \leftarrow \exists! y P(x,y) \\
\mathbf{discr}_x & P(x,y) \wedge \forall z (P(x,z) \rightarrow y \leq z) & \leftarrow \exists y P(x,y)
\end{array}
$$

The condition of existence and uniqueness of y s.t. $P(x,y)$ implies the discrimination and uniqueness conditions for the first discrimination. Mere assumption of existence proves by the *least number principle* ,which is reducible to complete induction (see [4]), the discrimination and uniqueness conditions for the second discrimination.

If PA proves $\exists! y P(x,y)$ unconditionally then the condition of the first discriminator is satisfied in any context. Similarly for the second discriminator.

Above discriminations will be used in clausal definitions constructed in the proofs of the theorems 5.8 and 6.14 and also in Par. 6.10.

5 Admissibility of Clausal Definitions

This section defines the semantics of the new extensible CL by showing PA closed under clausal definitions. We are here concerned with the questions of definability rather than computability which is the subject of the following section.

5.1 Reduction of predicates to functions. From every clausal definition of a predicate P we can construct a clausal definition of its characteristic function P_* by replacing the heads of the clauses $P(\overline{x})$ and $\neg P(\overline{x})$ by $P_*(\overline{x}) = 1$ and $P_*(\overline{x}) = 0$ respectively as well as the recursive applications $P(\overline{t})$ and $\neg P(\overline{t})$ by $P_*(\overline{t}) = y \wedge y > 0$ and $P_*(\overline{t}) = y \wedge y = 0$ respectively. Here y is a new variable. Note that the last replacement changes every recursion refinement of P to a recursion refinement of P_* followed by a zero discrimination.

It should be clear that if we manage to extend PA with the function symbol P_* to satisfy its clausal definition (i.e. to make its clauses theorems in PA$'$) we can explicitly define $P(\overline{x}) \leftrightarrow P_*(\overline{x}) > 0$ and then PA$''$ proves the clauses of the clausal definition of P.

Since the values of recursive applications 'vanish' in discriminations on zero, a course of values definition of a predicate is turned by the above process into a course of values definition of its characteristic function.

It is thus sufficient to investigate below properties of clausal definitions of functions and obtain corresponding properties of clausal definitions of predicates P via clausal definitions of their characteristic functions P_*.

5.2 Folding. A given clausal definition of a function f is constructed in steps numbered from 1 through s where $1 < s$ (at least one value refinement must be performed). We will now associate to every step $m = 1, \ldots, s$ a set of clauses, called the *m-th fold* of the definition and two formulas called the *m-th discrimination* formula $\boldsymbol{D}_m$ and its *governing* formula $\boldsymbol{G}_m$. This is done by induction on the reverse order of the construction in a *folding* process. We start the folding with a choice of a variable not occurring in the definition, say u, and by agreeing to call the $(s+1)$-th fold of the definition the clausal definition itself.

We then take any $m = 1 \ldots, s$ and assume that for all q such that $s \geq q > m$ we have defined the q-th fold, the discrimination formula $\boldsymbol{D}_q$, and its governing formula $\boldsymbol{G}_q$. We consider three cases corresponding to the kind of construction performed at the step m whose partially constructed clauses are as in Par. 3.3.

When a value refinement was performed at the step m as described in Par. 3.6 then the i-th clause of the $(m+1)$-th fold has the form:

$$f(\overline{x}) = \boldsymbol{t}[\overline{x}, \overline{y_i}] \leftarrow \boldsymbol{B}_i[\overline{x}, \overline{y_i}] \qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

We define the m-th discrimination formula

$$\boldsymbol{D}_m[\overline{x}, \overline{y_i}; u] \equiv \boldsymbol{t}[\overline{x}, \overline{y_i}] = u \tag{1}$$

and construct the m-th fold to be like the $(m+1)$-th fold except that its i-th clause is replaced by

$$f(\overline{x}) = u \leftarrow \boldsymbol{G}_m[\overline{x}, \overline{y_i}] \wedge \boldsymbol{D}_m[\overline{x}, \overline{y_i}; u] \qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

where $\boldsymbol{G}_m \equiv \boldsymbol{B}_i$ unless the last formula is empty and then $\boldsymbol{G}_m \equiv \top$.

When a discrimination refinement was performed at the step m as described in Par. 3.4 then the clauses i through $i+p-1$ of the $(m+1)$-th fold have the form:

$$\begin{array}{ll}
f(\overline{x}) = u \leftarrow \boldsymbol{G}_{q_1}[\overline{x}, \overline{y_i}, \overline{z_1}] \wedge \boldsymbol{D}_{q_1}[\overline{x}, \overline{y_i}, \overline{z_1}; u] & \leftarrow \boldsymbol{C}[\overline{x}] \\
\quad \cdots & \\
f(\overline{x}) = u \leftarrow \boldsymbol{G}q_j[\overline{x}, \overline{y_i}, \overline{z_j}] \wedge \boldsymbol{D}_{q_j}[\overline{x}, \overline{y_i}, \overline{z_j}; u] & \leftarrow \boldsymbol{C}[\overline{x}] \\
\quad \cdots & \\
f(\overline{x}) = u \leftarrow \boldsymbol{G}_{q_p}[\overline{x}, \overline{y_i}, \overline{z_p}] \wedge \boldsymbol{D}_{q_p}[\overline{x}, \overline{y_i}, \overline{z_p}; u] & \leftarrow \boldsymbol{C}[\overline{x}]
\end{array}$$

for some $q_1, \ldots, q_p > m$. We define the m-th discrimination formula as follows:

$$\begin{aligned}
\boldsymbol{D}_m[\overline{x}, \overline{y_i}; u] \equiv\ & \exists \overline{z_1}(\boldsymbol{A}_1[\overline{x}, \overline{y_i}, \overline{z_1}] \wedge \boldsymbol{D}_{q_1}[\overline{x}, \overline{y_i}, \overline{z_1}; u]) \vee \\
& \quad \cdots \vee \\
& \exists \overline{z_j}(\boldsymbol{A}_j[\overline{x}, \overline{y_i}, \overline{z_j}] \wedge \boldsymbol{D}_{q_j}[\overline{x}, \overline{y_i}, \overline{z_j}; u]) \vee \\
& \quad \cdots \vee \\
& \exists \overline{z_p}(\boldsymbol{A}_p[\overline{x}, \overline{y_i}, \overline{z_p}] \wedge \boldsymbol{D}_{q_p}[\overline{x}, \overline{y_i}, \overline{z_p}; u])
\end{aligned} \tag{2}$$

and construct the m-th fold to be like the $(m+1)$-th fold except that the clauses i through $i+n-1$ are replaced by a single clause:

$$f(\overline{x}) = u \leftarrow \boldsymbol{G}_m[\overline{x}, \overline{y_i}] \wedge \boldsymbol{D}_m[\overline{x}, \overline{y_i}; u] \qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

where $\boldsymbol{G}_m \equiv \boldsymbol{B}_i$ unless the last formula is empty and then $\boldsymbol{G}_m \equiv \top$.

When a recursion refinement was performed at the step m as described in Par. 3.5 then the i-th clause of the $(m+1)$-th fold has the form:

$$f(\overline{x}) = u \leftarrow \boldsymbol{G}_q[\overline{x}, \overline{y_i}, z] \land \boldsymbol{D}_q[\overline{x}, \overline{y_i}, z; u] \qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

for some $q > m$. We define the m-th discrimination formula as follows:

$$\boldsymbol{D}_m[\overline{x}, \overline{y_i}; u] \equiv \exists z(f(\overline{\boldsymbol{t}}[\overline{x}, \overline{y_i}]) = z \land \boldsymbol{D}_q[\overline{x}, \overline{y_i}, z; u]) \tag{3}$$

and construct the m-th fold to be like the $(m+1)$-th fold except that its i-th clause is replaced by

$$f(\overline{x}) = u \leftarrow \boldsymbol{G}_m[\overline{x}, \overline{y_i}] \land \boldsymbol{D}_m[\overline{x}, \overline{y_i}; u] \qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

where $\boldsymbol{G}_m \equiv \boldsymbol{B}_i$ unless the last formula is empty and then $\boldsymbol{G}_m \equiv \top$.

Note that the first fold consists of the single clause:

$$f(\overline{x}) = u \leftarrow \boldsymbol{G}_1 \land \boldsymbol{D}_1[\overline{x}; u] \qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

where $\boldsymbol{G}_1 \equiv \top$.

It is straightforward to see that PA$[+f]$ proves the equivalence of all folds. For all m such that $1 \le m \le s$ we have:

$$\mathrm{PA}[+f] \vdash \boldsymbol{C}[\overline{\boldsymbol{x}}] \land \boldsymbol{G}_m[\overline{x}, \overline{y}] \to \exists! u \boldsymbol{D}_m[\overline{x}, \overline{y}; u] \ . \tag{4}$$

This is proved by backward induction on $m \le s$. The cases when $\boldsymbol{D}_m$ is (1) or (3) are simple. When $\boldsymbol{D}_m$ is (2) then from the assumptions of (4) we have $\boldsymbol{D}[\overline{x}, \overline{y}]$ by the discrimination condition from Par. 3.4. But then by the discrimination condition from Par. 3.1 we consider p cases for some i s.t. $\underline{1} \le i \le \underline{p}$. In the j-th case we have $i = \underline{j}$ and $\boldsymbol{A}_j[\overline{x}, \overline{y}, \overline{z_j}]$ for some $\overline{z_j}$ but then $\boldsymbol{D}_{q_j}[\overline{x}, \overline{y}, \overline{z_j}; u]$ for some u from IH. This proves $\exists u \boldsymbol{D}_m[\overline{x}, \overline{y}; u]$. The uniqueness follows from the uniqueness conditions from Par. 3.1 because for given $\overline{x}, \overline{y}$ the j-th alternative is unique and so $\overline{z_j}$ are unique. We now use $\boldsymbol{U}_u \boldsymbol{D}_{q_j}[\overline{x}, \overline{y}, \overline{z_j}; u]$ from IH.

5.3 Theorem *PA is closed under explicit clausal definitions.*

Proof. By Par. 5.1 it is sufficient to prove the theorem for functions. Consider an explicit clausal definition of f under condition $\boldsymbol{C}$. We assume that all used function and predicate symbols have been previously introduced into PA and such symbols are also applied in $\boldsymbol{C}$. We consider the discriminated formula $\boldsymbol{D}_1[\overline{x}; u]$ which does not apply f and note that $\boldsymbol{G}_1 \equiv \top$. Thus for $m = 1$ the property 5.2(4) is provable in PA by conservativity and we can extend PA by definition:

$$\mathrm{PA}' \vdash f(\overline{x}) = u \leftarrow \boldsymbol{C}[\overline{x}] \land \boldsymbol{D}_1[\overline{x}; u] \lor \neg \boldsymbol{C}[\overline{x}] \land u = 0 \ .$$

Now PA$'$ proves the first fold of the clausal definition:

$$f(\overline{x}) = u \leftarrow \boldsymbol{G}_1[\overline{x}] \land \boldsymbol{D}_1[\overline{x}; u] \qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

and by equivalence also the $(s+1)$-th fold, i.e. the clauses of the definition. $\square$

5.4 Theorem *PA is closed under course of values clausal definitions.*

Proof. By Par. 5.1 it is sufficient to prove the theorem for clausal definitions of functions. Consider a course of values clausal definition of an n-ary function $f(\overline{u}, x, \overline{v})$ which is conditional under $\boldsymbol{C}$. We assume that all used function and predicate symbols have been previously introduced into PA and such symbols are also applied in $\boldsymbol{C}$. The parameters $\overline{u}$ and $\overline{v}$ play no active role in the following (they are not substituted for in recursive applications) and so without a loss of generality we assume them empty.

Let k be the depth of recursion in the definition, i.e the maximal number of recursive applications in one clause. By metatheoretical backward induction on $m \leq s$ we will find for each $\boldsymbol{D}_m[x, \overline{y}; u]$ a term $\boldsymbol{s}_m[x, \overline{x}, \overline{r}]$ where $\overline{r} \equiv r_1, \ldots, r_k$ and for $i = 1, \ldots, k$ terms $\boldsymbol{r}_{i,m}[x, \overline{y}]$ where neither $\boldsymbol{s}_m$, $\boldsymbol{r}_{i,m}$ nor functions applied in them depend on f. Let $l_m \leq k$ stand for the number of recursions in $\boldsymbol{G}_m$.

The terms will be such that for $i = 1, \ldots, k$ we will have:

$$\mathrm{PA}[+f] \vdash \boldsymbol{C}[x] \land \boldsymbol{G}_m[x, \overline{y}] \land \boldsymbol{s}_m[x, \overline{y}, \overline{\boldsymbol{r}}] = u \rightarrow \boldsymbol{D}_m[x, \overline{y}; u] \tag{1}$$

$$\mathrm{PA}[+f] \vdash \boldsymbol{C}[0] \land \boldsymbol{G}_m[0, \overline{y}] \rightarrow \exists u \forall \overline{r}\, \boldsymbol{s}_m[0, \overline{y}, \overline{r}] = u \tag{2}$$

$$\mathrm{PA}[+f] \vdash \boldsymbol{C}[x] \land \boldsymbol{G}_m[x, \overline{y}] \rightarrow \exists u \forall \overline{v}\, \boldsymbol{r}_{i,m}[x, \overline{y}] = u \tag{3}$$

$$\mathrm{PA}[+f] \vdash \boldsymbol{C}[x] \land \boldsymbol{G}_m[x, \overline{y}] \land x > 0 \rightarrow \boldsymbol{r}_{i,m}[x, \overline{y}] < x\ . \tag{4}$$

Here $\overline{\boldsymbol{r}}$ stands for the sequence $w_1, \ldots, w_{l_m}, f(\boldsymbol{r}_{l_m+1,m}[x, \overline{y}]), \ldots, f(\boldsymbol{r}_{k,m}[x, \overline{y}])$ with the sequence of variables $w_1, \ldots, w_{l_m}$ consisting of those among $\overline{y}$ which have been in that order assigned the values of recursions in $\boldsymbol{G}_m$.

The sequence $\overline{v}$ consists of those variables from among $\overline{y}$ which depend on recursion in $\boldsymbol{G}_m$ (note that $\overline{v}$ includes the variables $w_1, \ldots, w_{l_m}$).

The reader will note that the proofs must be carried out in PA[$+f$] because $\boldsymbol{G}_m$ and $\boldsymbol{D}_m$ may depend on f.

If $\boldsymbol{D}_m$ is of the form 5.2(1) obtained by a value refinement then we take $\boldsymbol{s}_m[x, \overline{y}, \overline{r}] \equiv \boldsymbol{t}[x, \overline{y}]$ and for $i = lm + 1, \ldots, k$ we construct $\boldsymbol{r}_{i,m} \equiv 0$. Neither of the terms depends on f. Its easy to see that the above is provable.

If $\boldsymbol{D}_m$ is of the form 5.2(2) obtained by a discrimination refinement then for the condition $\boldsymbol{D}$ of the provable discrimination we have by Par. 3.4:

$$\mathrm{PA}[+f] \vdash \boldsymbol{C}[x] \land \boldsymbol{G}_m[x, \overline{y}] \rightarrow \boldsymbol{D}[x, \overline{y}]\ .$$

By an explicit clausal definition we define

$$g_m(x, \overline{y}, \overline{r}) = \boldsymbol{s}_{q_1}[x, \overline{y}, \overline{z_1}, \overline{r}] \leftarrow \boldsymbol{A}_1[x, \overline{y}, \overline{z_1}] \qquad \leftarrow \boldsymbol{D}[x, \overline{y}]$$
$$\ldots$$
$$g_m(x, \overline{y}, \overline{r}) = \boldsymbol{s}_{q_p}[x, \overline{y}, \overline{z_p}, \overline{r}] \leftarrow \boldsymbol{A}_p[x, \overline{y}, \overline{z_p}] \qquad \leftarrow \boldsymbol{D}[x, \overline{y}]$$

and let $\boldsymbol{s}_m[x, \overline{y}, \overline{r}] \equiv g_m(x, \overline{y}, \overline{r})$. Similarly, for $i = l_{m+1}, \ldots, k$ we define

$$g_{i,m}(x, \overline{y}) = \boldsymbol{r}_{i,q_1}[x, \overline{y}, \overline{z_1}] \leftarrow \boldsymbol{A}_1[x, \overline{y}, \overline{z_1}] \qquad \leftarrow \boldsymbol{D}[x, \overline{y}]$$
$$\ldots$$
$$g_{i,m}(x, \overline{y}) = \boldsymbol{r}_{i,q_p}[x, \overline{y}, \overline{z_p}] \leftarrow \boldsymbol{A}_p[x, \overline{y}, \overline{z_p}] \qquad \leftarrow \boldsymbol{D}[x, \overline{y}]$$

and let $\boldsymbol{r}_{i,m}[x,\overline{y}] \equiv g_{i,m}(x,\overline{y})$. Since the discriminators $\boldsymbol{A}_j$ do not depend on f, neither by IH do $\boldsymbol{s}_m$ or $\boldsymbol{r}_{i,m}$ and so all four properties follow directly from IH. We only note that for $j = 1,\ldots,p$ the variables $\overline{v}$ from $\overline{y}$ not depending on recursion in $\boldsymbol{G}_m$ are among the variables among $\overline{y},\overline{z_j}$ not depending on recursion in $\boldsymbol{G}_{q_j}$.

The only interesting case is when $\boldsymbol{D}_m$ is of the form 5.2(3) obtained by a recursion refinement with $f(\boldsymbol{t}[x,\overline{y}]) = z$ which is the $l_m + 1 = l_q$-st recursive application in the clauses containing it. We let $\boldsymbol{s}_m[x,\overline{y},\overline{r}] \equiv \boldsymbol{s}_q[x,\overline{y},r_q,\overline{r}]$, we set $\boldsymbol{r}_{l_q,m}[x,\overline{y}] \equiv \boldsymbol{t}[x,\overline{y}]$, and for $i = l_q+1,\ldots,k$ we set $\boldsymbol{r}_{i,m}[x,\overline{y}] \equiv \boldsymbol{r}_{i,q}[x,\overline{y},0]$.

We now work in PA$[+f]$ where we assume $\boldsymbol{C}[x]$ and $\boldsymbol{G}_m[x,\overline{y}]$. We have $\boldsymbol{t}[x,\overline{y}] < x$ by regularity and hence $x > 0$. For (1) if $\boldsymbol{s}_m[x,\overline{y},\overline{\boldsymbol{r}}] = u$ then, since

$$u = \boldsymbol{s}_m[x,\overline{y},w_1,\ldots,w_{l_m},f(\boldsymbol{r}_{l_q,m}),\ldots,f(\boldsymbol{r}_{k,m})] = \\ \boldsymbol{s}_q[x,\overline{y},w_1,\ldots,w_{l_m},f(\boldsymbol{t}[x,\overline{y}]),f(\boldsymbol{r}_{l_q+1,m}),\ldots,f(\boldsymbol{r}_{k,m})] \ ,$$

we get $\boldsymbol{D}_q[x,\overline{y},f(\boldsymbol{t}[x,\overline{y}]);u]$ by IH and hence $\boldsymbol{D}_m[x,\overline{y};u]$.

(2) holds trivially because neither $\boldsymbol{C}[0]$ nor $\boldsymbol{G}_m[0,\overline{y}]$ can hold by regularity.

(3): When $i = l_q$ then $\boldsymbol{r}_{i,m} \equiv \boldsymbol{t}$ and no variable from among $\overline{v}$ can occur in $\boldsymbol{t}$ because the definition is by course of values recursion. When $i = l_q+1,\ldots,k$ we have $\exists u\forall z\forall\overline{v}\,\boldsymbol{r}_{i,q}[x,\overline{y},z] = u$ by IH and so $\exists u\forall\overline{v}\,\boldsymbol{r}_{i,q}[x,\overline{y},0] = u$ where the term inside the quantifiers is $\boldsymbol{r}_{i,m}$.

(4): When $i = l_q$ this holds because $\boldsymbol{r}_{i,m} \equiv \boldsymbol{t}$. When $i = l_q+1,\ldots,k$ we have

$$\boldsymbol{r}_{i,m}[x,\overline{y}] \equiv \boldsymbol{r}_{i,q}[x,\overline{y},0] \overset{(3)}{=} \boldsymbol{r}_{i,q}[x,\overline{y},f(x,\overline{y})] \overset{\mathrm{IH}}{<} x \ .$$

With the properties (1) through (4) proved for all m we use them with $m = 1$ where $\boldsymbol{G}_1 \equiv \top$ and $l_1 = 0$. We first explicitly define for all $i = 1,\ldots,k$:

$$\begin{array}{ll} h(x,\overline{r}) = \boldsymbol{s}_1[x,\overline{r}] & \leftarrow \boldsymbol{C}[x] \\ h(x,\overline{r}) = 0 & \leftarrow \neg\boldsymbol{C}[x] \\ & \\ R_i(x) = \boldsymbol{r}_{i,1}[x] & \leftarrow \boldsymbol{C}[x] \\ R_i(x) = 0 & \leftarrow \neg\boldsymbol{C}[x] \ . \end{array}$$

From (4) for all $i = 1,\ldots,k$ we have PA$[+f] \vdash R_i(x+1) < x+1$ and, since the definitions R_i do not depend on f, we get PA $\vdash R_i(x+1) < x+1$ by conservativity. This means that we can use the scheme of course of values recursion from Par. 2.6 to extend PA by definition with the function symbol f to satisfy:

$$\begin{aligned} f(0) &= h(0,0,\ldots,0) \\ f(x+1) &= h(x+1,f(R_1(x+1)),\ldots,f(R_k(x+1))) \ . \end{aligned}$$

We now work in PA$'$ (i.e. PA extended with f) and we will use the above schematic properties with the just defined f to prove that the clauses for f are theorems. For that it is sufficient to prove the first fold of the definition:

$$f(x) = u \leftarrow \top \wedge \boldsymbol{D}_1[x;u] \qquad\qquad \leftarrow \boldsymbol{C}[x]$$

We thus assume $\boldsymbol{C}[x]$ and $\boldsymbol{D}_1[x;u]$. We note that $\overline{\boldsymbol{r}}$ in (1) stands for $f(\boldsymbol{r}_1[x])$, $\ldots$, $f(\boldsymbol{r}_k)$ and we consider two cases. If $x = 0$ then

$$f(0) = \boldsymbol{s}_1[0,0,\ldots,0] \overset{(2)}{=} \boldsymbol{s}_1[0,\overline{\boldsymbol{r}}] \; .$$

If $x > 0$ then

$$f(x) = \boldsymbol{s}_1[x, f(R_1(x)),\ldots,f(R_k(x))] = \boldsymbol{s}_1[x,\overline{\boldsymbol{r}}] \; .$$

In both cases we use (1) to get $\boldsymbol{D}_1[x,f(x)]$. But now $f(x) = u$ by 5.2(4). $\square$

5.5 A direct way of showing the closure of PA under clausal definitions. With course of values recursion admissible it is natural to reduce the general clausal definitions into it. Namely, given a clausal definition of the symbol f of positive arity which is under condition $\boldsymbol{C}$ we can introduce an auxiliary predicate $Ev(a)$ holding of *evaluation trees* a for f. Such trees are among the (arithmetized) trees generated from the *empty* tree 0 by quintuples m,x,u,a_1,a_2 where a_1 and a_2 are trees again. In order to increase the readability we abbreviate $Ev(\underline{m},x,u,c_1,c_2)$ to $c_1,c_2 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x;u]$. This suggests a reading of the application as holding when the ***e***valuation (relatively to functions used in the definition) of the discriminated formula $\boldsymbol{D}_m[\overline{x},\overline{y};u]$ yields the value u for the values of input variables given by the *valuation* $x = \overline{x},\overline{y},0$. The numbers c_1 and c_2 encode (possibly empty) subevaluations. Evaluation can be viewed as a special form of a proof, hence the symbol $\vdash$. The symbol $\bullet$ designates that the proof predicate is arithmetized. See Sect. 6 for the use of the same provability predicate at the metatheoretical level. The recursion in the following definition of $Ev(\underline{m},x,u,c_1,c_2)$ is easily seen to be course of values because it goes at most to c_1 and c_2.

For every discriminated formula $\boldsymbol{D}_m[\overline{x},\overline{y};u]$ of the above clausal definition we generate one or more (abbreviated non-default) clauses for Ev as follows. When $\boldsymbol{D}_m[\overline{x},\overline{y};u]$ is of the form 5.2(1) obtained by a value refinement we form the clause:

$$0,0 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[\overline{x},\overline{y},0;\boldsymbol{t}[\overline{x},\overline{y}]] \qquad\qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

When $\boldsymbol{D}_m[\overline{x},\overline{y};u]$ is of the form 5.2(2) obtained by a discrimination refinement we form the following p clauses:

$$\begin{aligned}
&0,\underline{q_1},(\overline{x},\overline{y},\overline{z_1},0),u,c \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[\overline{x},\overline{y},0;u] \leftarrow \boldsymbol{A}_1[\overline{x},\overline{y},\overline{z_1}] \land \\
&\qquad\qquad c \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_{q_1}[\overline{x},\overline{y},\overline{z_1},0;u] \qquad \leftarrow \boldsymbol{C}[\overline{x}] \\
&\ldots \\
&0,\underline{q_p},(\overline{x},\overline{y},\overline{z_p},0),u,c \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[\overline{x},\overline{y},0;u] \leftarrow \boldsymbol{A}_p[\overline{x},\overline{y},\overline{z_p}] \land \\
&\qquad\qquad c \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_{q_p}[\overline{x},\overline{y},\overline{z_p},0;u] \qquad \leftarrow \boldsymbol{C}[\overline{x}]
\end{aligned}$$

When $\boldsymbol{D}_m[\overline{x},\overline{y};u]$ is of the form 5.2(3) obtained by a recursion refinement we form the clause:

$$\begin{aligned}
&(\underline{1},(\overline{\boldsymbol{t}}[\overline{x},\overline{y}],0),z,c_1),\underline{q},(\overline{x},\overline{y},z,0),u,c_2 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[\overline{x},\overline{y},0;u] \leftarrow \\
&\quad c_1 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{\boldsymbol{t}}[\overline{x},\overline{y}],0;z] \land c_2 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_q[\overline{x},\overline{y},z,0;u] \qquad \leftarrow \boldsymbol{C}[\overline{x}]
\end{aligned}$$

We can now introduce f and prove its clauses as theorems exactly as in the proof of Thm. 5.8. We refrain from the above definition of the evaluation predicate Ev because we wish to analyze in Thm. 6.13 its quantifier complexity. We also wish to compute in Sect. 6 clausal definitions by direct 'dearithmetization' of the evaluation trees in the metatheory. Toward both ends we introduce in the following paragraphs case discriminations (this will permit to 'decide' them) and we put clausal definitions into a normal form which can be easily evaluated.

5.6 Case discrimination. Consider a provable discrimination of a form as in Par. 3.1. A term $\boldsymbol{t}[\overline{x}]$ is called a provable *case term* for this discrimination if for all $j = 1, \ldots, p$ we have

$$\mathrm{PA} \vdash D[\overline{x}] \to (\boldsymbol{A}_j[\overline{x}, y_1, \ldots, y_{k_j}] \leftrightarrow \boldsymbol{t}[\overline{x}] = \underline{j}, y_1, \ldots, y_{k_j}, 0) \ .$$

The case term can be thus used to compute the discrimination. For a given $\overline{x}$ s.t. $D[\overline{x}]$ we have $\boldsymbol{t}[\overline{x}] = j, y$ for some j and y. The *tag* j is such that $1 \leq j \leq p$ and it selects the discriminator (alternative). The list y is of length k_j and it gives the values of the output variables such that $\boldsymbol{A}_j[\overline{x}, (y)_0, \ldots, (y)_{k_j-1}]$ holds.

For the above case term we call the following provable discrimination a *case* discrimination:

$$\mathbf{discr}_{\overline{x}} \quad \boldsymbol{t}[\overline{x}] = \underline{1}, \overline{y_1}, 0 \mid \cdots \mid \boldsymbol{t}[\overline{x}] = \underline{j}, \overline{y_j}, 0 \mid \cdots \mid \boldsymbol{t}[\overline{x}] = \underline{p}, \overline{y_p}, 0 \qquad \leftarrow \boldsymbol{D}[\overline{x}]$$

We can make case discriminations unconditional by the addition of an 'otherwise' discriminator:

$$\neg(\boldsymbol{t}[\overline{x}] = 0, y \wedge L(y) = \underline{k_1} \vee \cdots \vee \boldsymbol{t}[\overline{x}] = \underline{j}, y \wedge L(y) = \underline{k_j} \vee \cdots \vee \boldsymbol{t}[\overline{x}] = \underline{p}, y \wedge L(y) = \underline{k_p})$$

where L is the unary *list length* function defined by course of values recursion to satisfy $L(0) = 0$ and $L(v, w) = L(w) + 1$. Clausal definitions using such unconditional case discrimination can simply make the clauses corresponding to the otherwise discriminator the default clauses.

Every provable discrimination has a provable case term. Indeed, for the above discrimination we first introduce the auxiliary functions d_j for $j = 1, \ldots, p$ by explicit clausal definitions:

$$d_j(\overline{x}) = 0, \overline{y_j}, 0 \leftarrow (\boldsymbol{D}[\overline{x}] \wedge \exists \overline{y_j}(\boldsymbol{A}_j[\overline{x}, \overline{y_j}]) \wedge \boldsymbol{A}_j[\overline{x}, \overline{y_j}] \ .$$

The definition uses discrimination on negation with the parenthesized formula and the second clause with the negated not shown by default. The first clause is then discriminated by the least number discrimination (the first one in Par. 4.12) because its existence and uniqueness is implied by the parenthesized formula and by the uniqueness condition for $\boldsymbol{A}_j$.

We now define the case term for the discrimination to be $d(\overline{x})$ where the function d has an explicit clausal definition (with pair and zero discriminations):

$$\begin{aligned}
&d(\overline{x}) = \underline{1}, y \leftarrow d_1(\overline{x}) = 0, y \\
&d(\overline{x}) = \underline{2}, y \leftarrow d_1(\overline{x}) = 0 \wedge d_2(\overline{x}) = 0, y \\
&\quad \ldots \\
&d(\overline{x}) = \underline{n}, y \leftarrow d_1(\overline{x}) = 0 \wedge d_2(\overline{x}) = 0, y \wedge \cdots \wedge d_{n-1}(\overline{x}) = 0 \wedge d_n(\overline{x}) = 0, y \ .
\end{aligned}$$

A clausal definition is said to be in a *case form* if all its discriminations are case discriminations. Note that if such a definition defines a function then the definition cannot use predicate symbols as they can occur only in the formulas of provable discriminators. Every clausal definition has by the above a provably equivalent clausal definition in a case form.

5.7 Normal form of clausal definitions. Every clausal definition of f can be brought to a provably equivalent case form by Par. 5.6. We will now show how every clausal definition in a case form can be brought into a provably equivalent *normal* form. A definition in normal form has all applications of function symbols (including constants) in a form $\boldsymbol{p}(\overline{y}) = z$. The function symbol $\boldsymbol{p}$ is either f and then the above atomic formula is a recursion refinement or else p is a used function symbol and then the formula is a 'pure' let discrimination.

A normal form can be produced by reconstructing the clausal definition in a 'normalization' process with its refinements put into a more restrictive form. A value refinement from Par. 3.6 in the old definition will be allowed in the new definition only if its term $\boldsymbol{t}$ is a variable. If this is not the case we locate the leftmost and innermost application $\boldsymbol{g}(\overline{v})$ in $\boldsymbol{t}$ and precede the value refinement with a pure let discrimination $\boldsymbol{g}(\overline{v}) = z$ where z is a new variable. We then replace the application in $\boldsymbol{t}$ with the variable z whereby we obtain a term $\boldsymbol{t}_1$ with one application less than $\boldsymbol{t}$. We repeat this process until we obtain a variable term which is then used in a value refinement.

A discrimination refinement from Par. 3.4 in the old definition will be allowed in the new definition only if its case term $\boldsymbol{t}$ is a variable v. Note that then the discriminants will be $v = \underline{j}, \overline{y_j}, 0$ for $j = 1, \ldots, p$ (see Par. 5.6). If $\boldsymbol{t}$ is not a variable we precede the discrimination refinement by a number of pure let refinements until we reduce the case term into a variable.

A recursion refinement from Par. 3.5 in the old definition will be allowed in the new definition only if the argument terms $\overline{\boldsymbol{t}}$ are variables, i.e. the refinement looks like a pure let refinement. We precede the recursion refinement with sufficiently many pure let refinements until all arguments of f are variables.

In order to simplify the proof of Thm. 5.8 we can in the normalization process reserve the variables $x_0, \ldots, x_{n-1}$ for the arguments of the defined function and use as the output variables z in $\boldsymbol{p}(\overline{y}) = z$ and $\overline{y_j}$ in $v = \underline{j}, \overline{y_j}, 0$ the variables x with the successive indices.

The normalized definition thus obtained in s steps will have for $m = 1, \ldots, s$ the discrimination formulas in the form $\boldsymbol{D}_m[x_0, \ldots, x_{l_m}; u]$ with indices l_m such that $n - 1 = l_1 \leq l_m$. We list in the following the discrimination formulas obtained in that order by refinements by value (where $i \leq l_m$), by pure let (including recursion) (where $j_1, \ldots, j_k \leq l_m$, $l_m + 1 = l_q$), and by discrimination (where $i \leq l_m \leq l_{q_1}, \ldots, l_{q_p}$):

$$x_i = u \tag{1}$$

$$\exists x_{l_q} (\boldsymbol{p}(x_{j_1}, \ldots, x_{j_k}) = x_{l_q} \land \boldsymbol{D}_q) \tag{2}$$

$$\begin{aligned} &\exists x_{l_m+1} \ldots \exists x_{l_{q_1}} (x_i = \underline{1}, x_{l_m+1}, \ldots, x_{l_{q_1}}, 0 \land \boldsymbol{D}_{q_1}) \lor \\ &\quad \ldots \lor \\ &\exists x_{l_m+1} \ldots \exists x_{l_{q_p}} (x_i = \underline{p}, x_{l_m+1}, \ldots, x_{l_{q_p}}, 0 \land \boldsymbol{D}_{q_p}) \end{aligned} \tag{3}$$

5.8 Theorem *PA is closed under clausal definitions.*

Proof. By Par. 5.1 it is sufficient to prove the theorem for functions and by Par. 5.7 for clausal definitions in the normal form. So take any such definition of an n-ary function symbol f under a condition $\boldsymbol{C}[\overline{x}]$. Assume, further, that all used function and predicate symbols have been previously introduced into PA and such symbols are also applied in $\boldsymbol{C}$. If the definition is explicit we use Thm. 5.3, if the definition is by course of values recursion then use Thm. 5.4. Otherwise we have $n > 0$ and there is a provably well-founded relation $\prec$ such that all recursive applications are regular in it.

We introduce below the predicate *Ev* holding of evaluation trees (see Par. 5.5) by an explicit definition. For that we need the two place predicate $c \sqsubset a$ holding whenever c is a subtree of a. The predicate has a course of values clausal definition on a:

$$\begin{aligned} &c \sqsubset a \leftarrow c = a \\ &c \sqsubset a \leftarrow c \neq a \land a = m, x, y, a_1, a_2 \land c \sqsubset a_1 \\ &c \sqsubset a \leftarrow c \neq a \land a = m, x, y, a_1, a_2 \land c \not\sqsubset a_1 \land c \sqsubset a_2 \end{aligned}$$

PA easily proves that $\sqsubset$ is a partial order, i.e. that it is reflexive, transitive, and antisymmetric.

The evaluation predicate is now introduced into PA by an explicit definition:

$$\begin{aligned} Ev(a) \leftrightarrow \exists m \exists x \exists y \exists u \exists a_1 \exists a_2 (a = m, x, u, a_1, a_2 \land \boldsymbol{C}[(x)_0, \ldots, (x)_{l_1}]) \land \\ \forall c{\leq}a (c \sqsubset a \rightarrow c = 0 \lor \boldsymbol{P}_1[c] \lor \cdots \lor \boldsymbol{P}_m[c] \lor \cdots \lor \boldsymbol{P}_s[c]) \end{aligned}$$

where $m = 1, \ldots, s$ are the steps needed for the construction of the clausal definition and $\boldsymbol{P}_m$ is constructed according to the form of the discriminated formula $\boldsymbol{D}_m[x_0, \ldots, x_{l_m}]$ (see Par. 5.7). When $\boldsymbol{D}_m$ was formed by the value refinement 5.7(1) we abbreviate by $\boldsymbol{P}_m[c]$:

$$\exists x\, c = \underline{m}, x, (x)_i, 0, 0 \ .$$

When $\boldsymbol{D}_m$ was formed by the let refinement 5.7(2) with $\boldsymbol{p} \equiv g$ we abbreviate by $\boldsymbol{P}_m[c]$:

$$\exists x \exists z \exists u \exists c_1 (c = \underline{m}, x, u, 0, \underline{q}, x \oplus (z, 0), u, c_1 \land g((x)_{\underline{j_1}}, \ldots, (x)_{\underline{j_k}} = z)) \tag{1}$$

When $\boldsymbol{D}_m$ was formed by the recursion refinement 5.7(2) with $\boldsymbol{p} \equiv f$ and $k = n$ we abbreviate by $\boldsymbol{P}_m[c]$:

$$\exists x \exists u \exists z \exists c_1 \exists c_2\, c = \underline{m}, x, u, (\underline{1}, ((x)_{\underline{j_1}}, \ldots, (x)_{\underline{j_n}}, 0), z, c_1), \underline{q}, x \oplus (z, 0), u, c_2 \ .$$

When $\boldsymbol{D}_m$ was formed by the discrimination refinement 5.7(3) we abbreviate by

$$\boldsymbol{P}_m[c] \equiv \boldsymbol{P}_{1,m}[c] \vee \cdots \vee \boldsymbol{P}_{j,m}[c] \vee \cdots \vee \boldsymbol{P}_{p,m}[c]$$

where $\boldsymbol{P}_{j,m}[c]$ is:

$$\exists x \exists u \exists u \exists c_1 (c = \underline{m}, x, u, 0, \underline{q_j}, x \oplus y, u, c_1 \wedge (x)_{\underline{i}} = \underline{j}, y) \ .$$

From the transitivity of $\sqsubset$ we immediately get:

$$\mathrm{PA} \vdash Ev(a) \wedge c \sqsubset a \to Ev(c) \ . \tag{2}$$

We abbreviate by $\vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x; u]$ the formula $\exists c_1 \exists c_2\, c_1, c_2 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x; u]$. We would like to prove $\exists! u \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{x}, 0; u]$ whenever $\boldsymbol{C}[\overline{x}]$ holds. However, this seems to be impossible without introducing first a function f such that PA proves the following clause:

$$f(\overline{x}) = u \leftarrow \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{x}, 0; u] \qquad\qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

We are in an apparent vicious circle because for that we need the existence and uniqueness condition. We can go around the problem by defining f by an explicit clausal definition using the least number discrimination:

$$f(\overline{x}) = u \leftarrow \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[(\overline{x}, 0); u] \qquad\qquad \leftarrow \boldsymbol{C}[\overline{x}] \wedge \exists! u \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{x}, 0; u]$$

Until the end of the proof we will work in PA extended with this f.

We abbreviate by $\boldsymbol{W}[v]$ the following formula:

$$\begin{aligned} &\forall x_0 \ldots \forall x_{l_1} (v = x_0, \ldots, x_{l_1} \wedge \boldsymbol{C}[x_0, \ldots, x_{l_1}] \to \exists u \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[x_0, \ldots, x_{l_1}, 0; u] \wedge \\ &\qquad \forall u (\vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[x_0, \ldots, x_{l_1}, 0; u] \to \boldsymbol{D}_1[x_0, \ldots, x_{l_1}; u])) \end{aligned}$$

and prove

$$\mathrm{PA} \vdash \boldsymbol{W}[\overline{x}] \wedge \boldsymbol{C}[\overline{x}] \to \exists! u \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{x}, 0; u] \tag{3}$$

by assuming the antecedent and obtaining $\vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{x}, 0; u_1]$ for some u_1 from the first conclusion of $\boldsymbol{W}[\overline{x}]$ and then $\boldsymbol{D}_1[\overline{x}; u_1]$ from the second conclusion. If also $\vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{x}, 0; u_2]$ then $\boldsymbol{D}_1[\overline{x}; u_2]$ from the second conclusion again. We now use 5.2(4) to derive $u_1 = u_2$ (recall that $\boldsymbol{G}_1 \equiv \top$). Hence $\exists! u \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{x}, 0; u]$.

We prove next by metatheoretical backward induction on $m = 1, \ldots, s$:

$$\begin{aligned} \mathrm{PA} \vdash\ &\forall w \prec x_0, \ldots, x_{l_1}\, \boldsymbol{W}[w] \wedge \boldsymbol{C}[x_0, \ldots, x_{l_1}] \wedge \boldsymbol{G}_m[x_0, \ldots, x_{l_m}] \to \\ &\quad \exists u \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x_0, \ldots, x_{l_m}, 0; u] \wedge \\ &\quad \forall u (\vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x_0, \ldots, x_{l_m}, 0; u] \to \boldsymbol{D}_m[x_0, \ldots, x_{l_m}; u]) \ . \end{aligned} \tag{4}$$

We thus take any $x_0, \ldots, x_{l_m}$, set $x = x_0, \ldots, x_{l_m}, 0$, assume the antecedent, and consider the four cases for $\boldsymbol{D}_m$.

If $\boldsymbol{D}_m$ is 5.7(1) then for $a = \underline{m}, x, x_i, 0, 0$, and $c \sqsubset a$ we have either $c = 0$ or $c = a$ and in the latter case $\boldsymbol{P}_m[c]$. Hence $Ev(a)$, i.e. $0, 0 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x; x_i]$, holds. This proves the first part of the conclusion $\exists u \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x; u]$. For the proof of the second part assume $b_1, b_2 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x; u]$ for some b_1, b_2, u. Thus $Ev(b)$ where $b = \underline{m}, x, u, b_1, b_2$. Since $b \sqsubset b$, we have $\boldsymbol{P}_m[b]$ and hence $x_i = u$ which is $\boldsymbol{D}_m$.

If $\boldsymbol{D}_m$ is 5.7(2) with $\boldsymbol{p} \equiv g$ then we take $x_{l_q} = g(x_{j_1}, \ldots, x_{j_k})$ and $x' = x_0, \ldots, x_{l_m}, x_{l_q}, 0 = x \oplus (x_{l_q}, 0)$. We note that $\boldsymbol{G}_q$ is $\boldsymbol{G}_m \wedge g(x_{j_1}, \ldots, x_{j_k}) = x_{l_q}$.

For the first part of the conclusion we obtain from the first part of IH $b_1, b_2 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_q[x'; u]$ for some u, b_1, b_2. Thus $Ev(b)$ for $b = \underline{q}, x', u, b_1, b_2$. We consider $a = \underline{m}, x, u, 0, b$ and for the proof of $Ev(a)$ we take any $c \sqsubset a$ and consider the three cases for c. If $c = a$ we have $P_m[c]$. If $c \sqsubset 0$ then $c = 0$. If $c \sqsubset b$ then one of $\boldsymbol{P}_j[c]$ or $c = 0$ holds from $Ev(b)$. We have thus proved $0, b \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x; u]$ from which the first part follows.

For the proof of the second part assume $b_1, b_2 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x; u]$ for some b_1, b_2, u. Thus $Ev(b)$ where $b = \underline{m}, x, u, b_1, b_2$. From $b \sqsubset b$ we get $\boldsymbol{P}_m[b]$ and so $b_1 = 0$, $b_2 = \underline{q}, x', u, b_3$ for some b_3. Since $b_2 \sqsubset b$, we have $Ev(b_2)$ by (2). Thus $b_3 = b_4, b_5$ for some b_4, b_5 and $b_3, b_4 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_q[x'; u]$. We get $\boldsymbol{D}_q[x_0, \ldots, x_{l_m}, x_{l_q}; u]$ from the second part of the IH and hence $\boldsymbol{D}_m[x_0, \ldots, x_{l_m}; u]$.

If $\boldsymbol{D}_m$ is 5.7(2) with $\boldsymbol{p} \equiv f$ and $k = n$ we have $x_{j_1}, \ldots, x_{j_n} \prec v$ by the condition of regularity and $\boldsymbol{C}[x_{j_1}, \ldots, x_{j_n}]$ by the condition of applicability. We get $\boldsymbol{W}[x_{j_1}, \ldots, x_{j_n}]$ from the first assumption of (4) and $\exists! u \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[x_{j_1}, \ldots, x_{j_n}, 0; u]$ from (3). Thus $d_1, d_2 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[x_{j_1}, \ldots, x_{j_n}, 0; x_{l_q}]$ for some d_1, d_2, x_{l_q}. We now use the defining clause for f to obtain $f(x_{j_1}, \ldots, x_{j_n}) = x_{l_q}$. We note that $\boldsymbol{G}_q$ is $\boldsymbol{G}_m \wedge f(x_{j_1}, \ldots, x_{j_n}) = x_{l_q}$.

For the first part of the conclusion we set $x' = x_0, \ldots, x_{l_m}, x_{l_q}, 0 = x \oplus (x_{l_q}, 0)$ and obtain from the first part of IH $b_1, b_2 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_q[x'; u]$ for some u, b_1, b_2. Thus $Ev(b)$ for $b = \underline{q}, x', u, b_1, b_2$. We note that $d_1, d_2 \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[x_{j_1}, \ldots, x_{j_n}, 0; x_{l_q}]$ is $Ev(d)$ for $d = \underline{1}, (x_{j_1}, \ldots, x_{j_n}, 0), x_{l_q}, d_1, d_2$. The reader will note that this is where we need the function f to be introduced because with f as a (schematic) variable we would not be able to connect the result x_{l_q} of the recursive evaluation with the value $f(x_{j_1}, \ldots, x_{j_n})$ which is needed in order to apply IH.

We consider $a = \underline{m}, x, u, d, b$ and for the proof of $Ev(a)$ we take any $c \sqsubset a$ and consider the three cases for c. If $c = a$ we have $P_m[c]$. If $c \sqsubset d$ or $c \sqsubset b$ then one of $\boldsymbol{P}_j[c]$ or $c = 0$ holds from $Ev(d)$ or from $Ev(b)$ respectively. We have thus proved $d, b \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x; u]$ from which the first part of the conclusion follows.

The second part is proved similarly as in the case when $\boldsymbol{p} \equiv g$. Since this brings nothing new except more case analysis, we leave the discrimination case when $\boldsymbol{D}_m$ is 5.7(3) to the reader.

We now claim that

$$\mathrm{PA} \vdash \boldsymbol{C}[\overline{x}] \to \exists u \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{x}, 0; u] \wedge \forall u(\vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{x}, 0; u] \to \boldsymbol{D}_1[\overline{x}; u])) \tag{5}$$

This follows from $\boldsymbol{W}[v]$ with $v := x_0, \ldots, x_{l_1} \equiv \overline{x}$. It is thus sufficient to prove $\boldsymbol{W}[v]$ by by $\prec$-induction on v. We assume as IH $\forall w \prec v\, \boldsymbol{W}[w]$, take any $x_0, \ldots, x_{l_1}$

such that $v = x_0, \ldots, x_{l_1}$, and assume $\boldsymbol{C}[x_0, \ldots, x_{l_1}]$. Since $\boldsymbol{G}_1 \equiv \top$, we use (4) with $m = 1$ to obtain the conclusion.

The clauses for f and the theorem will follow by the equivalence of folds when we prove the first fold of the clauses for f:

$$f(\overline{x}) = u \leftarrow \boldsymbol{G}_1[x_0, \ldots, x_{l_1}] \wedge \boldsymbol{D}_1[\overline{x}; u] \qquad\qquad \leftarrow \boldsymbol{C}[\overline{x}]$$

We thus assume $\boldsymbol{C}[\overline{x}]$ and $\boldsymbol{D}_1[\overline{x}; u]$. Since $\boldsymbol{W}[\overline{x}]$ follows from (5), we get $\exists! u \vdash^{\bullet}_{e} \boldsymbol{D}_1[\overline{x}, 0; u]$ from (3). Hence $\vdash^{\bullet}_{e} \boldsymbol{D}_1[\overline{x}, 0; u_1]$ for some u_1 and $\boldsymbol{D}_1[\overline{x}; u_1]$ from the second conclusion of (5). We then get $u = u_1$ from 5.2(4). We have thus satisfied all assumptions of the defining clause for f and we can conclude $f(\overline{x}) = u$. $\square$

6 Efficient Computation with Clausal Definitions

We will define in this section a class of *clausally computable* functions which can be efficiently computed from their conditional clausal definitions for all arguments satisfying the conditions. If the condition cannot be decided the best one can do is to compute such a function as a partial function. We do not deal with predicates because they can be computed from their characteristic functions (see Par. 5.1).

We will characterize in Thm. 6.14 the clausally computable functions with conditions $\top$ as exactly the provably recursive functions of PA. The proof is far from trivial because it involves the removal of detours through functions with possibly non-computable conditions.

Clausally Computable Functions and their Characterization

6.1 General applicability conditions. If we intend to compute a function f defined by a clausal definition then there are additional conditions of *applicatibility* which must be satisfied for every application in the clausal definition of a used function or predicate symbol $\boldsymbol{p}(\overline{y})$ defined by a clausal definition under a condition $\boldsymbol{D}[\overline{y}]$. The conditions extend those from Par. 3.5 to arbitrary functions and predicates.

The symbol $\boldsymbol{p}$ can become applied as $\boldsymbol{p}(\overline{\boldsymbol{s}}[\overline{x}, \overline{y_i}])$ in a clasual definition of f by each of the three refinement steps. The application of $\boldsymbol{p}$ can occur in a discrimination refinement in a discriminator $\boldsymbol{A}_j[\overline{x}, \overline{y_i}, \overline{z_j}]$, in a recursion refinement within an argument of recursion, or in a value refinement within the value $\boldsymbol{t}[\overline{x}, \overline{y_i}]$. The condition of *applicability* for the application $\boldsymbol{p}(\overline{\boldsymbol{s}}[\overline{x}, \overline{y_i}])$ is

$$\mathrm{PA}[+f] \vdash \boldsymbol{C}[\overline{x}] \wedge \boldsymbol{B}_i[\overline{x}, \overline{y_i}] \rightarrow \boldsymbol{D}[\overline{\boldsymbol{s}}[\overline{x}, \overline{y_i}]] \ .$$

Note that the conditions of applicability are relevant only when we intend to compute applications of f. Its clausal definition remains perfectly legal (if possibly uncomputable) even if it does not satisfy applicability conditions for all symbols $\boldsymbol{p}$ applied in it. This is because we are obliged in the first-order logic to introduce all functions as total functions and so any application of $\boldsymbol{p}$ in the definition of f denotes in any model of PA regardless of the arguments. On the other hand, the applicability conditions for recursive applications of f are crucial because they are needed in the introduction of f via its evaluation tree (see Thm. 5.8).

6.2 Basic discriminations. Although monadic discrimination (see Par. 4.4) suffices for the clausal definition of all provably recursive functions of PA we insist for reasons of comfortable programming on arbitrary set of provable discriminations. Although case terms for any provable discrimination can be constructed in a computable way from the case term for monadic discrimination we will use in the following a different set of basic case terms because our main goal is not merely *effective* but also *efficient* computation.

Our basic case terms are $z(x)$, $b(x)$, and $p(x)$ and they are associated in that order with zero Par. 4.2, binary Par. 4.5, and pair Par. 4.7 discriminations. The functions applied in the case terms are introduced into PA by extension by definition as follows:

$$\begin{aligned}
\mathrm{PA}' &\vdash z(x) = y \leftrightarrow x = 0 \land y = 0, 0 \lor x > 0 \land y = 1, 0\\
\mathrm{PA}'' &\vdash b(x) = z \leftrightarrow \exists y(x = 2{\cdot}y \land z = 0, y, 0 \lor x = 2{\cdot}y + 1 \land z = 1, y, 0)\\
\mathrm{PA}''' &\vdash (x) = z \leftrightarrow z = 0, 0 \lor \exists v \exists w(x = v, w \land z = 1, v, w, 0)\ .
\end{aligned}$$

We leave to the reader the proofs in PA that the three are indeed case terms for the corresponding discriminations.

6.3 Clausally computable functions. We will now define the class of clausally computable functions where we admit also the functions of arity 0, i.e. constants.

We say that an n-ary function symbol f introduced into PA by an extension by definition *represents* a function over natural numbers if the function is the interpretation of f in the standard model $\mathcal{M}$ of PA.

The *basic* functions are the constant 0, the binary successors S_0, S_1, the pairing function $\cdot,\cdot$, and the basic discrimination functions z, b, and p. The functions are represented in PA by definitions in in Paragraphs 4.5, 2.7, and 6.2. We treat the basic functions as being unconditional, i.e. their conditions are $\top$.

The class of *clausally computable* functions is the least class of functions which contains the basic functions and every function represented in an extension of PA by a function symbol defined by a clausal definition in the case form whose used function symbols represent clausally computable functions and satisfy the applicability conditions. A clausally computable function is *unconditional* if it has a representing function symbol with unconditional clausal definition.

Note that there are no restriction on the conditions $\boldsymbol{C}$ of clausal definitions representing clausally computable functions.

6.4 Some clausally computable arithmetical functions. The successor, addition and multiplication functions are clausally computable functions because their (abbreviated) clauses given in Fig. 1 are obtained by binary and zero discriminations.

$$
\begin{array}{ll}
S(x\mathbf{0}) = x\mathbf{1} & \\
S(x\mathbf{1}) = S(x)\mathbf{0} & \\
 & \\
0\mathbf{0} + y = y & \\
x\mathbf{0} + 0 = x\mathbf{0} & \leftarrow x > 0 \\
x\mathbf{0} + y\mathbf{0} = (x + y)\mathbf{0} & \leftarrow x > 0 \land y > 0 \\
x\mathbf{0} + y\mathbf{1} = (x + y)\mathbf{1} & \leftarrow x > 0 \\
x\mathbf{1} + 0 = x\mathbf{1} & \\
x\mathbf{1} + y\mathbf{0} = (x + y)\mathbf{1} & \leftarrow y > 0 \\
x\mathbf{1} + y\mathbf{1} = S(x + y)\mathbf{0} & \\
 & \\
0\mathbf{0}{\cdot}y = 0 & \\
x\mathbf{0}{\cdot}y = (x{\cdot}y)\mathbf{0} & \leftarrow x > 0 \\
x\mathbf{1}{\cdot}y = (x{\cdot}y)\mathbf{0} + y &
\end{array}
$$

Fig. 1. Clauses for the basic arithmetic functions

The clauses in Fig. 1 can be easily proved theorems of PA. We, however, intend to compute the operations from their clauses and so PA must prove that the clauses are regular. The clauses for S and multiplication are by course of values recursion (in the case of multiplication on the first argument). The first argument x of addition also decreases in $<$ in recursive applications but it is not course of values recursion because the parameter y is modified.

6.5 Clausally computable terms, predicates, and discriminations. We will find it of advantage to speak of clausally computable terms, predicates, and discriminations.

A term $\boldsymbol{s}[\overline{x}]$ is *clausally computable under condition* $\boldsymbol{D}[\overline{x}]$ if the following explicit clausal definition of a new function symbol f represents a clausally computable function:

$$f(\overline{x}) = \boldsymbol{s}[\overline{x}] \qquad\qquad \leftarrow \boldsymbol{D}[\overline{x}]$$

This clearly means that the functions representing the symbols used in the definition, i.e. applied in $\boldsymbol{s}$, are clausually computable and their applicability conditions are provable.

A predicate P is *clausally computable under condition* $\boldsymbol{D}[\overline{x}]$ if its characteristic function is.

A provable discrimination under condition $\boldsymbol{D}[\overline{x}]$ is *clausally computable* if it has a provable case term $\boldsymbol{s}[\overline{x}]$ clausally computable under $\boldsymbol{D}[\overline{x}]$.

Thus clausal definitions with clausally computable discriminations are provably equivalent to clausal definitions in a case form. Consequently, we can use such definitions to represent clausally computable functions.

$$
\begin{array}{ll}
x\mathbf{0} \dot{-}_1 0 = 0, x\mathbf{0} & \leftarrow x > 0 \\
x\mathbf{0} \dot{-}_1 y\mathbf{0} = 0, z\mathbf{0} & \leftarrow x > 0 \land y > 0 \land x \dot{-}_1 y = 0, z \\
x\mathbf{0} \dot{-}_1 y\mathbf{1} = 0, z\mathbf{1} & \leftarrow x > 0 \land Pr(x) \dot{-}_1 y = 0, z \\
x\mathbf{1} \dot{-}_1 0 = 0, x\mathbf{1} & \\
x\mathbf{1} \dot{-}_1 y\mathbf{0} = 0, z\mathbf{1} & \leftarrow y > 0 \land x \dot{-}_1 y = 0, z \\
x\mathbf{1} \dot{-}_1 y\mathbf{1} = 0, z\mathbf{0} & \leftarrow x > 0 \land y > 0x \land \dot{-}_1 y = 0, z \\
 & \\
x \dot{-} y = z & \leftarrow x \dot{-}_1 y = 0, z \\
 & \\
d_2(x\mathbf{0}) = \underline{1}_b, x, 0, 0 & \\
d_2(x\mathbf{1}) = \underline{1}_b, x, \underline{1}_b, 0 & \\
 & \\
d_4(x) = \underline{1}_b, z, 2{\cdot}j + i, 0 & \leftarrow x = 2{\cdot}y + i \land i < 2 \land y = 2{\cdot}z + j \land j < 2 \\
 & \\
dich(x, y) = \underline{1}_b, 0 & \leftarrow x \dot{-} y = 0 \\
dich(x, y) = \underline{2}_b, 0 & \leftarrow x \dot{-} y > 0
\end{array}
$$

Fig. 2. Some clausally computable functions used in discrimination terms.

6.6 Clausally computable discrimination on addition and division. It should be clear that following in that order one-alternative discriminations on *addition*, *division by* 2 and *division by* 4 are provable in PA:

$$
\begin{array}{lll}
\mathbf{discr}_{x,y} & x = y + d & \leftarrow x \geq y \\
\mathbf{discr}_{x} & x = 2{\cdot}q + r \land r < 2 & \\
\mathbf{discr}_{x} & x = 4{\cdot}q + r \land r < 4 \ . &
\end{array}
$$

The three discriminators are an excellent example (see Par. 6.7 for their use) of the superiority in terms of readability of pattern matching over the explicit use of the inverse functions $x \dot{-} y$, $x \div 2$, $x \bmod 2$, $x \div 4$, $x \bmod 4$. The use of the latter functions is so much less readable that the new CL with the extensible syntax will not have the functions builtin. This will also simplify the theorem prover because the handling of properties of the inverse functions takes much of its code in the present implementation of CL..

The discriminations are claussally computable because the following are in that order their provable case terms $(\underline{1}_b, x \dot{-} y, 0)$, $d_2(x)$, and $d_4(x)$ with the clausally computable definitions of their three functions given in Fig. 2.

6.7 Some clausally computable functions based on pairing. Our modified Cantor's pairing function satisfies $x, y = f(x{+}y){+}x$ where $f(d) = 1{+}\sum_{i \leq d} i$ is the *diagonal* function. The diagonal function satisfies $2{\cdot}f(d) = d^2 + d + 2$ and it is clausally computable by the following explicit definition using discrimination on division by 2:

$$
f(d) = p + \underline{1}_b \leftarrow d{\cdot}(d + \underline{1}_b) = 2{\cdot}p + r \land r < 2 \ .
$$

In order to define the projections H and T as clausally computable functions we need the inverse f^{-1} of the diagonal function which satisfies $f(f^{-1}(x)) \leq x < f(f^{-1}(x)+1)$ whenever $x > 0$. This is for $f^{-1}(x) = d$ equivalent to $d^2+d+2 \leq 2{\cdot}x \leq d^2+3{\cdot}d+2$.

We claim that for all $i < 4$ and $x > 1$ we have $f^{-1}(4{\cdot}x+i) = 2{\cdot}d+j$ for a $j = 0,1,2$. This is because then $d > 0$ and

$$
\begin{aligned}
&2{\cdot}f(2{\cdot}d+0) = 4{\cdot}d^2+2{\cdot}d+2 < 4{\cdot}d^2+4{\cdot}d+8+2{\cdot}i \leq 2{\cdot}(4{\cdot}x+i) \leq \\
&\quad 4{\cdot}d^2+12{\cdot}d+8+2{\cdot}i \overset{2\cdot i<2\cdot d+6}{<} 4{\cdot}d^2+14{\cdot}d+14 = 2{\cdot}f((2{\cdot}d+2)+1)\ .
\end{aligned}
$$

We can thus define the inverse by a clausal definition using the discrimination on division by 4. The clauses are as follows: $f^{-1}(0) = f^{-1}(1) = 0$, $f^{-1}(2) = f^{-1}(3) = \underline{1}_b$, $f^{-1}(4) = f^{-1}(5) = f^{-1}(6) = \underline{2}_b$, $f^{-1}(7) = \underline{3}_b$ and

$$
\begin{aligned}
f^{-1}(4{\cdot}x+i) = d\mathbf{0} \quad &\leftarrow i < 4 \wedge x > 1 \wedge f^{-1}(x) = d \wedge f(d\mathbf{1}) = a \wedge \\
&\quad 4{\cdot}x+i < a \\
f^{-1}(4{\cdot}x+i) = d\mathbf{1} \quad &\leftarrow i < 4 \wedge x > 1 \wedge f^{-1}(x) = d \wedge f(d\mathbf{1}) = a \wedge \\
&\quad 4{\cdot}x+i \geq a \wedge 4{\cdot}x+i \leq a+d\mathbf{1} \\
f^{-1}(4{\cdot}x+i) = S(d)\mathbf{0} \quad &\leftarrow i < 4 \wedge x > 1 \wedge f^{-1}(x) = d \wedge f(d\mathbf{1}) = a \wedge \\
&\quad 4{\cdot}x+i \geq a \wedge 4{\cdot}x+i > a+d\mathbf{1}
\end{aligned}
$$

The reader will note that $a+d\mathbf{1}+1 = f(2{\cdot}d+2)$.

The following two place *conversion* function is just the pairing function. It is clausally computable by:

$$P_b(x,y) = f(x+y)+x\ .$$

The clauses of P_b will be used for computation only when conversions among numerals will be called for (see Par. 6.20). For instance, the term $(\boldsymbol{s},\boldsymbol{t})+1$ will be computed as $P_b(\boldsymbol{s},\boldsymbol{t})+1$ because the conversion will yield a binary numeral.

The other conversion function is the unary function HT_b which satisfies PA $\vdash x > 0 \to HT_b(x) = H(x), T(y)$, i.e. it is extensionally just an identity on positive numbers. It will be used intensionally to compute pair discrimination when we will be looking for numbers x and y such that $S_i(\boldsymbol{s}) = x,y$ with $i = 0,1$ and $S_i(\boldsymbol{s}) > 0$. The following clausally computable definition of HT_b uses the fact that for $x > 0$ we have $f(f^{-1}(x)) + H(x) = x$ and $f^{-1}(x) = H(x)+T(x)$:

$$
\begin{aligned}
&HT_b(0) = 0,0 \\
&HT_b(x) = h,t \leftarrow x > 0 \wedge f^{-1}(x) = d \wedge f(d) = x+h \wedge d = h+t\ .
\end{aligned}
$$

6.8 Clausal computability of trichotomy and negation discriminations. The modified subtraction function $\dot{-}$ was shown clausally computable in Fig. 2 and we have PA $\vdash x \leq y \leftrightarrow x \dot{-} y = 0$. This means that the term $dich(x,y)$ with the clausally computable function defined in Fig. 2 is a provable case term for the dichotomy discrimination making the latter clausally computable.

Terms $\boldsymbol{s}[\overline{x}]$ and $\boldsymbol{t}[\overline{x}]$ with arbitrary clausally computable terms under conditions $\boldsymbol{C}[\overline{x}]$ can be used in clausally computable discrimination on trichotomy but only under condition $\boldsymbol{C}[\overline{x}]$.

If the characteristic constant P_* of the predicate constant P from discrimination on negation (see Par. 4.1) is computable then the negation discrimination is clausally computable because its provable case term $\underline{2}_b \dot{-} P_*, 0$ is clausally computable.

The identity predicate $x = y$ is clausally computable because its characteristic function $x =_* y$ is:

$$x =_* y = \underline{1}_b \leftarrow x \leq y \land y \leq x \ .$$

The reader will note that the definition discriminates twice on dichotomy and that we have omitted the two default clauses.

The same condition as above applies to the clausal computability of discrimination on negation with P substituted for by the identity $\boldsymbol{s}[\overline{x}] = \boldsymbol{t}[\overline{x}]$.

If P and Q are predicate constants with clausally computable characteristic constants then the P and Q can be used in discrimination on negation and thus also the formula $P \land Q$ can be used in the discrimination on negation because its characteristic constant c is clausally computable by:

$$c = \underline{1}_b \leftarrow P \land Q$$

where we have omitted the two default clauses.

This property is preserved (modulo conditions) when the constants P and Q are substituted for with formulas with clausally computable characteristic functions. Note that if we substitute for P a formula $\boldsymbol{A}[\overline{x}]$ and for Q a formula $\boldsymbol{B}[\overline{x}]$ then c will become a characteristic function $c(\overline{x})$.

Any formula applying clausally computable functions and predicates and composed by boolean connectives can be now easily shown to posses a clasually computable characteristic function. This means that such formulas can be freely substituted for P in discriminations on negations.

6.9 Clausal computability of the least number discrimination. We will now show that in any extension of PA where the the two place predicate P is clausally computable both least number discriminations from Par. 4.12 are clausally computable and they have a provable case term $\underline{1}_b, m(x,0), 0$ where m satisfies the following clausal definition:

$$\begin{array}{lll} m(x,y) = y & \leftarrow P(x,y) & \leftarrow \exists y P(x,y) \land \forall z (P(x,z) \rightarrow y \leq z) \\ m(x,y) = m(x,y+1) & \leftarrow \neg P(x,y) & \leftarrow \exists y P(x,y) \land \forall z (P(x,z) \rightarrow y \leq z) \end{array}$$

There are three facts to prove. The first one is that that the definition of m is legal, the second one that $\underline{1}_b, m(x,0), 0$ is a provable case term for the two discriminations, and the third one that the term is clausally computable under the respective conditions of the two discriminations.

Regarding the first fact, we first show that the definition is regular in the relation $\prec$ introduced into PA by explicit definition:

$$a \prec b \leftrightarrow \exists x \exists y (a = x, y+1 \land b = x, y \land \exists y P(x,y) \land \forall z (P(x,z) \rightarrow y < z))$$

We thus work in PA extended with $\prec$ (but not yet with m) and assume the condition of the definition of m as well as not $P(x,y)$. So if for any z we have $P(x,z)$ it must be the case that $y < z$. But this proves $x, y+1 \prec x, y$ and the above definition satisfies the conditions of regularity. This also proves the condition of applicability for $m(x, y+1)$.

We show next that $\prec$ is provably well-founded:

$$\mathrm{PA}[+R] \vdash \forall b (\forall a{\prec}b\, R(a) \to R(b)) \to R(b)$$

by working in PA$[+R]$ where we assume the antecedent and take any b. If for no a we have $a \prec b$ then $R(b)$ holds from the assumption because $\forall a{\prec}b\, R(a)$ holds vacuously. Otherwise, we have $b = x, y$ for some x, y such that $\exists y P(x,y)$. By the least number principle we have $P(x,v)$ and $\forall z(P(x,z) \to v \leq z)$ for some v for which we must also have $y < v$. By induction on n we prove:

$$\forall w(w + n = v \to R(x,w)) \ . \tag{1}$$

In the base case from $w = v$ we get $R(x,v)$ as above because for no a we have $a \prec x, v$. In the inductive case we assume $w + (n+1) = v$, i.e. $(w+1) + n = v$, and we get $R(x, w+1)$ from IH. Since $w < v$, for any a s.t. $a \prec x, w$ we get $a = x, w+1$. Thus $\forall a{\prec}x, w\, R(a)$ holds and we obtain $R(x,w)$ from the assumption. This ends the proof of (1). From $y < v$ we have $y + n = v$ for some n and so $R(x,y)$ by (1). The above definition of m is now justified.

We now work in PA extended with m and regarding the second fact we show that the term $\underline{1}_b, m(x,0), 0$ is a provable case term for both least number discriminations. The formulas to be proved are in that order:

$$\exists! y P(x,y) \to (P(x,y) \leftrightarrow \underline{1}_b, m(x,0), 0 = \underline{1}_b, y, 0) \ . \tag{2}$$

$$\exists y P(x,y) \to (P(x,y) \land \forall z(P(x,z) \to y \leq z) \leftrightarrow \underline{1}_b, m(x,0), 0 = \underline{1}_b, y, 0) \ . \tag{3}$$

It is easy to see that (3) implies (2). For the proof of (3) we assume $\exists y P(x,y)$ and by the least number principle obtain a v such that $P(x,v)$ and $\forall z(P(x,z) \to v \leq z)$. We then prove

$$\forall x \forall y(b = x, y \land y \leq v \to v = m(x,y)) \tag{4}$$

by assuming $b = x, y$ and $y \leq v$ and proving $v = m(x,y)$ by $\prec$-induction on b where we consider two cases. If $v = y$ then we get $v = m(x,y)$ from the first clause for m. If $y < v$ then $x, y+1 \prec x, y = b$ and we obtain $v \overset{IH}{=} m(x, y+1) = m(x,y)$ from the second clause for m.

With (4) proved we get $v = m(x,0)$ from it and the equivalence in (3) easily follows.

It remains to show the third fact that both discriminations are clausally computable whenever P is. The function m is clausally computable because its discrimination on negation is. It remains to show that the case term for both discriminations is clausally computable. This boils down to the computability of the term $m(x,0)$ under conditions of the discriminations $\exists! y P(x,y)$ and $\exists y P(x,y)$ respectively. The term is computable if the application of m satisfies the applicability condition which is the condition of m with $y = 0$. But that is trivial.

6.10 Clausal computability of bounded quantification. We now show that if P is clausally computable then so is the characteristic function $c(x)$ of bounded quantification $\exists y{<}x\, P(x,y)$ which means that we can use the latter in computable discriminations on negation. We clearly have PA $\vdash \exists y\, \boldsymbol{A}[x,y]$ where $\boldsymbol{A}[x,y] \equiv y < x \wedge P(x,y) \vee y = x$ and so we can use the least number discrimination in the clausal definition of the function c:

$$c(x) = \underline{1}_b \leftarrow \boldsymbol{A}[x,y] \wedge \forall z(\boldsymbol{A}[x,z] \rightarrow y \leq z) \wedge x > y$$

where we have not shown the default clause with $x \leq y$ for the dichotomy discrimination.

It remains to show that c is clausally computable. This boils down to the clausal computability of the characteristic function $\boldsymbol{A}_*$ of $\boldsymbol{A}[x,y]$ which is substituted for the predicate P in the least number discrimination. The function $\boldsymbol{A}_*(x,y)$ is clausally computable as discussed in Par. 6.8.

6.11 Conditional graphs. For a function symbol $f(\overline{x})$ and formulas $\boldsymbol{C}[\overline{x}]$, $\boldsymbol{A}[\overline{x},u]$ of PA and not applying f we say that $\boldsymbol{A}$ *is a graph of* f *under condition* $\boldsymbol{C}$ if

$$\text{PA} \vdash \boldsymbol{C}[\overline{x}] \rightarrow (f(\overline{x}) = u \leftrightarrow \boldsymbol{A}[\overline{x},u]) \ . \qquad (1)$$

6.12 Detours through conditional definitions. We will characterize in Thm. 6.14 the unconditional clausally computable functions as provably recursive functions of PA. Unconditional clausally computable functions are represented by function symbols with unconditional clausal definitions. However, such definitions may use function symbols with conditional definitions. The following theorem is crucial for the elimination of detours through conditionally defined functions.

6.13 Theorem *Clausally computable functions under conditions* $\boldsymbol{C}$ *have* Σ_1 *graphs conditional on* $\boldsymbol{C}$.

Proof. By induction on the construction of clausally computable functions f. If f is basic then $\boldsymbol{C}$ is $\top$. Functions S_0, S_1 (see Par. 4.5) and pairing (see Par. 2.7) have representing definitions of a form $f(\overline{x}) = z \leftrightarrow \boldsymbol{t} = z$ where the identity is Σ_1 in PA. The discrimination functions z, b, and p (see Par. 6.2) are introduced by extensions by definition with Σ_1 in PA formulas. When f is 0 its graph $0 = z$ is provably equivalent to the Σ_1 formula $S(z) \times S(z) = S(z)$.

If f has a representing clausal definition then by Par. 5.7 we can assume w.l.o.g. that the definition is in the normal form. If the definition is recursive then there is a provably well-founded relation $\prec$ in which the recursion descends. otherwise we take $\prec$ to be $<$. We introduce into PA an evaluation predicate Ev similarly as in the proof of Thm. 5.8 where the predicate was used to introduce f into PA (if the definition of f is recursive). If the predicate is already in PA then we use a different name without changing anything else below. The predicate Ev is defined as in Thm. 5.8 except we change the abbreviations for the formulas

$\boldsymbol{P}_m$ when the discrimination formula $\boldsymbol{D}_m$ was formed by a let refinement 5.7(2) with $\boldsymbol{p} \equiv g$. Instead of $\boldsymbol{P}_m[c]$ being 5.8(1) we abbreviate by it the following formula:

$$\exists x \exists z \exists u \exists c_1 (c = \underline{m}, x, u, 0, \underline{g}, x \oplus z, 0, u, c_1 \wedge \boldsymbol{H}[(x)_{\underline{j_1}}, \ldots, (x)_{\underline{j_k}}; z])$$

where $\boldsymbol{H}[x_1, \ldots, x_k, z]$ is a conditional Σ_1 graph of g which exists by IH.

We now prove in PA by $\prec$-recursion on v the formula:

$$\forall \overline{x}(v = x_0, \ldots, x_{l_1} \wedge \boldsymbol{C}[x_0, \ldots, x_{l_1}] \rightarrow \\ (\boldsymbol{D}_1[x_1, \ldots, x_{l_1}; u] \leftrightarrow \exists c\, c \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[x_1, \ldots, x_{l_m}, 0; u]) \ .$$

For that we need to prove by the metatheoretical backward induction on $m \leq s$:

$$\mathrm{PA} \vdash \boldsymbol{C}[x_0, \ldots, x_{l_1}] \wedge \boldsymbol{G}_m[x_0, \ldots, x_{l_m}] \rightarrow \\ (\boldsymbol{D}_m[x_1, \ldots, x_{l_m}; u] \leftrightarrow \exists c\, c \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_m[x_1, \ldots, x_{l_m}, 0; u]) \ .$$

This is straightforward for all m other than those where $\boldsymbol{P}_m$ was changed as above. Because of the applicability condition PA proves under the assumptions in the antecedent $\boldsymbol{C}'[x_{j_1}, \ldots, x_{j_k}]$ where $\boldsymbol{C}'$ is the condition on the clausal definition of g. But then $\boldsymbol{H}[x_{j_1}, \ldots, x_{j_k}, z]$ is equivalent to $g(x_{j_1}, \ldots, x_{j_k}) = z$ and the proof goes through similarly as in Thm. 5.8.

We have thus proved

$$\mathrm{PA} \vdash \boldsymbol{C}[x_0, \ldots, x_{l_1}] \rightarrow (\boldsymbol{D}_1[x_1, \ldots, x_{l_1}; u] \leftrightarrow \exists c\, c \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[x_1, \ldots, x_{l_1}, 0; u])$$

and, since for the first fold of the clausal definition of f we have

$$\mathrm{PA} \vdash \boldsymbol{C}[x_0, \ldots, x_{l_1}] \rightarrow (f(x_0, \ldots, x_{l_1}) = u \leftrightarrow \boldsymbol{D}_1[x_1, \ldots, x_{l_1}; u]) \ ,$$

it is clear that the first part of the theorem will be proved when we show that $\exists c\, c \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[x_1, \ldots, x_{l_1}, 0; u]$ is Σ_1 in PA.

After the elimination of used functions, the above definition of *Ev* still explicitly applies the pairing function (which has a Σ_1 graph by the above), the concatenation function $\oplus$, and the subtree predicate $\sqsubset$. Both have unconditional course of values clausal definition and so they have Σ_1 graphs because functions defined by course of values definitions (see Par. 2.6) from functions with Σ_1 graphs have such graphs.

All explicit existential quantifiers $\exists w$ in the definition of *Ev* can be bounded as $\exists w{<}a$ and the predicate $\sqsubset$ which was defined from its characteristic function with a Σ_1 graph has also also Π_1 graph (because of uniqueness) and thus the right hand side of the definition of *Ev* can be provably in PA brought into the form $\forall c{\leq}a(\boldsymbol{A} \vee \boldsymbol{B})$ with all unbounded quantifiers in $\boldsymbol{A}$ and $\boldsymbol{B}$ existential. We now use the *collection* theorems of PA (see [4]) to swap the existential quantifiers with the bounded universal quantifiers and to bring them into prenex. This makes the formula $\exists c\, c \vdash^{\bullet}_{\boldsymbol{e}} \boldsymbol{D}_1[x_1, \ldots, x_{l_1}, 0; u]$ Σ_1 in PA. $\square$

6.14 Theorem (Characterization of clausally computable functions). *Unconditional clausally computable functions are exactly the provably recursive functions of PA.*

Proof. For the inclusion $\subseteq$ we note that all unconditional clausally computable functions f are by Thm. 5.8 introduced into PA by extension by definition: $f(\overline{x}) = u \leftrightarrow \boldsymbol{A}[\overline{x}, u]$. The formula $\boldsymbol{A}$ is by Thm. 6.13 Σ_1 in PA and so f is provably recursive.

For the inclusion $\supseteq$ we take a provably recursive function f of PA and show it unconditionally clausally computable. Thus PA proves $f(\overline{x}) = u \leftrightarrow \exists\overline{z}\boldsymbol{A}[\overline{z}, \overline{x}, u]$ for some Σ_0 formula $\boldsymbol{A}$. We designate by $\boldsymbol{B}[x, y]$ the following formula

$$\begin{aligned}&\exists x_1{<}x \cdots \exists x_m{<}x \exists u{<}y \exists z_1{<}y \cdots \exists z_m{<}y(\\ &\quad x = x_1, \ldots, x_n, 0 \wedge y = z_1, \ldots, z_m, u \wedge \boldsymbol{A}[z_1, \ldots, z_m, x_1, \ldots, x_n, u])\end{aligned}$$

which is Σ_0 in PA and such that PA proves $\forall x \exists y \boldsymbol{B}[x, y]$. Its two argument characteristic function $\boldsymbol{B}_*(x, y)$ is unconditionally clausally computable by the methods discussed in Paragraphs 6.8 and 6.10 and so and so f provably satisfies the clause

$$f(\overline{x}) = u \leftarrow \overline{x}, 0 = x \wedge \boldsymbol{B}[x, y] \wedge \forall z(\boldsymbol{B}[x, z] \rightarrow y \leq z) \wedge y = u, z$$

obtained by the least number discrimination from Par. 4.12 which is clausally computable by Par. 6.9. Thus also f is clausally computable. □

Evaluation of Clausally Computable Functions

Any computation, be it by a computer, Turing machine, or by hand on a sheet of paper deals with *reduction* (simplification, rewriting) operating on concrete symbols (bits, Turing tape symbols, or symbols written on paper). We will now describe an effective reduction procedure which can be visualized as an effective construction of evaluation trees for discriminated formulas arising in the clausal definitions. We have already seen such trees in an arithmetized form in the proof of the Thm. 5.8 on the admissibility of general clausal definitions. We now lift the trees from PA to the metatheory.

6.15 Mixed numerals. We will compute (reduce) a clausally computable function f in such a way that the evaluation process for arguments $\overline{\boldsymbol{x}}$ given as certain terms, called *irreducible terms*, will terminate with an evaluation tree c and an irreducible term $\boldsymbol{u}$ such that $c \vdash_{\boldsymbol{e}} f(\overline{\boldsymbol{x}}) = \boldsymbol{u}$.

From logical point of view the natural candidates for the irreducible terms are the monadic numerals $\underline{n}$, i.e. $S^n(0)$. However, for complexity reasons it is better that the reduction process works in binary and so we will count the binary numerals (see Par. 4.5) as irreducible terms. In order to make the reduction process efficient (this is done also in our implementation of CL) it is better to take as the irreducible terms the *mixed numerals* which include the pairing terms $\boldsymbol{u}, \boldsymbol{v}$ whenever $\boldsymbol{u}$ and $\boldsymbol{v}$ are irreducible. Mixed numerals are thus generated from

0 by applications of binary successors and pairing. We again exclude the term $0\mathbf{0}$.

Two or more mixed numerals can denote the same number. For instance, the three different mixed numerals $(0,0,0)$, $(0,0\mathbf{1})$, and $0\mathbf{10}$ denote the same number 2.

6.16 Evaluation trees. We will record computations in the form of concretely presented *evaluation trees* which are obtained by the 'de-arithmetization' of the evaluation trees from the proof of Thm. 5.8.

For a given clausally computable function $f(\overline{x})$ represented by a clausal definition in the normal form which is under condition $\boldsymbol{C}[\overline{x}]$ we will be able to construct an evaluation tree c for any n-tuple of mixed numerals $\overline{\boldsymbol{x}}$ such that $\mathcal{M} \models \boldsymbol{C}[\overline{\boldsymbol{x}}]$ holds. We fill also find the mixed numeral $\boldsymbol{u}$ denoting in $\mathcal{M}$ the same as the function value $f(\overline{\boldsymbol{x}})$ by ascertaining that $c \vdash_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{\boldsymbol{x}};\boldsymbol{u}]$ holds. We will be able to evaluate any discriminated formula $\boldsymbol{D}_m$ obtained from the definition by folding. The evaluation satisfies the following *soundness* property for any mixed numerals $\overline{\boldsymbol{x}}$, $\overline{\boldsymbol{y}}$, and $\boldsymbol{u}$:

$$\begin{aligned}&\text{if } \mathcal{M} \models \boldsymbol{C}[\overline{\boldsymbol{x}}] \land \boldsymbol{G}_m[\overline{\boldsymbol{x}},\overline{\boldsymbol{y}}] \text{ and } c \vdash_{\boldsymbol{e}} \boldsymbol{D}_m[\overline{\boldsymbol{x}},\overline{\boldsymbol{y}};\boldsymbol{u}] \text{ then}\\ &\qquad\qquad \mathrm{PA} \vdash \boldsymbol{C}[\overline{\boldsymbol{x}}] \land \boldsymbol{G}_m[\overline{\boldsymbol{x}},\overline{\boldsymbol{y}}] \to \boldsymbol{D}_m[\overline{\boldsymbol{x}},\overline{\boldsymbol{y}};\boldsymbol{u}]\ .\end{aligned} \tag{1}$$

The relation $c \vdash_{\boldsymbol{e}} \boldsymbol{D}_m[\overline{\boldsymbol{x}},\overline{\boldsymbol{y}};\boldsymbol{u}]$ can be thus viewed as a 'is a proof of' relation and so we will define it below in the usual way by rules of inference. We write as usual $\vdash_{\boldsymbol{e}} \boldsymbol{D}_m[\overline{\boldsymbol{x}},\overline{\boldsymbol{y}};\boldsymbol{u}]$ when $c \vdash_{\boldsymbol{e}} \boldsymbol{D}_m[\overline{\boldsymbol{x}},\overline{\boldsymbol{y}};\boldsymbol{u}]$ holds for an evaluation tree c.

6.17 Conditionally computable functions. We say that a function f introduced into PA by a clausal definition under the condition $\boldsymbol{C}[\overline{x}]$ is *conditionally computable* if for any n-tuple of mixed numerals $\overline{\boldsymbol{x}}$ we have:

$$\text{if } \mathcal{M} \models \boldsymbol{C}[\overline{\boldsymbol{x}}] \text{ then } \vdash_{\boldsymbol{e}} f(\overline{\boldsymbol{x}}) = \boldsymbol{u} \text{ for some mixed numeral } \boldsymbol{u}\ . \tag{1}$$

Most clausal definitions of f have conditions $\top$, and such function are *computable* if they are conditionally computable. Note that then the above condition of computability simplifies to:

> For any n-tuple of mixed numerals $\overline{\boldsymbol{x}}$ there is a mixed numeral $\boldsymbol{u}$ such that $\vdash_{\boldsymbol{e}} f(\overline{\boldsymbol{x}}) = \boldsymbol{u}$.

6.18 Evaluation rules for basic functions. For the computation of basic functions we have four groups of rules. The first group specifies the evaluations of mixed numerals, the remaining three describe in that order the evaluation of zero, binary, and pair discrimination functions:

$$\frac{}{\boldsymbol{x} = \boldsymbol{x}} \qquad \frac{}{0\mathbf{0} = 0} \qquad \boldsymbol{x} \text{ a mixed numeral}$$

$$\frac{}{z(0) = 0, 0} \qquad \frac{}{z(\boldsymbol{x}) = \underline{1}_b, 0} \qquad \boldsymbol{x} \text{ a mixed numeral other than } 0$$

$$\frac{}{b(0) = 0, 0, 0} \qquad \frac{}{b(S_i(\boldsymbol{x})) = \underline{i}_b, \boldsymbol{x}, 0} \qquad \frac{P_b(\boldsymbol{v_1}, \boldsymbol{v_2}) = S_i(\boldsymbol{u})}{b(\boldsymbol{v_1}, \boldsymbol{v_2}) = \underline{i}_b, \boldsymbol{u}, 0} \star_1$$

for $i = 0, 1$ and any mixed numerals $S_i(\boldsymbol{x})$, $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$.

$$\frac{}{p(0) = 0, 0} \quad \frac{}{p(\boldsymbol{v_1}, \boldsymbol{v_2}) = \underline{1}_b, \boldsymbol{v_1}, \boldsymbol{v_2}, 0} \quad \frac{HT_b(S_i(\boldsymbol{x})) = \boldsymbol{v_1}, \boldsymbol{v_2}}{p(S_i(\boldsymbol{x})) = \underline{1}_b, \boldsymbol{v_1}, \boldsymbol{v_2}, 0} \star_2$$

for $i = 0, 1$ and any mixed numerals $S_i(\boldsymbol{x})$, $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$.

For the discussion of the *conversion* rules $\star_1$ and $\star_2$ see Par. 6.20.

6.19 Evaluation rules for clausally defined functions. Every clausally defined n-ary function f (conditionally or not) is computed from its normalized definition (see Par. 5.7). Applications of f are evaluated from its main discrimination formula:

$$\frac{\boldsymbol{D}_1[\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1}; \boldsymbol{u}]}{f(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1}) = \boldsymbol{u}}$$

The constituent discrimination formulas $\boldsymbol{D}_m[x_0, \ldots, x_{j-1}; u]$ of $\boldsymbol{D}_1$ are evaluated according to their formation in that order by a value 5.7(1), let 5.7(2), or the discrimination 5.7(3) refinement as follows:

$$\frac{}{\boldsymbol{D}_m[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_{l_m}; \boldsymbol{x}_i]}$$

$$\frac{\boldsymbol{p}(\boldsymbol{x}_{j_1}, \ldots, \boldsymbol{x}_{j_k}) = \boldsymbol{x}_{l_q} \qquad \boldsymbol{D}_q[\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{l_q}; \boldsymbol{u}]}{\boldsymbol{D}_m[\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{l_m}; \boldsymbol{u}]}$$

$$\frac{\boldsymbol{D}_{q_a}[\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{l_m}, \ldots, \boldsymbol{x}_{l_{q_a}}; \boldsymbol{u}]}{\boldsymbol{D}_m[\boldsymbol{x}_0, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_{l_m}; \boldsymbol{u}]} \ \boldsymbol{x}_i \equiv \underline{a}_b, \boldsymbol{x}_{l_m+1}, \ldots, \boldsymbol{x}_{l_{q_a}}, 0 \text{ for } a = 1, \ldots, p \star_3$$

For the discussion of possible conversions of mixed numerals in the rule $\star_3$ see Par. 6.20.

6.20 Conversions of mixed numerals. The rules $\star_1$ for binary and the rules $\star_2$ for pair discriminations in Par. 6.18 call functions which convert mixed numerals to binary numerals. Clauses for both functions are given in Par. 6.7.

The binary function P_b is just the pairing function but its clauses are written solely by discrimination on zero and binary. One can prove:

> For any mixed numerals $\boldsymbol{v}$, $\boldsymbol{w}$ there is an $i > 0$ such that $\vdash_{\boldsymbol{e}} P_b(\boldsymbol{v}, \boldsymbol{w}) = \underline{i}_b$.

By the soundness property 6.16(1), the binary numeral $\underline{i}_b$ denotes the result of pairing of (the denotations of) the two mixed numerals. The reader will note that the evaluation tree for the last identity may contain nodes constructed by rules $\star_1$. The rules trigger nested subevaluations of the function P_b but to lesser arguments. This happens when the mixed numerals $\boldsymbol{v}$ and $\boldsymbol{w}$ contain pairing.

The clausal definition of the unary function $HT_b(x)$ uses the zero and binary discriminations to access its argument. For reasons of efficiency the function yields a pair consisting of both projections of x. One can then prove:

For any mixed numeral $S_i(\boldsymbol{v})$ there are mixed numerals $\boldsymbol{u_1}$, $\boldsymbol{u_2}$ such that $\vdash_{\boldsymbol{e}} HT_b(S_i(\boldsymbol{v})) = \boldsymbol{u_1}, \boldsymbol{u_2}$.

As there is no pairing discrimination in the clauses for HT_b the evaluation tree for the above identity will not use the rules $\star_2$. On the other hand, the tree may use the rules $\star_1$ because the mixed numeral $S_i(\boldsymbol{v})$ may contain pairing.

The third type of conversion among mixed numerals is involved in the application of the discrimination rules $\star_3$ from Par. 6.19. It is guaranteed by the formation rules for clausal definitions that the mixed numeral $\boldsymbol{x}_i$ is such that we have in the standard model $\mathcal{M} \vdash \boldsymbol{x}_i = \underline{a}_b, \boldsymbol{x}_{l_m+1}, \ldots, \boldsymbol{x}_{l_{q_a}}, 0$ for exactly one a and unique numbers $x_{l_m+1}, \ldots, x_{l_{q_a}}$. Unless the discrimination is zero, binary, or pair we have no guarantee that the numerals have the correct shape, i.e that $\boldsymbol{x}_i \equiv \underline{a}_b, \boldsymbol{x}_{l_m+1}, \ldots, \boldsymbol{x}_{l_{q_a}}, 0$. The shape of the numerals depends on the definitions of functions constructing the appropriate case term. It is always possible to define the functions such that the value of the discrimination term will be always in the expected shape. This can be guaranteed for instance by typing of clausal definitions. If the typing is not enforced the discrimination rules should be viewed as rules with three premises:

$$\frac{\boldsymbol{d}(\boldsymbol{x}_i) = \underline{a}_b, \boldsymbol{w} \quad \boldsymbol{L}_{l_{q_a}-l_m}(\boldsymbol{w}) = \boldsymbol{x}_{l_m+1}, \ldots, \boldsymbol{x}_{l_{q_a}}, 0 \quad \boldsymbol{D}_{q_a}[\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{l_m}, \ldots, \boldsymbol{x}_{l_{q_a}}; \boldsymbol{u}]}{\boldsymbol{D}_m[\boldsymbol{x}_0, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_{l_m}; \boldsymbol{u}]}$$

where the clausal definitions of the shape forcing functions $\boldsymbol{B}$, $\boldsymbol{d}$ and $\boldsymbol{L_n}$ for each $n \geq 0$. are given in Fig. 3. The reader will note that all functions f are extensionally identities $f(x) = x$ because their sole role is to force conversion (if needed) of mixed numerals. Their definitions use only the basic discriminations and their main property for any $n \geq 0$ and a mixed numeral $\boldsymbol{v}$ is:

if $\mathcal{M} \vDash \exists a \exists y_1, \ldots, \exists y_n\, \boldsymbol{v} = a, y_1, \ldots, y_n, 0$ then $\vdash_{\boldsymbol{e}} \boldsymbol{L_n}(\boldsymbol{d}(\boldsymbol{v})) = \underline{a}_b, \boldsymbol{u_1}, \ldots, \boldsymbol{u_n}, 0$ for some $a \in \mathbb{N}$ and mixed numerals $\boldsymbol{u_1}, \ldots, \boldsymbol{u_n}$.

6.21 Theorem *Clausally computable functions are conditionally computable.*

Proof. We first prove the three conversion properties from Par. 6.20. We then use a triple induction with the outer induction on the construction of the clausally computable functions where we prove that each such function f is conditionally computable. Conditional computability of f is the existence property $\vdash_{\boldsymbol{e}}^{\bullet} \boldsymbol{D}_m[x_0, \ldots, x_{l_m}, 0; u]$ from the property 5.8(4) lifted up into the metatheory. Its

$$
\begin{aligned}
&\boldsymbol{B}(x\mathbf{0}) = 0 && \leftarrow x = 0\\
&\boldsymbol{B}(x\mathbf{0}) = \boldsymbol{B}(x)\mathbf{0} && \leftarrow x > 0\\
&\boldsymbol{B}(x\mathbf{1}) = \boldsymbol{B}(x)\mathbf{1} && \leftarrow x > 0\\
&\boldsymbol{d}(0) = 0\\
&\boldsymbol{d}(i, w) = \boldsymbol{B}(i), w\\
&\boldsymbol{L_0}(x) = 0\\
&\boldsymbol{L_{n+1}}(0) = 0\\
&\boldsymbol{L_{n+1}}(v, w) = v, \boldsymbol{L_n}(w)\ .
\end{aligned}
$$

Fig. 3. Clausal definitions of auxiliary functions forcing correct shape of discrimination terms.

proof proceeds by double induction ($\prec$-induction and induction on the construction of $\boldsymbol{D}_m$) exactly as in Thm. 5.8 except that we use the outermost IH to compute the functions g used in the definition of f. $\square$

6.22 Effectivity of evaluation. For an unconditional clausal definition of a function f we take the main discriminated formula $\boldsymbol{D}_1[\overline{x}; u]$ from the normalized clausal definition and we know from the proof of Thm. 6.21 that to every tuple of mixed numerals $\overline{\boldsymbol{x}}$ on input there is a mixed numeral $\boldsymbol{u}$ on output such that $c \vdash_{\boldsymbol{e}} \boldsymbol{D}_1[\overline{\boldsymbol{x}}, \overline{u}]$ for some evaluation tree c. The proof of Thm. 6.21 is constructive and it gives an effective procedure for the construction of c and $\boldsymbol{u}$.

This procedure can be carried out by the construction of a partially constructed evaluation tree for $\boldsymbol{D}[\overline{\boldsymbol{x}}, u]$ with the output numeral left undetermined. The leftmost branch of the tree ends in a rule without a premise which gives the value of the output variable and the value is propagated downwards and to the next branch on the right for the rules with more premises. In this way we can contruct the tree c from left to right and on the final way down determine the output numeral $\boldsymbol{u}$. Of course, computer implementations will translate this process into an effective linear reduction sequence.

The construction of clausal definitions guarantees that the applicability conditions for functions g applied in f are satisfied (in the standard model $\mathcal{M}$) and so the definitions can be effectively computed by the above process.

The only problem is when the outermost function f has a conditional clausal definition under an undecidable condition $\boldsymbol{C}$. For a tuple of mixed numerals $\overline{\boldsymbol{x}}$ on input we generally do not know whether $\mathcal{M} \models \boldsymbol{C}[\overline{\boldsymbol{x}}]$ holds and unless we can prove $PA \vdash \boldsymbol{C}[\overline{\boldsymbol{x}}]$ and we can only initiate the evaluation process in the hope that it will terminate. If it terminates on no evaluation rule being applicable then we know that the condition does not hold. More probably the construction will not stop because of some infinite evaluation tree due to non-terminating recursion (either of f itself or of a nested g whose applicability condition was not satisfied).

6.23 Well-typed programs do not need conversions. The language of clausal definitions is typeless. There are many good reasons why we sometimes want to work outside of sometimes constraining type straightjacket. For instance, when we define a function we do not wish to compute with because the function is needed to specify or to prove properties of our programs.

The standard argument for types that they help catching **some** programming errors is not as important in the presence of a theorem prover because proving programs correct catches **all** errors.

There is, however, a very good reason why we should type those clausal definitions which we intend to compute with. Correctly typed programs do not need the conversion rules $\star_1$, $\star_2$, $\star_3$ during their evaluations. This is because the typing keeps separated binary numerals (which we intend to use in arithmetic operations) from 'data structures' which are usually lists and trees containing binary numerals. The data structures are constructed and destructed by pairing.

Typing guarantees that the evaluation produces mixed numerals of certain shapes. For instance, if a function is typed to yield a cartesian of n-components then its evaluation yields a mixed numeral $\boldsymbol{u_1}, \ldots, \boldsymbol{u_n}$ and we can efficiently compile the function into, say, C^{++} with locations for results defined by a structure of C.

7 Conclusions

The current implementation of CL syntactically analyzes the clauses of a definition by a complicated parsing algorithm. To make it worse, it is hard to describe the syntax of clauses directly without the concept of unfolding. The clauses of a massively extensible language described in this paper are almost impossible to parse. This means that a definition of a function must be prompted from its discriminated formula. The formula is constructed in a top down fashion by a refinement process. Something similar is already done when one constructs proofs in the theorem prover. As can be seen from our web-browser implementation of CL [2], a good and flexible system of prompters (forms) makes the construction by prompting of a complex object (discriminated formula or a proof) far easier than by writing it down in its entirety.

We have seen how the proofs of theorems $\boldsymbol{A}[f]$ in $\mathrm{PA}[+f]$ act as schemes where we obtain in a uniform way a proof of theorems $\boldsymbol{A}[g]$ for many different functions g. Such schemes of proofs play the role of universal quantification over functions because proving $\forall f \boldsymbol{A}[f]$ in a system of a second-order arithmetic achieves the same effect.

Schemes of proofs cannot simulate the role of existential quantification over second-order objects. Suppose, for instance, that we have proved in a system of second-order arithmetic the formula

$$\forall g(\boldsymbol{A}[g] \rightarrow \exists f \boldsymbol{B}[g, f]) \ .$$

We can extract from the proof a uniform definition of f in terms of g. Hence, for any concrete function g satisfying $\boldsymbol{A}[g]$ we can automatically obtain a definition of f satisfying the specifications $\boldsymbol{B}[g, f]$. This is called *reusability* or *modularization* in computer programming. The gains of this are so obvious that we would like to replace in a future version of CL the first-order Peano Arithmetic with a system of second-order arithmetic. We are contemplating the weakest such system: 'Predicative second-order arithmetic with arithmetical comprehension'. If we formulate this system with a single induction axiom then the system is easily seen conservative over PA and so its provably recursive functions are exactly as in PA, i.e the $< \epsilon_0$-recursive functions.

References

1. C, Mc Bride and J. Mc Kinna, *The view from the left.* Journal of Functional Programming 2003.
2. *The programming language and proof system CL*, www.ii.fmph.uniba.sk/cl/.
3. M. Davis, *Computability and Unsolvability*, McGraw Hill, New York, 1958.
4. P. Hájek and P. Pudlák. *Metamathematics of First-Order Arithmetic.* Springer Verlag, 1993.
5. T. Nipkow and L. C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS 2283, Springer, 2002.
6. S. Owre, N. Shankar . *The formal semantics of PVS.* Tech. Report Computer Science Laboratory, SRI International, www.csl.sri.com/papers/csl-97-2, 1997.
7. H. E. Rose. *Subrecursion: Functions and Hierarchies.* Oxford Logic Guides 9. Clarendon Press, Oxford, 1982.
8. J. R. Shoenfield. *Mathematical Logic*, Addison-Wesley, 1967.
9. W. W. Tait. On nested recursion. *Mathematische Annalen*, vol. 143 (1961), pages 236-250.
10. P. J. Voda. *Subrecursion as a basis for a feasible programming language*, in L. Pacholski and J. Tiuryn, editors, Proceedings of CSL'94, LNCS 933, Springer Verlag, 1995, pages 324-338.
11. P. J. Voda. *Metamathematics of Computer Programming,* www.ii.fmph.uniba.sk/∼voda, 2001.
12. P. J. Voda. *Recursion and Coding in PA by Techniques of Computer Programming,* www.ii.fmph.uniba.sk/∼voda, 2002.
13. P. J. Voda, *Predicative Justification and Development of Theory of Finite Sets*, www.fmph.uniba.sk/∼voda, 2003.