Automatic Verification Based on Abstract Interpretation

Mizuhito Ogawa[1]

NTT Communication Science Laboratories
3-1 Morinosato-Wakamiya Atsugi Kanagawa, 243-0198 Japan
mizuhito@theory.brl.ntt.co.jp
http://www.brl.ntt.co.jp/people/mizuhito/

Abstract. This paper reconstructs and extends the automatic verification technique of *Le Métayer, Proving properties of programs defined over recursive data structures (ACM PEPM '95)*, based on a backward abstract interpretation.
To show the effectiveness of extensions, we show two examples of the declarative specifications of sorting and formatting programs, which are directly and concisely expressed in our specification language.

1 Introduction

Program errors cause failures during execution that can be classified into three categories.

1. Execution eventually stops as a result of illegal operations.
2. Execution does not terminate.
3. Execution results are not what was intended.

Errors of the first kind are detected by type inference, with such languages as ML. In addition, although termination is in general undecidable, errors of the second kind can be automatically prevented by several techniques, such as simple termination [12, 13], termination analysis [16], and dependency pairs [2].

The third kind of error cannot be prevented without a specification language, and there is always a trade-off between expressiveness and feasibility. If the aim is to express everything, it is easy to fall into the trap of undecidability. Moreover, too much expressiveness may make users hard to learn. For compile-time error detection, an automatic verifier that functions without any human guidance is desirable even if it verifies only partial specifications. Then the user can concentrate on what kind of properties, under the limitation of a simple and restricted specification language, properly approximate the program behavior.

By restricting both properties and languages, Le Métayer developed an automatic verification technique [19]. Its target language is a strongly-typed first-order functional language with product types and recursive types. The important restriction is that the conditional part of an if-expression contains only basic predicates (such as *null*, *leq*, *geq*, and *equal*) without any functional symbols.

He defines a language which prescribes a class of *uniform* predicates over recursive types. These predicates are constructed by predicate constructors from basic predicates on base types. As an example, his system expresses that a sort program returns a list of decreasing values (if the sort program terminates) and automatically verifies it. This property is called *orderedness* of the sort program, which is expressed by $\mathbf{true} \rightarrow \nabla geq(sort\ X)$ in our specification language. Note that the termination of the sort program is not verified; this verification is left to a termination analysis.

Similar ideas to those of uniform predicates are also found in Refs. [15, 17, 3, 21]; however, the significant differences are that

- binary predicates are allowed in constructing predicates, and
- free variables in binary predicates are allowed.

The former extends the expressiveness of target properties from other flow analyses. The latter maintains the power of inter-functional inferences. However, the expressive power of the specification language is still fairly restricted as a verification; for instance, the input-output relation cannot be described.

This paper reconstructs and extends the automatic verification technique of Le Métayer [19] based on a backward abstract interpretation [11, 1, 7]. The termination and soundness proofs of the verification are naturally derived from the formalization as a backward abstract interpretation.

Extensions are achieved by (1) using the input variable in function properties, (2) introducing new predicate constructors, and (3) using uninterpreted function/predicate symbols. They are demonstrated by verifying the sorting and formatting programs. The first and the second extensions expand the ability of the specification language so that it covers another major specification of the sorting program; namely, *weak preservation*, i.e., the input and the output are the same *set*. This is expressed by $\mathbf{true} \rightarrow \forall_l \exists_r equal \wedge \forall_r \exists_l equal(sort\ X)$. Note that since our specification language cannot express the number of elements in a list, our algorithm cannot detect the full specification of sort, called *preservation*, i.e., the input and the output are the same *multiset*.

The third extension expands the range of both target programs and the specification language. The expansion of target programs is achieved by loosening the restrictions on the conditional part of an if-expression. The running example is `format`, which formats a given sentence (expressed as a list of strings) to a specified width. The technique behind this extension is the use of *uninterpreted functions*. We also show how partial evaluation will cooperate with the verification. Other major specifications of `format` become expressible by the use of *uninterpreted predicates*. This technique drastically expands the expressive ability, such as the specification that the order of words is preserved by `format`.

This paper is organized as follows: Section 2 defines programming and specification languages. Section 3 provides the verification algorithm based on a backward abstract interpretation. The termination and soundness proofs are also given. Section 4 demonstrates the verification of *orderedness* of the (simple but inefficient) sort program to explain the algorithm. Section 5 presents extensions

and demonstrates the verification of major specifications of the sorting and formatting programs. Section 6 discusses related work and Section 7 concludes the paper and discusses future work.

2 Preliminaries

2.1 Programming Language

The target language is a strongly-typed first-order functional language with ML-like syntax, in which product types and recursive types, such as lists $list(A) = \mu\alpha.nil + A \times list(\alpha)$, are allowed. We use :: to mean infix *cons*, @ to mean infix *append*, and [ ] to mean a list, namely, $[a_1, a_2, a_3] = a_1 :: (a_2 :: (a_3 :: nil))$. The semantics of the language is given by an ordinary least fix-point computation. We assume that the language is strict, but the same technique can be applied to a lazy language as well. The precise syntax and semantics of the language are shown in Fig. 1 and Fig. 2. Parentheses in the syntax are used for either making pairs or clarifying the order of applications of infix operators. Basic concrete domains D_{Bool} and D_{Int} are flat cpo's (as usual), and the other concrete domains D_α of type α are constructed by the list and pair constructors. The interpretation ψ of expressions has the hidden argument, i.e., for simplicity the environment fve of function variables are omitted in Fig. 2.

The language of expressions

$$
\begin{aligned}
E = {} & x \mid C \mid (E_1, E_2) \mid E_1 :: E_2 \mid f\ E \mid op\ E \mid (E) \mid \\
& \texttt{if}\ Cond\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \mid \texttt{let val}\ x = E_1\ \texttt{in}\ E_2\ \texttt{end} \mid \\
& \texttt{let val}\ (x, y) = E_1\ \texttt{in}\ E_2\ \texttt{end} \mid \texttt{let val}\ x :: xs = E_1\ \texttt{in}\ E_2\ \texttt{end} \\
Cond = {} & p_u\ x \mid p_b\ (x, y)
\end{aligned}
$$

$$
\text{where} \begin{cases} E \in Exp & \text{expressions} \\ C \in Const & \text{constants} \\ x \in Bv & \text{bound variables} \end{cases} \qquad \begin{cases} op \in Prim & \text{primitive functions} \\ p_u, p_b \in Pred & \text{basic predicates} \\ f \in Fv & \text{functional variables} \end{cases}
$$

The syntax of programs $\quad Prog = \{\ \texttt{fun}\ f_i\ x_i = E_i\ ;\ \}$

The language of types

$$
\begin{aligned}
T &= T_G \mid T_F & T_F &= T_G \to T_G \\
T_G &= T_U \mid T_P \mid T_R & T_P &= T_G \times T_G \\
T_R &= \mu\alpha.\texttt{nil} + T_G :: \alpha & T_U &= \tau \quad \text{(basic types)}
\end{aligned}
$$

Fig. 1. Syntax of Programming Language.

$$
\begin{array}{ll}
\varphi[\![\ \{\ \texttt{fun}\ f_i\ x_i = E_i\ ;\ \}\ ]\!] & = fve \text{ whererec} \\
 & \quad fve = [(\lambda y_1 \cdots y_n .\mathit{if}\ (\mathit{bottom}?\ y_1 \cdots y_n) \\
 & \qquad \mathit{then}\ \bot\ \mathit{else}\ \psi[\![E_i]\!][y_j/x_j])/f_i] \\
\psi[\![C]\!]bve & = \xi_c[\![C]\!] \\
\psi[\![x]\!]bve & = bve[\![x]\!] \\
\psi[\![op\ E]\!]bve & = \xi_f[\![op]\!](\psi[\![E]\!]bve) \\
\psi[\![p_u\ x]\!]bve & = \xi_p[\![p_u]\!](bve[\![x]\!]) \\
\psi[\![p_b\ (x,y)]\!]bve & = \xi_p[\![p_b]\!](bve[\![x]\!], bve[\![y]\!]) \\
\psi[\![f\ E]\!]bve & = fve[\![f]\!](\psi[\![E]\!]bve) \\
\psi[\![(E_1, E_2)]\!]bve & = (\psi[\![E_1]\!]bve, \psi[\![E_2]\!]bve) \\
\psi[\![E_1 :: E_2]\!]bve & = (\psi[\![E_1]\!]bve) :: (\psi[\![E_2]\!]bve) \\
\psi[\![\texttt{if}\ Cond\ \texttt{then}\ E_1\ \texttt{else}\ E_2]\!]bve & = \mathit{if}\ (\mathit{bottom}?\ (\psi[\![Cond]\!]bve))\ \mathit{then}\ \bot \\
 & \quad \mathit{elsif}\ \psi[\![Cond]\!]bve\ \mathit{then}\ \psi[\![E_1]\!]bve\ \mathit{else}\ \psi[\![E_2]\!]bve \\
\psi[\![\texttt{let val}\ x = E_1\ \texttt{in}\ E_2]\!]bve & = \psi[\![E_2]\!](bve[\psi[\![E_1]\!]bve/x]) \\
\psi[\![\texttt{let val}\ (x,y) = E_1\ \texttt{in}\ E_2]\!]bve & = \psi[\![E_2]\!](bve[\psi[\![E_1]\!]bve/(x,y)]) \\
\psi[\![\texttt{let val}\ x :: xs = E_1\ \texttt{in}\ E_2]\!]bve & = \psi[\![E_2]\!](bve[\psi[\![E_1]\!]bve/x :: xs]) \\
\mathit{bottom}?\ y_1 \cdots y_n & = (y_1 = \bot) \vee \cdots \vee (y_n = \bot)
\end{array}
$$

$$
\text{where} \begin{cases}
\psi : (Fve \rightarrow) Exp \rightarrow Bve \rightarrow D & \xi_f : Prim \rightarrow D \rightarrow D \\
\psi_p : Prog \rightarrow Fve & \xi_p : Pred \rightarrow D \rightarrow Bool \\
fve \in Fve = Fv \rightarrow D \rightarrow D & \xi_c : Const \rightarrow D \\
bve \in Bve = Bv \rightarrow D &
\end{cases}
$$

Fig. 2. Semantics of Programming Language.

An important restriction is that the conditional part of an if-expression must consist only of basic predicates without any functional symbols. Section 5 discusses how this requirement can be loosened. Until then, we use only null as the unary basic predicate on lists, leq, geq, and equal as the binary basic predicates on integers, :: as a binary primitive function, and *nil* as a constant. The type description in a program is often omitted if it can be easily deduced by type inference.

For technical simplicity, we also set the following restrictions.

- Basic types are *Int* and *Bool*.
- Product types and recursive types are pairs and lists, respectively.
- Each function is unary, i.e., pairs must be used to compose variables.

The third restriction means that binary functions and predicates are respectively regarded as unary functions and predicates which accept the argument of pair

type. This assumption can easily be extended to a more general setting, for instance, with tuple types.

Values are denoted by $a, b, c, \cdots$, lists by $as, bs, cs, \cdots$, and lists of lists by $ass, bss, css, \cdots$.[1] We also assume that variable names of input variables and locally defined variables are different. The following functions present a sorting program with types $sort : int\ list \rightarrow int\ list$ and $max : int\ list \rightarrow int \times int\ list$.

```
fun sort as = if null as then nil else
                 let val (b,bs) = max as in b::sort bs end;

fun max cs = let val d::ds = cs in
               if null ds then (d,nil) else
                  let val (e,es) = max ds in
                   if leq(e,d) then (d,e::es) else (e,d::es) end
             end;
```

2.2 Specification Language

The language for specifying properties is constructed by using predicate constructors $\forall, \forall_l, \forall_r$, and ∇ on basic predicates, constants, free variables, and variables appearing in a program. Predicate constructors will be extended in Section 5. A basic unary predicate is denoted by p_U, a basic binary predicate by p_B, a unary predicate by P_U, and a binary predicate by P_B. Indexes U and B are often omitted when they are clear from the context. As convention, bound variables are denoted by $a, b, c, \cdots, x, y, z, \cdots, as, bs, cs, \cdots, xs, ys, zs, \cdots$, free variables by $X, Y, Z, \cdots$, constants by $C, M, \cdots$, and expressions by $E, E_1, E_2, \cdots$.

A binary predicate P is transformed into a unary predicate P^E by substituting an expression E for the second argument. That is, $P^E(E') = P(E', E)$. $\bar{P}$ is defined by $\bar{P}(E_1, E_2) = P(E_2, E_1)$. The grammar of the construction of predicates is shown in Fig. 3. Specification of a function f is expressed with a free variable by $Q(X) \rightarrow P(f\ X)$, which means if input X satisfies Q then output $f\ X$ satisfies P, when $P, Q \in P_G^-$. Each input X is a distinct free variable for each function and property to avoid name crash.

Note that negation is not allowed. The meanings and examples of the predicate constructors $\forall, \forall_l, \forall_r$, and ∇ are given as follows.

- $\forall P_U(xs)$ iff either xs is nil or the unary predicate $P_U(x)$ holds for each element x in xs.
- $\nabla P_B(xs)$ iff either xs is nil or $\forall \bar{P}_B^y\ (ys) \wedge \nabla P_B(ys)$ for $xs = y :: ys$.

$\forall_l P_B(xs, y)$ and $\forall_r P_B(x, ys)$ are defined by $\forall P_B^y(xs)$ and $\forall \bar{P}_B^x(ys)$, respectively. The examples are shown in the table below.

[1] as is a reserved word of ML, but we ignore it.

P_F	$= P_G^-\ X \rightarrow P_G^-\ (f\ X)$	properties of functions
P_G^-	= predicates in P_G without bound variables	
P_G	$= P_S \mid P_P \mid \mathbf{true} \mid \mathbf{false}$	ground properties
P_S	$= P_U \mid P_R$	unary predicates
P_R	$= \forall P_U \mid \nabla P_G \mid P_P^E \mid P_R \wedge P_R \mid P_R \vee P_R$	properties of lists
P_P	$= P_B \mid \bar{P}_P \mid P_S \times P_S \mid \forall_r P_P \mid \forall_l P_P \mid$ $P_P \wedge P_P \mid P_P \vee P_P$	properties of pairs
P_B	$= p_b \mid \bar{P}_B \mid P_B \wedge P_B \mid P_B \vee P_B$	basic binary predicates
P_U	$= p_u \mid P_B^X \mid P_U \wedge P_U \mid P_U \vee P_U$	basic unary predicates
E		expressions
V	$= X \mid x \mid c$	basic expressions
X		(finitely many) free variables
x		bound variables
C		constants

Fig. 3. Language for specification of properties.

predicate	*true*	*false*
$geq^3(a)$	4	2
$\forall geq^3(as)$	[3,6,4], nil	[3,6,1]
$\forall_l leq(as, a)$	([3,6,4], 8), (nil, 8)	([3,6,4], 5)
$\forall_r leq(a, as)$	(3, [3,6,4]), (3, nil)	(5, [3,6,4])
$\nabla geq(as)$	[6,4,3], nil	[4,6,3]

For instance, the sorting program is fully specified by the following conditions.

1. Output must be ordered (called **orderedness**).
2. Input and output are the same *multiset* (called **preservation**).

Orderedness is expressed as $\mathbf{true} \rightarrow \nabla geq(sort\ X)$. That is, the output of the sorting program is decreasing if the input satisfies **true** (i.e., empty assumptions). For preservation, the weaker condition called weak preservation, i.e., the input and the output are the same *set*, is expressed by

$$\mathbf{true} \rightarrow (\forall_l \exists_r equal \wedge \forall_r \exists_l equal)^X (sort\ X).$$

with the introduction of additional predicate constructors $\exists, \exists_l$, and $\exists_r$ (which intuitively mean $\neg\forall\neg, \neg\forall_l\neg$, and $\neg\forall_r\neg$, respectively), which will be discussed in Section 5. Note that only the input variable of a function remains in the scope when a function call occurs, thus our definition of properties of functions (P_F) is possible (which was neglected in [19]).

3 Automatic Verification as Abstract Interpretation

3.1 Verification Algorithm as Abstract Interpretation

An abstract interpretation consists of an abstract domain, its order, and an interpretation (on an abstract domain) of primitive functions [11, 1, 7]. Our choice is a backward abstract interpretation with

abstract domain	a set of predicates (in Fig. 3) satisfying the type,
order	the entailment relation defined in Fig. 4, and
interpretation	defined in Fig. 5.

Let { `fun` $f_i\ x_i = E_i$; } be a program. The verification algorithm is the least fixed point computation to solve *whererec* equations in Fig. 5. Section 4 explains how this algorithm performs on the sorting program.

The entailment relation $\sqsubseteq$ (in Fig. 4) is intuitively the opposite of the logical implication. That is, $P \sqsubseteq Q$ means that Q implies P. By definition, **true** is the least element and **false** is the greatest element in the abstract domain. We denote by $P \equiv Q$ if $P \sqsubseteq Q$ and $P \sqsupseteq Q$. The entailment relation may be used to trim at each step of interpretation Ψ. Formally, the entailment relation consists of axioms on basic predicates/predicate constructors, and ordinary logical rules, and their extensions by predicate constructors, as defined in Fig. 4.

In Fig. 5, let *Formula* be a set of all formulae constructed from predicates in Fig. 3 and bound variables (in the scope of an expression) with logical connectives $\wedge$ and $\vee$. The disjunction $\vee$ is regarded as composing branches of the verification, i.e., each branch (conjunctive formula) is analyzed independently.

The projection $\downarrow$ extracts a predicate P, in which a bound variable x (in the scope of an expression E) is substituted in a formula. When regarding a conjunctive formula γ as an assignment from bound variables to predicates, $\gamma \downarrow_x$ coincides with the restriction to x. For instance, $(leq(x,y) \wedge \nabla(xs)) \downarrow_x = leq^y(x)$. Note that the $\downarrow$ operator is used when the local definition of x in a `let`-expression is analyzed, thus $P = \gamma \downarrow_x$ must exclude x. In our specification language, if such case occurs then $P(x)$ can be reduced **true** or **false** using the entailment relation.

The $\neg$-elimination operator $\ominus$ is defined by

$$\ominus[\ (Cond \wedge P) \vee (\neg Cond \wedge P')\ ] = \vee_i Q_i$$

where each Q_i satisfies $Cond \wedge P \sqsubseteq Q_i$ and $\neg Cond \wedge P' \sqsubseteq Q_i$. For instance, $\ominus[\ (leq(e,d) \wedge \forall leq^d(es)) \vee (\neg leq(e,d) \wedge \forall leq^e)\ ] = \forall leq^d(es) \vee \forall leq^e(es)$.

The interpretation of a function call

$$\Psi[\![f\ E]\!]P = \Psi[\![E]\!]((fvp[\![f]\!]P^{\beta\bar{\eta}})\theta^{\beta\bar{\eta}})$$

requires another operation, called $\beta\bar{\eta}$-expansion (similar to the substitution calculus in higher-order rewrite systems [24]). When a function f is called in an expression E, bound variables (except for an input variable to f) become out of the scope. Thus, if a predicate P contains a bound variable, then it must be replaced with a free variable and the substitution to the free variable must be kept. They are $P^{\beta\bar{\eta}}$ and $\theta^{\beta\bar{\eta}}$. For instance, $\Psi[\![f\ E]\!]\forall leq^b$ creates $P^{\beta\bar{\eta}} = \forall leq^Z$ and $\theta^{\beta\bar{\eta}} = [Z \leftarrow b]$.

Axioms on basic predicates

$$\overline{equal} \equiv equal \qquad \overline{leq} \equiv geq \qquad null(\mathtt{nil}) \equiv \mathbf{true} \qquad null(x :: xs) \equiv \mathbf{false}$$

$$equal(x,x) \equiv \mathbf{true} \qquad geq(x,x) \equiv \mathbf{true} \qquad leq(x,x) \equiv \mathbf{true}$$

$$equal \sqsubseteq equal^x \times \overline{equal}^x \qquad geq \sqsubseteq geq^x \times \overline{geq}^x \qquad leq \sqsubseteq leq^x \times \overline{leq}^x$$

$$\frac{P \sqsubseteq P^x \times \bar{P}^x}{\forall P^y \sqsubseteq \forall P^z \wedge P(z,y)} \qquad \frac{P \sqsubseteq P^x \times \bar{P}^x}{\forall_l \forall_r P \sqsubseteq (\forall_l \forall_r P)^{xs} \times (\overline{\forall_l \forall_r P})^{xs}} \ \textbf{Transitivity}$$

Ordinary logical rules on logical connectives

$$P \wedge P \equiv P \qquad P_1 \sqsubseteq P_1 \wedge P_2 \qquad \mathbf{true} \wedge P \equiv P \qquad \mathbf{false} \wedge P \equiv \mathbf{false}$$

$$P \vee P \equiv P \qquad P_1 \vee P_2 \sqsubseteq P_1 \qquad \mathbf{true} \vee P \equiv \mathbf{true} \qquad \mathbf{false} \vee P \equiv P$$

$$\frac{P_1 \sqsubseteq P_2 \qquad P_1' \sqsubseteq P_2'}{P_1 \wedge P_1' \sqsubseteq P_2 \wedge P_2'} \qquad \frac{P_1 \sqsubseteq P_2 \qquad P_1' \sqsubseteq P_2'}{P_1 \vee P_1' \sqsubseteq P_2 \vee P_2'}$$

Entailment relation of predicate constructors

$$\frac{P_1 \sqsubseteq P_2}{\bar{P}_1 \sqsubseteq \bar{P}_2} \qquad \frac{P_1 \sqsubseteq P_2}{\dagger P_1 \sqsubseteq \dagger P_2}\ \textbf{list} \qquad \dagger (P_1 \wedge P_2) \equiv \dagger P_1 \wedge \dagger P_2 \quad \text{with } \dagger \in \{\forall, \forall_l, \forall_r, \nabla\}$$

$$\frac{P_1 \sqsubseteq P_1' \qquad P_2 \sqsubseteq P_2'}{P_1 \times P_2 \sqsubseteq P_1' \times P_2'}\ \textbf{pair} \qquad (P_1 \times P_2) \wedge (P_1' \times P_2') \equiv (P_1 \wedge P_1') \times (P_2 \wedge P_2')$$

$$\overline{\bar{P}} \equiv P \qquad \overline{P_1 \wedge P_2} \equiv \bar{P}_1 \wedge \bar{P}_2 \qquad \overline{P_1 \vee P_2} \equiv \bar{P}_1 \vee \bar{P}_2 \qquad \overline{P_1 \times P_2} \equiv P_2 \times P_1$$

$$\overline{\forall_l P} \equiv \forall_r \bar{P} \qquad \overline{\forall_r P} \equiv \forall_l \bar{P} \qquad (\forall_l P)^E \equiv \forall (P^E) \qquad \forall_l \forall_r P \equiv \forall_r \forall_l P$$

$$\forall P(a :: as) \equiv P(a) \wedge \forall P(as) \qquad \forall P(nil) \equiv \mathbf{true}$$

$$\nabla P(a :: as) \equiv \forall \bar{P}^a(as) \wedge \nabla P(as) \qquad \nabla P(nil) \equiv \mathbf{true}$$

Fig. 4. Entailment relation

Theorem 1. *The verification algorithm always terminates.*

(Sketch of proof) Basic predicates, variables, and constants appearing in a program are finite. A free variable is introduced as a substitute for a bound variable only when function-calls; thus, only finitely many free variables are used during verifications. Since each predicate constructor enriches types, once types of functions are fixed only finitely many applications of predicate constructors are possible. The finiteness of an input-dependent abstract domain is then guaranteed. The algorithm is therefore formulated as the least fix-point computation on a finite abstract domain, so that it terminates. ■

$$
\begin{aligned}
\Phi[\![\{\ \texttt{fun}\ f_i\ x_i = E_i\ ;\ \}\ ]\!] &= fvp \text{ whererec } fvp = [\ (\lambda P_1 \cdots P_n.(\Psi[\![E_i]\!]P_i)\downarrow_{x_i})/f_i\ ] \\
\Psi[\![C]\!]P &= \begin{cases} \textbf{true} & \text{if } P(C) \\ \textbf{false} & \text{otherwise} \end{cases} \\
\Psi[\![x]\!]P &= P(x) \\
\Psi[\![op\ E]\!]P &= \Psi[\![E]\!](\Xi[\![op]\!]P) \\
\Psi[\![f\ E]\!]P &= \Psi[\![E]\!]((fvp[\![f]\!]P^{\beta\bar{\eta}})\theta^{\beta\bar{\eta}}) \\
\Psi[\![(E_1, E_2)]\!]P &= \begin{cases} \Psi[\![E_1]\!]P_1 \wedge \Psi[\![E_2]\!]P_2 & \text{if } P = P_1 \times P_2 \\ \Psi[\![E_1]\!]P^{E_2} \vee \Psi[\![E_2]\!]\bar{P}^{E_1} & \text{otherwise} \end{cases} \\
\Psi[\![E_1 :: E_2]\!]P &= \begin{cases} \Psi[\![E_1]\!]Q \wedge \Psi[\![E_2]\!]\forall Q & \text{if } P = \forall Q \\ \Psi[\![(E_1, E_2)]\!]\forall_r Q \wedge \Psi[\![E_2]\!]\nabla Q & \text{if } P = \nabla Q \end{cases} \\
\Psi[\![\texttt{if}\ Cond\ \texttt{then}\ E_1\ \texttt{else}\ E_2]\!]P &= \ominus[(Cond \wedge \Psi[\![E_1]\!]P) \vee (\neg Cond \wedge \Psi[\![E_2]\!]P)] \\
\Psi[\![\texttt{let val}\ x = E_1\ \texttt{in}\ E_2]\!]P &= \Psi[\![E_1]\!](\Psi[\![E_2]\!]P)\downarrow_x \\
\Psi[\![\texttt{let val}\ (x,y) = E_1\ \texttt{in}\ E_2]\!]P &= \Psi[\![E_1]\!](\Psi[\![E_2]\!]P)\downarrow_{(x,y)} \\
\Psi[\![\texttt{let val}\ x :: xs = E_1\ \texttt{in}\ E_2]\!]P &= \begin{cases} \Psi[\![E_1]\!]\nabla Q \text{ if } (\Psi[\![E_2]\!]P)\downarrow_{(x,xs)} = \forall_r Q \\ \Psi[\![E_1]\!]\forall Q \text{ if } (\Psi[\![E_2]\!]P)\downarrow_x = Q \text{ or } (\Psi[\![E_2]\!]P)\downarrow_{xs} = \forall Q \end{cases}
\end{aligned}
$$

$$
\text{where } \begin{cases} \Psi : (Fvp \rightarrow) Exp \rightarrow Pred \rightarrow Formula & \Xi : Prim \rightarrow Pred \rightarrow Pred \\ \Phi : Prog \rightarrow Fvp & \downarrow : Formula \rightarrow Bv \rightarrow Pred \\ fvp \in Fvp = Fv \rightarrow Pred \rightarrow Pred & \ominus : Formula \rightarrow Formula \end{cases}
$$

Fig. 5. Abstract semantics of verification

3.2 Soundness by Domain-Induced Abstract Interpretation

In this section, we will show how the abstract interpretation Φ is obtained as a domain-induced abstract interpretation, i.e., an abstract interpretation induced from domain abstractions. As a consequence, the soundness proof is given. Note that an automatic verification cannot be complete by nature.

Let the domain and codomain of a function f of type $\alpha \rightarrow \beta$ be D_α and D_β, respectively. Let the power domain $PD[D_\alpha]$ of D_α be $\{cl^{\downarrow}_{D_\alpha}(\mathcal{X}) \mid \mathcal{X} \subseteq D_\alpha\}$ with the order $\sqsubseteq_{-1} = \supseteq$, where $cl^{\downarrow}_{D_\alpha}$ is the downward closure operator in D_α.

Φ (in Fig. 5) is expressed as the two-step domain-induced abstract interpretation (as indicated in Fig. 6). The first step is backward and consists of

- the abstract domain $PD[D_\alpha]$
- the concretization map $conc^1_\alpha = id_{D_\alpha}$
- the abstraction map $abs^1_\alpha = cl^{\downarrow}_{D_\alpha}$

This step precisely detects how much of the input is enough to produce the output satisfying the specification. The next step approximates according to the specification language in order to make the analysis decidable. Let $Pred_\alpha$ be a

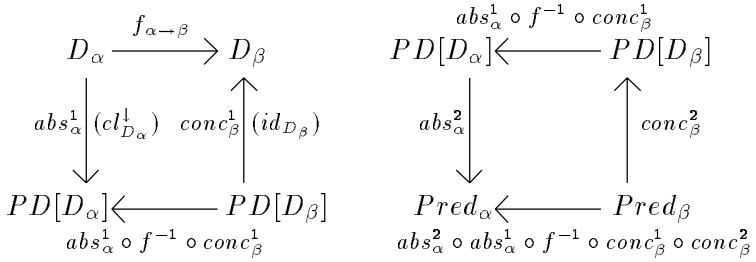

Fig. 6. Two steps of domain-induced abstract interpretation

set of predicates on D_α generated as P^-_G in Fig. 3. The second step is forward and consists of

- the abstract domain $Pred_\alpha$.
- the concretization map $conc^2_\alpha(P) = cl^{\downarrow}_{D_\alpha}(\{x \in D_\alpha \mid P(x)\})$ for $P \in Pred_\alpha$.
- the abstraction map $abs^2_\alpha(\mathcal{X}) = \sqcap(\{P \in Pred_\alpha \mid conc^2_\alpha(P) \subseteq \mathcal{X}\})$ for $\mathcal{X} \in PD[D_\alpha]$.

Note that the abstract domain $Pred_\alpha$ is a lattice wrt the entailment relation. For instance, $P \sqcup Q$ and $P \sqcap Q$ always exists as $P \wedge Q$ and $P \vee Q$, respectively.

Thus an abstract interpretation Ξ on a primitive function op of type $\alpha \to \beta$ is defined by $\Xi(op) = abs \cdot op^{-1} \cdot conc$, where $abs_\alpha = abs^2_\alpha \cdot abs^1_\alpha$ and $conc_\beta = conc^1_\beta \cdot conc^2_\beta$. Similar to Ψ on expressions. The abstract interpretation Φ on recursively defined functions f_i's is obtained by the least fix-point computation.

Definition 2. For an abstract interpretation Φ, a function f is *safe* if f satisfies $\Phi(f) \sqsubseteq abs \cdot f^{-1} \cdot conc$. An abstract interpretation Ψ is safe if each primitive function is safe.

Theorem 3. *The verification algorithm is sound (i.e., the detected property of a program always holds if a program terminates).*

(Sketch of proof) Since the concretization map $conc_\alpha$ and the abstraction map abs_β satisfy $abs_\alpha \cdot conc_\alpha = id_{D_\alpha}$ and $conc_\alpha \cdot abs_\alpha \subseteq cl^{\downarrow}_{D_\alpha}$, a recursively defined function is safe. Thus the detected property is sound. ■

4 Example: Verifying Orderedness of Sorting

The verification algorithm is explained here by an example of orderedness $\mathbf{true} \to \nabla geq(sort\ X)$. When unknown properties of user-defined functions are required, new conjectures are produced. For instance, when verifying $\mathbf{true} \to \nabla geq(sort\ X)$, it automatically produces and proves the lemmata; $\forall leq^Z(X) \to \forall leq^Z(sort\ X)$, $\neg null \wedge \forall leq^Z(Y) \to leq^Z \times \forall leq^Z(max\ Y)$, and $\neg null(Y) \to \forall_r geq(max\ Y)$. The generation of lemmata is shown at the top of Fig. 7. The vertical wavy arrow indicates an iterative procedure, the double arrow indicates the creation of a conjecture, and the arrow returns the resulting lemma.

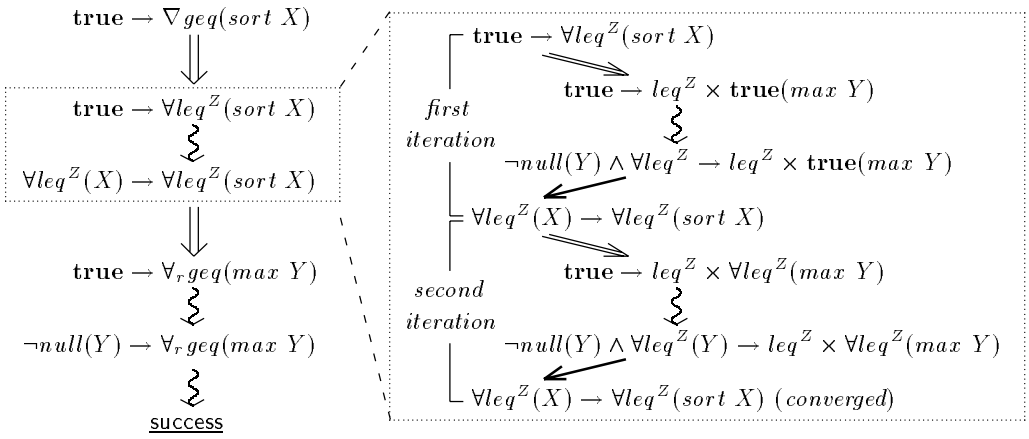

Fig. 7. Generation of lemmata for $\mathbf{true} \rightarrow \nabla geq(sort\ X)$

For instance, $\forall leq^Z(X) \rightarrow \forall leq^Z(sort\ X)$ means that if an input of *sort* is *less-than-or-equal-to* any given Z, an output is also *less-than-or-equal-to* Z. This lemma is generated as a conjecture $\mathbf{true} \rightarrow \forall leq^Z(sort\ X)$ at the `else`-branch of the `if`-expression in `sort` ($\Psi[\![$`b::sort bs`$]\!]\nabla geq$) as follows.

$$\begin{aligned} \nabla geq(b :: sort\ bs) &\equiv \forall_r geq(b, sort\ bs) \wedge \nabla geq(sort\ bs) \\ &\equiv \quad \forall leq^b(sort\ bs) \wedge \nabla geq(sort\ bs) \end{aligned}$$

Since there are no conjectures related to $\forall leq^b(sort\ X)$ in the recursion hypothesis, a new conjecture is created. But properties of functions (P_F in Fig. 3) exclude bound variables. Thus, by the $\beta\bar{\eta}$-expansion, $\forall leq^b(sort\ X)$ is transformed to $\forall leq^Z(sort\ X)$ with the substitution $[Z \leftarrow b]$, and $\mathbf{true} \rightarrow \forall leq^Z(sort\ X)$ is created. This means that no local information on b is used during the verification of $\mathbf{true} \rightarrow \forall leq^Z(sort\ X)$. This conjecture does not hold; instead, we obtain $\forall leq^Z(X) \rightarrow \forall leq^Z(sort\ X)$ as a lemma.

A typical example of the use of the entailment relation appears in the verification of $\neg null(Y) \rightarrow \forall_r geq(max\ Y)$. At the second `if`-expression in `max`, $\Psi[\![$`if leq(e,d) then (d,e::es) else (e,d::es)`$]\!]\forall_r geq$ is created. Thus, $(leq(e,d) \wedge \forall leq^d(es)) \vee (\neg leq(e,d) \wedge \forall leq^e(es))$ is obtained. From the transitivity of leq, $leq(e,d) \wedge \underline{\forall leq^d(es)} \sqsubseteq leq(e,d) \wedge \underline{\forall leq^e(es)}$ (see the underlined parts), therefore we obtain $\forall leq^e(es)$ by the $\neg$-elimination. Note that the $\neg$-elimination also creates $\forall leq^d(es)$, but only $\forall leq^e(es)$ branch is successful, i.e., from the recursion hypothesis $\Psi[\![$`max ds`$]\!]\forall_r geq$ is reduced to $\neg null(ds)$, as desired. Thus $\forall leq^d(es)$ is omitted.

5 Extensions

5.1 New Predicate Constructors

In this section we introduce new predicate constructors and extend the entailment relation to make it possible to verify weak preservation of sort programs.

The new predicate constructors are $\exists, \exists_l, \exists_r$, and $\triangle$. The predicates are extended by updating part of the grammar in Fig. 3 with

$$\begin{array}{ll} P_R = \forall P_U \mid \underline{\exists P_U} \mid \nabla P_G \mid \underline{\triangle P_G} \mid P_P^E \mid P_R \wedge P_R \mid P_R \vee P_R & \text{properties of lists} \\ P_P = P_B \mid \overline{P_P} \mid P_S \times P_S \mid \underline{\forall_r P_B} \mid \forall_l P_B \mid \underline{\exists_l P_B} \mid \underline{\exists_r P_B} \mid & \text{properties of pairs} \\ \quad P_P \wedge P_P \mid P_P \vee P_P & \end{array}$$

where the underlined parts are newly added. Their meanings are shown by examples in the table below. The entailment relation is enriched as in Fig. 8.

predicate	*true*	*false*
$\exists geq^5(as)$	[3,6,4]	([3,2,4]), nil
$\exists_l leq(as, a)$	([3,6,4], 5)	([3,6,4], 2), (nil, 5)
$\exists_r leq(a, as)$	(5, [3,6,4])	(7, [3,6,4]), (5, nil)
$\triangle leq(as)$	[3,2,4]	[3,6,4], [3], nil

Then weak preservation of `sort` is expressed by

$$\mathbf{true} \rightarrow \overline{\forall_l \exists_r equal \wedge \forall_r \exists_l equal}^X (sort\ X).$$

During the verification of $\mathbf{true} \rightarrow \overline{\forall_l \exists_r equal}^X (sort\ X)$, the key step is at $\forall_l \exists_r equal(as, b :: sort(bs))$ in `sort`. By transitivity $\forall_l \exists_r equal(as, b :: bs) \wedge \forall_l \exists_r equal(b :: bs, b :: sort(bs))$ is inferred. To solve the second component, the entailment relation $\forall_l \exists_r P(a :: as, b :: bs) \sqsubseteq P(a,b) \wedge \forall_l \exists_r P(as, bs)$ is used. This is obtained as a transitive closure by

$$\begin{aligned} \forall_l \exists_r P(a :: as, b :: bs) &\equiv \exists_r P(a, b :: bs) \wedge \underline{\forall_l \exists_r P(as, b :: bs)} \\ &\sqsubseteq (P(a,b) \underline{\vee \exists_r P(a, bs)}) \wedge \forall_l \exists_r P(as, bs) \\ &\sqsubseteq P(a,b) \wedge \forall_l \exists_r P(as, bs). \end{aligned}$$

Thus $\forall_l \exists_r equal(b :: bs, b :: sort(bs))$ is reduced to $\forall_l \exists_r equal(bs, sort(bs))$ which is a recursion hypothesis. The rest $\forall_l \exists_r equal(as, b :: bs)$ creates the conjecture

$$\mathbf{true} \rightarrow \overline{\forall_l (equal \times \mathbf{true} \vee \mathbf{true} \times \exists_r equal)}^Y (max\ Y)$$

at `(b,bs) = max as`, and similar approximations occur in `max` at expressions `(d,e::es)` and `(e,d::es)`. $\mathbf{true} \rightarrow \overline{\exists_l \forall_r equal}^X (sort\ X)$ is similarly verified.

5.2 Uninterpreted Functions and Predicates

This section extends the range of conditional expressions that can be included in the programs to be verified. Function symbols (either primitive or user-defined) in the conditional part of an `if`-expression are allowed. They are left uninterpreted during the verification, and the result will be refined by partial evaluation of these function symbols.

The example is a formatting program `format` that formats a sentence (expressed by a list of strings) as a list of sentences each of which has a width *less-than-or-equal-to* a specified number M. Its specifications are as follows.

$$\frac{P_1 \sqsubseteq P_2}{\triangle P_1 \sqsubseteq \triangle P_2} \qquad \frac{P_1 \sqsubseteq P_2}{\dagger P_1 \sqsubseteq \dagger P_2}\ \textbf{list} \qquad \dagger\,(P_1 \vee P_2) \equiv \dagger P_1 \vee \dagger P_2 \quad \text{with } \dagger \in \{\exists, \exists_l, \exists_r\}$$

$$\overline{\exists_l P} \equiv \exists_r \bar{P} \qquad \overline{\exists_r P} \equiv \exists_l \bar{P} \qquad (\exists_l P)^E \equiv \exists (P^E) \qquad \exists_l \exists_r P \equiv \exists_r \exists_l P$$

$$\frac{P \sqsubseteq P^x \times \bar{P}^x}{\forall_l \exists_r P \sqsubseteq (\forall_l \exists_r P)^{xs} \times (\overline{\forall_l \exists_r P})^{xs}} \qquad \frac{P \sqsubseteq P^x \times \bar{P}^x}{\exists_l \forall_r P \sqsubseteq (\exists_l \forall_r P)^{xs} \times (\overline{\exists_l \forall_r P})^{xs}}\ \textbf{Transitivity}$$

$$\exists P(a :: as) \equiv P(a) \vee \exists P(as) \qquad \forall_l \exists_r P(as, b :: bs) \sqsubseteq \forall_l \exists_r P(as, bs)$$

$$\exists P(nil) \equiv \textbf{false} \qquad \forall_r \exists_l P(a :: as, bs) \sqsubseteq \forall_r \exists_l P(as, bs)$$

$$\triangle P(a :: b :: bs) \equiv \exists \bar{P}^a (b :: bs) \wedge (null(bs) \vee \triangle P(b :: bs))$$

$$\triangle P(a :: nil) \equiv \textbf{false} \qquad \triangle P(nil) \equiv \textbf{false}$$

Fig. 8. New entailment relation

- Each sentence of the output must have a width less-than-equal to M.
- The order of each word in the input must be kept in the output.
- Each word of the input must appear in the output, and vice versa.

```
fun format as = f (as,nil);

fun f (bs,cs) = if null bs then [cs] else
                   let val d::ds = bs
                    in if leq (width (cs@[d]),M)
                          then f (ds,cs@[d]))
                          else cs::f (ds,[d]) end;

fun width es = if null es then 0 else
                  let val f::fs=es in
                   if null fs then size f
                              else 1+size f+width fs end;
```

In this example, *string* is added to base types. Basic functions + and constants $0, 1$ also are used in the program, but they are not directly related to the verification. Thus, their interpretation and entailment relations are omitted.

The first specification of `format` states that an output must satisfy $\forall(leq^M \cdot width)$. Note that this predicate allows a function symbol $width$ in it. Verification starts with $\textbf{true} \rightarrow \forall(leq^M \cdot width)(format\ X)$, which is immediately reduced to $\textbf{true} \rightarrow \forall(leq^M \cdot width)(f\ Y)$. The result of the verification is

$$(\forall leq^M \cdot width \cdot [\,]) \times (leq^M \cdot width)(Y) \rightarrow \forall(leq^M \cdot width)(f\ Y),$$

and this deduces

$$(\forall leq^M \cdot width \cdot [\,])(X) \wedge (leq^M \cdot width)(nil) \rightarrow \forall(leq^M \cdot width)(format\ X).$$

Note that the result is not affected by whatever $width$ is, since $width$ is left uninterpreted. The key steps are at `if leq (width (cs@[d]),M) then ...`. Since the throughout of `then`-branches $leq(width\ (cs@[d]), M)$ holds, the second argument of $f\ (ds, cs@[d])$ always satisfies $leq^M \cdot width$. These steps depend only on $\vee$-elimination so that the function symbol $width$ remains uninterpreted.

With the aid of partial evaluation which leads to $width\ nil = 0$, $width\ [x] = size\ x$, we obtain $\forall(leq^M \cdot size)(X) \wedge leq(0, M) \rightarrow \forall(leq^M \cdot width)(format\ X)$. For the partial evaluation, only the relation between $size$ and $width$ is important. The information on the function $size$ is required only when the final result above is interpreted by a human being. Note that in general a partial evaluation may not terminate. However, this final step is devoted to transforming the detected property into a more intuitive form for a human being, and even if it fails the detected property is correct.

The second and the third specification of `format` are similar to orderedness and weak preservation of `sort`, respectively. They require further extensions.

The second specification of `format` is expressed by a *fresh* binary predicate Rel on pairs of strings as $\nabla Rel\ (X) \rightarrow \forall\nabla Rel \wedge \nabla\Box Rel\ (format\ X)$ where $\Box$ is an abbreviation of $\forall_l\forall_r$. Note that throughout the verification the predicate Rel is left uninterpreted. This implies that the specification above holds for any binary relation Rel. Finally, the meaning of Rel is assumed by a human being, and in this case it is suitable to be interpreted as the appearance order of strings.

The third specification is expressed by $\mathbf{true} \rightarrow \overline{\forall_l\exists_r\exists_r equal}^X (format\ X)$ and $\mathbf{true} \rightarrow \overline{\forall_r\forall_r\exists_l equal}^X (format\ X)$. Our algorithm detects the latter, but for the former we also need a new transitivity-like entailment relation of type $list(\alpha) \times list(list(\alpha))$, i.e.,

$$\frac{P \sqsubseteq P^x \times \bar{P}^x}{\forall_l\exists_r\exists_r P \sqsubseteq (\forall_l\exists_r P)^{xs} \times (\overline{\forall_l\exists_r\exists_r P})^{xs}}.$$

6 Related Work

Many studies have been undertaken on verification. Most are based on theorem provers, for example, Coq, LCF, Boyer-Moore prover, Larch, and EQP. They require either complex heuristics or strong human guidance (or both), either of which is not easy to learn. However, for huge, complex, and critical systems, this price is worth paying.

The complementary approach uses intelligent compile-time error detection for easy debugging. For imperative programs, Bourdoncle proposed an assertion-based debugging called *Abstract debugging* [5, 4]. For logic programs, Comini, et. al. [8] and Bueno, et. al. [6] proposed extensions of declarative diagnosis based on abstract interpretation. Cortesi, et. al. [10, 18] proposed the automatic verification based on abstract interpretation. Levi and Volpe proposed the framework based on abstract interpretation to classify various verification methods [20]. Among them, target specifications primarily focus on behavior properties, such as termination, mutual exclusion of clauses, and size/cardinality relation between inputs and outputs.

In contrast, Métayer's [19] and our specification language (for functional programs) directly express the programmer's intention in a concise and declarative description. This point is more desirable for some situation, such as, when a novice programmer writes a relatively small program.

As an abstract interpretation, our framework is similar to *inverse image analysis* [14]. The significant difference is that inverse image analysis determines *how much of the input is needed to produce a certain amount of output* and computes Scott's *open* sets. Our framework, in contrast, determines *how much of the input is enough to produce a certain amount of output* and computes Scott's *closed* sets. In terms of [22], the former is expressed by a HOMT $(id, -, \sqsubseteq_{-0}, min)$, and the latter is expressed by $(id, -, \sqsubseteq_{-1}, max)$.

Similar techniques that treat abstract domain construction as a set of predicates are found in several places. However, predicates are either limited to unary [15, 17, 3, 21] (such as *null* and $\neg null$), or are limited to propositions corresponding to variables appearing in a (logic) program [9].

7 Conclusion

This paper reconstructs and extends the automatic verification technique of Le Métayer [19] based on a backward abstract interpretation. To show the effectiveness, two examples of the declarative specifications of the sorting and formatting programs are demonstrated. Although we adopted the simple and inefficient sorting program here, we also tried efficient sort programs, such as orderedness of *quick-sort* and *merge-sort* (both topdown and bottomup), and weak preservation of the *topdown merge-sort*. These verifications are quite messy by hand [23].

Future work will include the implementation and the exploration of its use on more complex examples. An efficient implementation may require an efficient reachability test algorithm (as well as a congruence-closure algorithm) and a strategy to prune highly-nondeterministic $\neg$-eliminations and transitivity.

Acknowledgments The author would like to thank CACA members for their useful comments and discussions.

References

1. S. Abramsky and C. Hankin, editors. *Abstract interpretation of declarative languages*. Ellis Horwood Limited, 1987.
2. T. Arts and J. Gisel. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 1999. *to appear*.
3. P.N. Benton. Strictness properties of lazy algebraic datatypes. In *Proc. 3rd WSA*, pages 206–217, 1993. Springer LNCS 724.
4. F. Bourdoncle. Abstract debugging of higher-order imperative programs. In *ACM SIGPLAN PLDI 1993*, pages 46–55, 1993.
5. F. Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In *4th ESEC*, pages 501–516, 1993. Springer LNCS 717.

6. F. Bueno et al. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *Proc. AADEBUG '97*, pages 155–169, 1997.
7. G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* The MIT Press, 1991.
8. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in LNCS, pages 22–50. Springer-Verlag, 1996.
9. A. Cortesi, G. Filé, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proc. 6th LICS*, pages 322–327, 1991.
10. A. Cortesi, B. Le Charlier, and S. Rossi. Specification-based automatic verification of Prolog programs. In *Proc. LOPSTR '96*, pages 38–57, 1996. Springer LNCS 1207.
11. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. 4th ACM POPL*, pages 238–252, 1977.
12. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier Science Publishers, 1990.
13. D. Detlefs and R. Forgaard. A procedure for automatically proving the termination of a set of rewrite rules. In *Proc. 1st RTA*, pages 255–270, 1985. Springer LNCS 202.
14. P. Dybjer. Inverse image analysis generalizes strictness analysis. *Information and Computation*, 90(2):194–216, 1991.
15. C. Ernoult and A. Mycroft. Uniform ideals and strictness analysis. In *Proc. 18th ICALP*, pages 47–59, 1991. Springer LNCS 510.
16. J. Gisel. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.
17. T.P. Jensen. Abstract interpretation over algebraic data types. In *Proc. Int. Conf. on Computer Languages*, pages 265–276. IEEE, 1994.
18. B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. Automatic verification of behavioral properties of Prolog programs. In *Proc. ASIAN '97*, pages 225–237, 1997. Springer LNCS 1345.
19. D. Le Métayer. Proving properties of programs defined over recursive data structures. In *Proc. ACM PEPM '95*, pages 88–99, 1995.
20. G. Levi and P. Volpe. Derivation of proof methods by abstract interpretation. In *ALP'98*, pages 102–117, 1998. Springer LNCS 1490.
21. M. Ogawa and S. Ono. Deriving inductive properties of recursive programs based on least-fixpoint computation. *Journal of Information Processing*, 32(7):914–923, 1991. *in Japanese.*
22. M. Ogawa and S. Ono. Transformation of strictness-related analyses based on abstract interpretation. *IEICE Trans.*, E 74(2):406–416, 1991.
23. L.C. Paulson. *ML for the working programmer.* Cambridge University Press, 2nd edition, 1996.
24. V. Van Oostrom. *Confluence for Abstract and Higher-Order Rewriting.* PhD thesis, Vrije universiteit, Amsterdam., 1994.

This article was processed using the LaTeX macro package with LLNCS style