Automatic Software Model Checking using CLP

Cormac Flanagan

Systems Research Center
Hewlett Packard Laboratories
flanagan@hpl.hp.com

Abstract. This paper proposes the use of constraint logic programming (CLP) to perform model checking of traditional, imperative programs. We present a semantics-preserving translation from an imperative language with heap-allocated mutable data structures and recursive procedures into CLP. The CLP formulation (1) provides a clean way to reason about the behavior and correctness of the original program, and (2) enables the use of existing CLP implementations to perform bounded software model checking, using a combination of symbolic reasoning and explicit path exploration.

1 Introduction

Ensuring the reliability of software systems is an important but challenging problem. Achieving reliability through testing alone is difficult, due to the test coverage problem. For finite state systems, model checking techniques that explore all paths have been extremely successful. However, verifying software systems is a much harder problem, because such systems are inherently infinite-state: many variables are (essentially) infinite-domain and the heap is of unbounded size.

A natural method for describing and reasoning about infinite-state systems is to use *constraints*. For example, the constraint $a[i] > y$ describes states in which the i^{th} component of a is greater than y. The close connection between constraints and program semantics is illustrated by Dijkstra's weakest precondition translation [10]. This translation expresses the behavior of a code fragment that does not use iteration or recursion as a boolean combination of constraints. Fully automatic theorem provers, such as Simplify [9], provide an efficient means for reasoning about the validity of such combinations of constraints. These techniques provide the foundation of the extended static checkers ESC/Modula-3 [8] and ESC/Java [14].

Unfortunately, iterative and recursive constructs, such as `while` loops, `for` loops, and recursive procedure calls, cannot be directly translated into boolean combinations of constraints. Instead, extended static checkers rely on the programmer to supply loop invariants and procedure specifications to aid in this

translation.[1] The need for invariants and specifications places a significant burden on programmer, and is perhaps the reason these checkers are not more widely used, even though they catch defects and improve software quality [14].

This paper presents a variant of the extended static checking approach that avoids the need for programmer-supplied invariants and specifications. Instead, we start with an unannotated program, which may include iterative and recursive constructs, and asserted correctness properties. We translate this program into in an extended logic called *constraint logic programming* (CLP) [19, 21, 20, 22]. Essentially, a constraint logic program consists of the sequence of rules, each of which defines a particular relation symbol as a boolean combination of constraints. Since constraints may refer to relation symbols, these rules can be self- and mutually-recursive. By expressing iterative and recursive constructs of the original imperative program as recursive CLP rules, we avoid the need for programmer-supplied invariants and specifications.

This paper presents a semantics-preserving translation into CLP from an imperative language that is infinite-state and that supports global and local variables, heap-allocated mutable data structures, and recursive procedure calls. We use this translation to illustrate the connection between imperative programs and CLP, between program executions and depth-first CLP derivations, between procedure behaviors and sets of ground atoms, and between erroneous program executions and satisfiable CLP queries.

Our translation enables the use of efficient CLP implementations, such as SICStus Prolog [27], to check correctness properties of software. This implementation performs a depth-first search for a satisfying assignment, using efficient constraints solvers to symbolically reason about boolean variables, linear arithmetic, and functional maps. This search strategy corresponds to *explicitly* exploring all program execution paths, but *symbolically* reasoning about data values. That is, instead of explicitly enumerating all possible values for an integer variable x, the CLP implementation symbolically reasons about the consistency of a collection of constraints or linear inequalities on x. This symbolic analysis provides greater coverage and more efficient checking.

The depth-first search strategy may diverge on software with infinitely long or infinitely many execution paths. To cope with such systems, we bound the depth of the CLP search, thus producing a bounded software model checker. Our translation also facilitates software model checking using other CLP implementation techniques, such as breadth-first search, tableaux methods, or subsumption, which may provide stronger termination and error detection properties.

The remainder of the paper proceeds as follows. The next section provides a review of CLP. Section 3 illustrates our CLP translation by applying it to an example program, and uses the CLP representation to detect defects in the program. Section 4 presents the imperative language that is the basis for our formal development, and section 5 translates this language into CLP. Section 6 uses the CLP representation for program checking and defect detection. Section 7 discusses related work, and we concluded in Section 8.

[1] Loops without invariants are handled in a manner that is unsound but still useful.

2 A Review of Constraint Logic Programming

In this section, we provide a brief review of the constraint logic programming paradigm [19, 21, 20, 22]. A *term* t is either a variable or the application of a primitive function f to a sequence of terms. An *atom* $r(\vec{t})$ is the application of a user-defined relation r to a term sequence $\vec{t}$. A *primitive constraint* $p(\vec{t})$ is the application of a primitive predicate p to a term sequence. *Constraints* include primitive constraints and their negations, conjunction, disjunction, and atoms. A *rule* $r(\vec{t}) \leftarrow c$ is an (implicitly universally quantified) implication, and provides a definition of the relational symbol r. For example, the rule $r(x, y) \leftarrow x = y$ defines r as the identity relation.

Primitive functions include binary functions for addition and subtraction, nullary constants, and the `select` and `store` operations, which are explained in Section 5. Primitive predicates include equality, disequality, inequalities, and the nullary predicates *true* and *false*. We sometimes write binary function and predicate applications using infix instead of prefix notation.

CLP Syntax

(*terms*)	$t ::= x \mid f(\vec{t})$	(*variables*)	x, y, z
(*atoms*)	$a ::= r(\vec{t})$	(*constants*)	$k \in \{0, 1, 2, \ldots\}$
(*constraints*)	$c ::= p(\vec{t}) \mid \neg p(\vec{t})$	(*primitive fns*)	$f \in \{k, +, -, \texttt{select}, \texttt{store}\}$
	$\mid \; c \wedge c \mid c \vee c \mid a$	(*primitive preds*)	$p \in \{\mathit{true}, \mathit{false}, =, \neq, <, \ldots\}$
(*rules*)	$R ::= a \leftarrow c$	(*relation names*)	r

A CLP program $\vec{R}$ is a sequence of rules. These rules may be self- or mutually-recursive, and so the CLP program $\vec{R}$ may yield multiple models. We are only interested in the least model of $\vec{R}$ that is compatible with the intended interpretation $\mathcal{D}$ of the primitive functions and predicates. In particular, we are interested in the question of whether this least compatible model of $\vec{R}$ implies a particular *goal* or atom a, which we write as $lm(\vec{R}, \mathcal{D}) \models \tilde{\exists}\, a$, where $\tilde{\exists}\, a$ existentially quantifies over all free variables in a.

Much work on the implementation and optimization of CLP programs has focused on answering such queries efficiently. In the following section, we leverage this effort to check correctness properties of an example program, without the need for procedure specifications or loop invariants.

3 Overview

To illustrate our method, consider the example program shown in Figure 1, column 1. This program is a variant of the locking example used by the BLAST checker [18]. The procedures `lock` and `unlock` acquire and release the lock L, respectively, where $L = 1$ if the lock is held, and is zero otherwise. The correctness property we wish to check is that (1) the procedure `lock` is never called when the lock is already held, and (2) the procedure `unlock` is never called unless the lock is already held. These correctness properties are expressed as assertions in the `lock` and `unlock` procedures. Hence, checking these properties reduces to

Program	Transfer relations	Error relations

`lock() {`
`  assert L = 0;`
`  L := 1;`
`}`
	$\mathbf{T}\text{lock}(L, N, D, L_1, N, D) \leftarrow$
	$\wedge\ L = 0$
	$\wedge\ L_1 = 1$
		$\mathbf{E}\text{lock}(L, N, D) \leftarrow$
		$L \neq 0$

`unlock() {`
`  assert L = 1;`
`  L := 0;`
`}`
	$\mathbf{T}\text{unlock}(L, N, D, L_1, N, D) \leftarrow$
	$\wedge\ L = 1$
	$\wedge\ L_1 = 0$
		$\mathbf{E}\text{unlock}(L, N, D) \leftarrow$
		$L \neq 1$

`main() {`
`  loop();`
`  unlock();`
`}`
	$\mathbf{T}\text{main}(L, N, D, L_2, N_2, D_2) \leftarrow$
	$\wedge\ \mathbf{T}\text{loop}(L, N, D, L_1, N_1, D_1)$
	$\wedge\ \mathbf{T}\text{unlock}(L_1, N_1, D_1, L_2, N_2, D_2)$
		$\mathbf{E}\text{main}(L, N, D) \leftarrow$
		$\vee\ \mathbf{E}\text{loop}(L, N, D)$
		$\vee\wedge\ \mathbf{T}\text{loop}(L, N, D, L_1, N_1, D_1)$
		$\wedge\ \mathbf{E}\text{unlock}(L_1, N_1, D_1)$

`loop() {`
`  lock();`
`  D := N;`
`  unl();`
`  if (N != D) {`
`  loop();`
`  } else {`
`  // skip`
`  }`
`}`
	$\mathbf{T}\text{loop}(L, N, D, L_4, N_4, D_4) \leftarrow$
	$\wedge\ \mathbf{T}\text{lock}(L, N, D, L_1, N_1, D_1)$
	$\wedge\ D_2 = N_1$
	$\wedge\ \mathbf{T}\text{unl}(L_1, N_1, D_2, L_3, N_3, D_3)$
	$\wedge\vee\wedge\ N_3 \neq D_3$
	$\wedge\ \mathbf{T}\text{loop}(L_3, N_3, D_3, L_4, N_4, D_4)$
	$\vee\wedge\ N_3 = D_3$
	$\wedge\ L_4 = L_3$
	$\wedge\ N_4 = N_3$
	$\wedge\ D_4 = D_3$
		$\mathbf{E}\text{loop}(L, N, D) \leftarrow$
		$\vee\ \mathbf{E}\text{lock}(L, N, D)$
		$\vee\wedge\ \mathbf{T}\text{lock}(L, N, D, L_1, N_1, D_1)$
		$\wedge\ D_2 = N_1$
		$\wedge\vee\ \mathbf{E}\text{unl}(L_1, N_1, D_2)$
		$\vee\wedge\ \mathbf{T}\text{unl}(L_1, N_1, D_2, L_3, N_3, D_3)$
		$\wedge\ N_3 \neq D_3$
		$\wedge\ \mathbf{E}\text{loop}(L_3, N_3, D_3)$

`unl() {`
`  if (*) {`
`  unlock();`
`  // N++;`
`  } else {`
`  // skip`
`  }`
`}`
	$\mathbf{T}\text{unl}(L, N, D, L_1, N_1, D_1) \leftarrow$
	$\vee\ \mathbf{T}\text{unlock}(L, N, D, L_1, N_1, D_1)$
	$\vee\wedge\ L_1 = L$
	$\wedge\ N_1 = N$
	$\wedge\ D_1 = D$
		$\mathbf{E}\text{unl}(L, N, D) \leftarrow$
		$\mathbf{E}\text{unlock}(L, N, D)$

Fig. 1. The example program and the corresponding error and transfer relations.

checking whether the example program *goes wrong* by violating either of these assertions.

The example contains three other routines, which manipulate two additional global variables, N and D. Thus, the state of the store is captured by the triple $\langle L, N, D\rangle$. The example uses the notation `if (*) ...` to express nondeterministic choice.

Our method translates each procedure m into two CLP relations:

1. the *error relation* $\mathbf{E}m(L, N, D)$, which describes states $\langle L, N, D\rangle$ from which the execution of m may go wrong by failing an assertion, and
2. the *transfer relation* $\mathbf{T}m(L, N, D, L', N', D')$, which, when m terminates normally, describes the relation between the pre-state $\langle L, N, D\rangle$ and post-state $\langle L', N', D'\rangle$ of m.

The transfer and error relations for the example program are shown in Figure 1, columns 2 and 3, respectively. The relation **E**lock says that `lock` goes wrong if L is not initially 0, and **T**lock says that `lock` terminates normally if L is initially 0, where $L = 1$ and N and D are unchanged the post-state. The relation **E**main says that `main` goes wrong if either `loop` goes wrong, or `loop` terminates normally and `unlock` goes wrong in the post-state of `loop`. The other relation definitions are similarly intuitive. Automatically generating these definitions from the program source code is straightforward.

We use these relation definitions to check if an invocation of `main` may go wrong by asking the CLP query $\mathbf{E}\text{main}(L, N, D)$. This query is satisfiable in the case where $L = 1$, indicating that the program may go wrong if the lock is held initially, and an inspection of the source code shows that this is indeed the case.

If we provide the additional precondition that the lock is not initially held, then the corresponding CLP query

$$L = 0 \wedge \mathbf{E}\text{main}(L, N, D)$$

is still satisfiable. An examination of the satisfying CLP derivation shows that it corresponds to the following execution trace: `main` calls `loop`, which calls `lock`, which returns to `loop`, which calls `unl`, which calls `unlock`, which returns to `unl`, which returns to `loop`, which returns to `main`, which calls `unlock`, which fails its assertion, since there are two calls to `unlock` without an intervening call to `lock`.

The reason for this bug is that the increment operation `N++` in `unl` (which is present in the original BLAST example) is commented out. After uncommenting this increment operation, the modified transfer relation for `unl` is:

$$\begin{aligned}
&\mathbf{T}\text{unl}(L, N, D, L_1, N_2, D_1) \leftarrow \\
&\qquad \vee \wedge \mathbf{T}\text{unlock}(L, N, D, L_1, N_1, D_1) \\
&\qquad\quad \wedge N_2 = N_1 + 1 \\
&\qquad \vee \wedge L_1 = L \\
&\qquad\quad \wedge N_2 = N \\
&\qquad\quad \wedge D_1 = D
\end{aligned}$$

The above CLP query is now unsatisfiable, indicating that the fixed example program does not go wrong and thus satisfies the desired correctness property.

4 The Source Language

This section presents the syntax and semantics of the imperative language that we use as the basis for our formal development.

4.1 Syntax

A program is a sequence of procedure definitions. Each procedure definition consists of a procedure name and a sequence of formal parameters, which are bound

in the procedure body, and can be α-renamed in the usual fashion. The procedure body is an expression. Expressions include variable reference and assignment, `let`-expressions, application of primitive functions and user-defined procedures, conditionals, and assertions. To illustrate the handling of heap-allocated data structures, the language includes mutable pairs, and provides operations to create pairs and to access and update each field i of a pair, for $i = 1, 2$. Although our language does not include iterative constructs such as `while` or `for` loops, they can easily being encoded as tail-recursive procedures. In addition to local variables bound by `let`-expressions and parameter lists, programs may also manipulate the global variables $\vec{g}$. For simplicity, the language is untyped, although we syntactically distinguish boolean expressions, which are formed by the application of a primitive predicate to a sequence of arguments.

Programming Language Syntax

(*programs*)	$P ::= \vec{D}$	(*procedure names*)	m
(*definitions*)	$D ::= m(\vec{x})\ \{e\}$	(*global variables*)	$\vec{g}$
(*expressions*)	$e ::= x \mid x := e \mid \texttt{let}\ x = e\ \texttt{in}\ e$	(*special variables*)	$\vec{h} = h.h_1.h_2$
	$\mid\ f(\vec{e}) \mid m(\vec{e}) \mid \texttt{if}\ p(\vec{e})\ e\ e$		
	$\mid\ \texttt{assert}\ p(\vec{e}) \mid \langle e, e\rangle \mid e.i \mid e.i := e$		

Throughout this paper, we assume the original program and the desired correctness property have already been combined into an *instrumented program*, which includes `assert` statements that check that the desired correctness property is respected by the program. We say an execution of the instrumented program *goes wrong* if it fails an assertion because the original program fails the desired correctness property. The focus of this paper is to statically determine if the instrumented program can go wrong.

Notation We use $\vec{X}$ to denote a sequence of entities, $\vec{X}.\vec{Y}$ denotes sequence concatenation, and ϵ is the empty sequence. We sometimes interpret sequences as sets, and vice-versa. If M is a (partial) map, then the map $M[X := Y]$ maps X to Y and is otherwise identical to M, and the map $M[-X]$ is undefined on X and is otherwise identical to M. The operations $M[\vec{X} := \vec{Y}]$ and $M[-\vec{X}]$ are defined analogously. We use $\vec{X} = \vec{Y}$ to abbreviate $X_1 = Y_1 \wedge \ \ldots\ \wedge X_n = Y_n$. We use $e_1\ ;\ e_2$ to abbreviate $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, where x is not free in e_2.

4.2 Semantics

We formalize the meaning of programs using a "big step" operation semantics. A store σ is a partial mapping from variables to values. The set of values includes constants and maps. To represent pairs, the store σ maps three special variables, h, h_1, and h_2, to maps. The map $\sigma(h)$ describes which locations have been allocated, and $\sigma(h_1)$ and $\sigma(h_2)$ describe the components of allocated pairs. For any heap location l, if $\sigma(h)(l) = 0$ then the location l is not allocated, otherwise the

$$\boxed{P \vdash e : \sigma \to \sigma', v \qquad P \vdash e : \sigma\ \texttt{wr}}$$

$$\frac{}{P \vdash x : \sigma \to \sigma, \sigma(x)} \qquad \frac{P \vdash e : \sigma\ \texttt{wr}}{P \vdash x := e : \sigma\ \texttt{wr}} \qquad \frac{P \vdash e : \sigma \to \sigma', v}{P \vdash x := e : \sigma \to \sigma'[x := v], v}$$

$$\frac{P \vdash e_1 : \sigma\ \texttt{wr}}{P \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma\ \texttt{wr}} \qquad \frac{\begin{array}{c}P \vdash e_1 : \sigma \to \sigma', v_1 \\ P \vdash e_2 : \sigma'[x := v_1]\ \texttt{wr}\end{array}}{P \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma\ \texttt{wr}}$$

$$\frac{\begin{array}{c}P \vdash e_1 : \sigma \to \sigma', v_1 \\ P \vdash e_2 : \sigma'[x := v_1] \to \sigma'', v_2\end{array}}{P \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma \to \sigma''[-x], v_2}$$

$$\frac{P \vdash \vec{e} : \sigma\ \texttt{wr}}{P \vdash f(\vec{e}) : \sigma\ \texttt{wr}} \qquad \frac{P \vdash \vec{e} : \sigma \to \sigma', \vec{v}}{P \vdash f(\vec{e}) : \sigma \to \sigma', \mathcal{M}_f(f, \vec{v})} \qquad \frac{P \vdash \vec{e} : \sigma\ \texttt{wr}}{P \vdash m(\vec{e}) : \sigma\ \texttt{wr}}$$

$$\frac{\begin{array}{c}P \vdash \vec{e} : \sigma \to \sigma', \vec{v} \\ m(\vec{x})\ \{e\} \in P \\ \vec{x} \cap dom(\sigma') = \emptyset \\ P \vdash e : \sigma'[\vec{x} := \vec{v}]\ \texttt{wr}\end{array}}{P \vdash m(\vec{e}) : \sigma\ \texttt{wr}} \qquad \frac{\begin{array}{c}P \vdash \vec{e} : \sigma \to \sigma', \vec{v} \\ m(\vec{x})\ \{e\} \in P \\ \vec{x} \cap dom(\sigma') = \emptyset \\ P \vdash e : \sigma'[\vec{x} := \vec{v}] \to \sigma'', v\end{array}}{P \vdash m(\vec{e}) : \sigma \to \sigma''[-\vec{x}], v} \qquad \frac{\begin{array}{c}P \vdash \vec{e} : \sigma \to \sigma', \vec{v} \\ \text{if } \mathcal{M}_p(p, \vec{v}) \text{ then } i = 1 \text{ else } i = 2 \\ P \vdash e_i : \sigma' \to \sigma'', v\end{array}}{P \vdash \texttt{if}\ p(\vec{e})\ e_1\ e_2 : \sigma \to \sigma'', v}$$

$$\frac{P \vdash \vec{e} : \sigma\ \texttt{wr}}{P \vdash \texttt{if}\ p(\vec{e})\ e_1\ e_2 : \sigma\ \texttt{wr}} \qquad \frac{\begin{array}{c}P \vdash \vec{e} : \sigma \to \sigma', \vec{v} \\ \text{if } \mathcal{M}_p(p, \vec{v}) \text{ then } i = 1 \text{ else } i = 2 \\ P \vdash e_i : \sigma'\ \texttt{wr}\end{array}}{P \vdash \texttt{if}\ p(\vec{e})\ e_1\ e_2 : \sigma\ \texttt{wr}}$$

$$\frac{P \vdash \vec{e} : \sigma\ \texttt{wr}}{P \vdash \texttt{assert}\ p(\vec{e}) : \sigma\ \texttt{wr}} \qquad \frac{\begin{array}{c}P \vdash \vec{e} : \sigma \to \sigma', \vec{v} \\ \mathcal{M}_p(p, \vec{v}) = \mathit{false}\end{array}}{P \vdash \texttt{assert}\ p(\vec{e}) : \sigma\ \texttt{wr}} \qquad \frac{\begin{array}{c}P \vdash \vec{e} : \sigma \to \sigma', \vec{v} \\ \mathcal{M}_p(p, \vec{v}) = \mathit{true}\end{array}}{P \vdash \texttt{assert}\ p(\vec{e}) : \sigma \to \sigma, 0}$$

$$\frac{P \vdash e_1.e_2 : \sigma\ \texttt{wr}}{P \vdash \langle e_1, e_2 \rangle : \sigma\ \texttt{wr}} \qquad \frac{\begin{array}{c}P \vdash e_1.e_2 : \sigma \to \sigma', v_1.v_2 \qquad \sigma'(h)(l) = 0 \\ \sigma'' = \sigma'[h := \sigma'(h)[l := 1], h_i := \sigma'(h_i)[l := v_i]^{i \in 1,2}]\end{array}}{P \vdash \langle e_1, e_2 \rangle : \sigma \to \sigma'', l}$$

$$\frac{P \vdash e : \sigma\ \texttt{wr}}{P \vdash e.i : \sigma\ \texttt{wr}} \qquad \frac{P \vdash e : \sigma \to \sigma', l}{P \vdash e.i : \sigma \to \sigma', \sigma(h_i)(l)}$$

$$\frac{P \vdash e_1.e_2 : \sigma\ \texttt{wr}}{P \vdash e_1.i := e_2 : \sigma\ \texttt{wr}} \qquad \frac{\begin{array}{c}P \vdash e_1.e_2 : \sigma \to \sigma', v_1.v_2 \\ \sigma'' = \sigma'[h_i := \sigma'(h_i)[v_1 := v_2]]\end{array}}{P \vdash e_1.i := e_2 : \sigma \to \sigma'', v_2}$$

$$\boxed{P \vdash \vec{e} : \sigma \to \sigma', \vec{v} \qquad P \vdash \vec{e} : \sigma\ \texttt{wr}}$$

$$\frac{}{P \vdash \epsilon : \sigma \to \sigma, \epsilon} \qquad \frac{P \vdash e : \sigma\ \texttt{wr}}{P \vdash e.\vec{e} : \sigma\ \texttt{wr}} \qquad \frac{\begin{array}{c}P \vdash e : \sigma \to \sigma', v \\ P \vdash \vec{e} : \sigma'\ \texttt{wr}\end{array}}{P \vdash e.\vec{e} : \sigma\ \texttt{wr}} \qquad \frac{\begin{array}{c}P \vdash e : \sigma \to \sigma', v \\ P \vdash \vec{e} : \sigma' \to \sigma'', \vec{v}\end{array}}{P \vdash e.\vec{e} : \sigma \to \sigma'', v.\vec{v}}$$

Fig. 2. Evaluation rules.

components of the pair at location l are given by $\sigma(h_1)(l)$ and $\sigma(h_2)(l)$, respectively. This representation of pairs significantly simplifies the correspondence proof between imperative programs and constraint logic programs.

The judgment $P \vdash e : \sigma \to \sigma', v$ states that, when started from an initial store σ, the evaluation of expression e may terminate normally yielding a result value v and resulting store σ'. The judgment $P \vdash e : \sigma$ `wr` states that, when started from an initial store σ, the evaluation of expression e may go wrong by failing an assertion. Similarly, the judgments $P \vdash \vec{e} : \sigma \to \sigma', \vec{v}$ and $P \vdash \vec{e} : \sigma$ `wr` describes whether an expression sequence $\vec{e}$ terminates normally, yielding value sequence $\vec{v}$, or goes wrong, respectively. The rules defining these judgments are shown in Figure 2. These rules rely on the function $\mathcal{M}_f : \mathit{FnSym} \times \mathit{Value}^* \to \mathit{Value}$ and the relation $\mathcal{M}_p \subseteq \mathit{PredSym} \times \mathit{Value}^*$ to provide the meaning of primitive functions and predicates, respectively.

5 Translating Imperative Programs into CLP

We now describe the translation of imperative programs into CLP. At each step in the translation, the environment Γ maps each program variable x into a CLP term that provides a symbolic representation of the value of x. Given the initial environment Γ for an expression e, the judgment

$$\Gamma \vdash e \to w \,|\, n \cdot \Gamma' \cdot t$$

describes the behavior of e. The *wrong condition* w is a constraint describing initial states from which e may go wrong by failing an assertion. For example, the wrong condition of `assert` $x = 0$ is $\Gamma(x) \neq 0$, *i.e.*, the assertion goes wrong if x is not initially 0. Similarly, the *normal condition* n describes the initial states from which e may terminate normally. In this case, the environment Γ' symbolically describes values of variables in the post-state, and the term t is a symbolic representation of the result of e. The judgment $\Gamma \vdash \vec{e} \to w \,|\, n \cdot \Gamma' \cdot \vec{t}$ behaves in a similar manner on expression sequences, which may go wrong or may terminate normally producing a value sequence represented by $\vec{t}$.

The rules defining these judgements are shown in Figure 3. The rule [EXP VAR] states that the variable access x never goes wrong and always terminate normally without changing the program state. The rule retrieves a symbolic representation $\Gamma(x)$ for the value of x from the environment. The rule [EXP ASSIGN] for an assignment $x := e$ determines a symbolic representation t for e, and updates the environment to record that t represents of the current value of x. The rule [EXP LET] states that `let` $x = e_1$ `in` e_2 goes wrong if either e_1 goes wrong or if e_1 terminates normally and e_2 goes wrong.

Some translation rules are more complicated. For example, the rule [EXP IF] for the conditional `if` $p(\vec{e})$ e_1 e_2 needs to merge the environments Γ'_i produced by the translation of e_i, for $i = 1, 2$. To accomplish this merge, the rule determines the set $\vec{y}$ of variables assigned in either e_1 or e_2, and introduces an environment Γ'' that maps $\vec{y}$ to fresh variables. Then, having determined that the branch e_i of the conditional is executed, the rule asserts that the $\Gamma''(\vec{y}) = \Gamma'_i(\vec{y})$, thus

$\boxed{\Gamma \vdash e \to w \,|\, n \cdot \Gamma' \cdot t \qquad \Gamma \vdash \vec{e} \to w \,|\, n \cdot \Gamma' \cdot \vec{t}}$

[EXP VAR]
$$\frac{}{\Gamma \vdash x \to \mathit{false} \,|\, \mathit{true} \cdot \Gamma \cdot \Gamma(x)}$$

[EXP ASSIGN]
$$\frac{\Gamma \vdash e \to w \,|\, n \cdot \Gamma' \cdot t}{\Gamma \vdash x := e \to w \,|\, n \cdot \Gamma'[x := t] \cdot t}$$

[EXP LET]
$$\frac{\begin{array}{c}\Gamma \vdash e_1 \to w_1 \,|\, n_1 \cdot \Gamma_1 \cdot t_1 \\ \Gamma_1[x := t_1] \vdash e_2 \to w_2 \,|\, n_2 \cdot \Gamma_2 \cdot t_2\end{array}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \to w_1 \vee (n_1 \wedge w_2) \,|\, n_1 \wedge n_2 \cdot \Gamma_2[-x] \cdot t_2}$$

[EXP FN]
$$\frac{\Gamma \vdash \vec{e} \to w \,|\, n \cdot \Gamma' \cdot \vec{t}}{\Gamma \vdash f(\vec{e}) \to w \,|\, n \cdot \Gamma' \cdot f(\vec{t})}$$

[EXP CALL]
$$\frac{\begin{array}{c}\Gamma \vdash \vec{e} \to w \,|\, n \cdot \Gamma' \cdot \vec{t} \\ z, \vec{g'}, \vec{h'} \text{ fresh} \\ w' \equiv w \vee (n \wedge E_m(\vec{t}, \Gamma'(\vec{g}), \Gamma'(\vec{h}))) \\ n' \equiv n \wedge T_m(\vec{t}, \Gamma'(\vec{g}), \Gamma'(\vec{h}), \vec{g'}, \vec{h'}, z) \\ \Gamma'' \equiv \Gamma'[\vec{g} := \vec{g'}, \vec{h} := \vec{h'}]\end{array}}{\Gamma \vdash m(\vec{e}) \to w' \,|\, n' \cdot \Gamma'' \cdot z}$$

[EXP IF]
$$\frac{\begin{array}{c}\Gamma \vdash \vec{e} \to w \,|\, n \cdot \Gamma' \cdot \vec{t} \qquad \Gamma' \vdash e_i \to w_i \,|\, n_i \cdot \Gamma_i' \cdot t_i \\ z \text{ fresh} \qquad \vec{y} = \{y \mid \Gamma_1'(y) \neq \Gamma_2'(y)\} \\ \Gamma''(x) = \begin{cases} \Gamma_1'(x) & \text{if } x \notin \vec{y} \\ \text{fresh var} & \text{if } x \in \vec{y} \end{cases} \\ w' \equiv w \vee (n \wedge p(\vec{t}) \wedge w_1) \vee (n \wedge \neg p(\vec{t}) \wedge w_2) \\ n_1' \equiv n \wedge \ p(\vec{t}) \wedge n_1 \wedge z = t_1 \wedge \Gamma''(\vec{y}) = \Gamma_1'(\vec{y}) \\ n_2' \equiv n \wedge \neg p(\vec{t}) \wedge n_2 \wedge z = t_2 \wedge \Gamma''(\vec{y}) = \Gamma_2'(\vec{y})\end{array}}{\Gamma \vdash \texttt{if}\ p(\vec{e})\ e_1\ e_2 \to w' \,|\, (n_1' \vee n_2') \cdot \Gamma'' \cdot z}$$

[EXP ASSERT]
$$\frac{\Gamma \vdash \vec{e} \to w \,|\, n \cdot \Gamma' \cdot \vec{t}}{\Gamma \vdash \texttt{assert}\ p(\vec{e}) \to w \vee (n \wedge \neg p(\vec{t})) \,|\, n \wedge p(\vec{t}) \cdot \Gamma' \cdot 0}$$

[EXP PAIR]
$$\frac{\begin{array}{c}\Gamma \vdash e_1.e_2 \to w \,|\, n \cdot \Gamma' \cdot t_1.t_2 \\ \Gamma'' \equiv \Gamma'[h_i := \texttt{store}(\Gamma'(h_i), l, t_i)^{i \in 1,2}, h := \texttt{store}(\Gamma'(h), l, 1)] \\ l \text{ fresh} \qquad n' \equiv n \wedge \texttt{select}(\Gamma(h), l) = 0)\end{array}}{\Gamma \vdash \langle e_1, e_2 \rangle \to w \,|\, n' \cdot \Gamma'' \cdot l}$$

[EXP FIELD REF]
$$\frac{\Gamma \vdash e \to w \,|\, n \cdot \Gamma' \cdot t}{\Gamma \vdash e.i \to w \,|\, n \cdot \Gamma' \cdot \texttt{select}(\Gamma'(h_i), t)}$$

[EXP FIELD ASSIGN]
$$\frac{\begin{array}{c}\Gamma \vdash e_1.e_2 \to w \,|\, n \cdot \Gamma' \cdot t_1.t_2 \\ \Gamma'' \equiv \Gamma'[h_i := \texttt{store}(\Gamma'(h_i), t_1, t_2)]\end{array}}{\Gamma \vdash e_1.i := e_2 \to w \,|\, n \cdot \Gamma'' \cdot t_2}$$

[EXPS NONE]
$$\frac{}{\Gamma_1 \vdash \epsilon \to \mathit{false} \,|\, \mathit{true} \cdot \Gamma \cdot \epsilon}$$

[EXPS SOME]
$$\frac{\Gamma \vdash e \to w \,|\, n \cdot \Gamma' \cdot t \qquad \Gamma' \vdash \vec{e} \to w' \,|\, n' \cdot \Gamma'' \cdot \vec{t}}{\Gamma \vdash e.\vec{e} \to w \vee (n \wedge w') \,|\, n \wedge n' \cdot \Gamma'' \cdot t.\vec{t}}$$

$\boxed{\vdash D \to \vec{R} \qquad \vdash P \to \vec{R}}$

[DEF]
$$\frac{\begin{array}{c}\Gamma \equiv [\vec{x} := \vec{x}, \vec{g} := \vec{g}, \vec{h} := \vec{h}] \\ \Gamma \vdash e \to w \,|\, n \cdot \Gamma' \cdot t \\ \vec{R} = \left\{ \begin{array}{l} E_m(\vec{x}, \vec{g}, \vec{h}) \leftarrow w \\ T_m(\vec{x}, \vec{g}, \vec{h}, \Gamma'(\vec{g}), \Gamma'(\vec{h}), t) \leftarrow n \end{array} \right\}\end{array}}{\vdash m(\vec{x})\ \{e\} \to \vec{R}}$$

[DEFS]
$$\frac{P = D_1. \cdots . D_n \qquad \vdash D_i \to \vec{R_i}}{\vdash P \to \vec{R_1}. \cdots . \vec{R_n}}$$

Fig. 3. Translation rules.

recording that the representation of $\vec{y}$ in the resulting environment Γ'' come from the branch e_i. This translation of conditionals avoids the exponential blow-up of traditional VC generation algorithms [10], and is analogous to the compact VC generation algorithm of ESC/Java [16].

Our translation for pairs relies on the primitive functions `select` and `store`, where $\texttt{store}(a, i, v)$ extends a functional map a at index i with value v, and $\texttt{select}(a, i)$ selects the element at index i from map a. These two functions satisfy the select-of-store axioms:

$$\texttt{select}(\texttt{store}(a, i, v), i) = v$$
$$i \neq j \;\Rightarrow\; \texttt{select}(\texttt{store}(a, i, v), j) = \mathit{select}(a, j)$$

To aid in the translation, the environment Γ maps the special variables h, h_1, h_2 into CLP terms that symbolically model of the current state of the heap. The rule [EXP PAIR] for the pair creation expression $\langle e_1, e_2 \rangle$ introduces a fresh variable l and asserts that $\texttt{select}(\Gamma(h), l) = 0$, which means that the location l is not yet allocated. The rule then updates the environment (1) to map h to $\texttt{store}(\Gamma(h), l, 1)$, indicating that location l is now allocated, and (2) to map each h_i to $\texttt{store}(\Gamma(h_i), l, t_i)$, where the term t_i represents the value of e_i, for $i = 1, 2$. Thus, the rule records the contents of the pair in the new terms for h_1 and h_2. The rules for accessing and updating pairs operate in a similar manner.

The most novel aspect of our translation concerns its handling of procedure calls. Earlier approaches translated procedure calls using user-supplied specifications. However, since writing specifications for all procedures imposes a significant burden on the programmer, we use a different approach that leverages the ability to define relation symbols recursively in CLP.

We translate each procedure definition $m(\vec{x})\ \{e\}$ into two rules. The first rule defines an *error relation* E_m that describes pre-states from which an invocation of m may go wrong; the second rule defines a *transfer relation* T_m that, in situations where m terminates normally, describes the pre-state/post-state relation of m. The arguments to the error relation E_m are the formal parameters $\vec{x}$, the global variables $\vec{g}$, plus the three special variables $\vec{h} = h.h_1.h_2$ that model the heap. The arguments to T_m are again the formal parameters $\vec{x}$, the globals $\vec{g}$, the special variables $\vec{h}$, followed by $\vec{g'}$, which represents the post-state of the global variables, followed by $\vec{h} = h'.{h_1}'.{h_2}'$, which represents the post heap state, followed by a term representing the return value of m. The rule [EXP CALL] for a procedure call $m(\vec{e})$ generates a wrong condition w' that uses E_m to express states from which the execution of $m(\vec{e})$ may go wrong, and generates a normal condition n' that uses T_m to describe how $m(\vec{e})$ may terminate normally.

5.1 Correctness of the Translation

Given an imperative program P, we translate it into error and transfer relations $\vec{R}$ according to the translation rule $\vdash P \rightarrow \vec{R}$. For any expression e, the judgement

$$\Gamma \vdash e \rightarrow w \,|\, n \cdot \Gamma' \cdot t$$

describes the behavior of that expression from any initial state σ that is compatible with Γ, *i.e.*, where $dom(\Gamma) \subseteq dom(\sigma)$ and $lm(\vec{R}, \mathcal{D}) \models \tilde{\exists}\,(\sigma = \Gamma)$. We use the notation $\sigma = \Gamma$ to abbreviate $\bigwedge_{x \in dom(\sigma)} \sigma(x) = \Gamma(x)$, where $\sigma(x)$ means the ground term representing the value $\sigma(x)$.

To determine if e goes wrong from σ (*i.e.*, $P \vdash e : \sigma$ `wr`), we check

$$lm(\vec{R}, \mathcal{D}) \models \tilde{\exists}\,(\sigma = \Gamma \wedge w) \ .$$

Similarly, to check if e terminates normally, yielding post-store σ' and result v, we check

$$lm(\vec{R}, \mathcal{D}) \models \tilde{\exists}\,(\sigma = \Gamma \wedge n \wedge \sigma' = \Gamma' \wedge v = t) \ .$$

Thus, to check if the program's initial procedure *main* goes wrong, we use the CLP query:

$$lm(\vec{R}, \mathcal{D}) \models \tilde{\exists}\, E_{main}(\vec{g}, \vec{h}) \ .$$

If this query is satisfiable, the CLP implementation returns a satisfying assignment for $\vec{g}$ and $\vec{h}$, describing the initial state of an erroneous execution. If the CLP implementation also returns a CLP derivation, then this derivation corresponds in a fairly direct manner to a trace of the erroneous execution.

6 Applications

We next consider the example program shown in Figure 4, which, for clarity, is presented using Java syntax. This class implements rational numbers, where a rational is represented as a pair of integers for the numerator and denominator. The class contains a constructor for creating rationals and a method `trunc` for converting a rational to an integer. The example also contains a test harness, which reads in two integers, `n` and `d`, ensures that `d` is not zero, creates a corresponding rational, and then repeatedly prints out the truncation of the rational.

We wish to check that a division-by-zero error never occurs. We express this correctness property as an assertion in the `trunc` method, and translate the instrumented program into CLP rules. The CLP query **E**`main`() is satisfiable, indicating an error in the program. An investigation of a satisfying CLP derivation reveals the source of the error: the arguments are passed to the `Rational` constructor in the wrong order. Note that since both arguments are integers, Java's type system does not catch this error.

After fixing this bug, the query **E**`main`() is now unsatisfiable, indicating that a division-by-zero error cannot occur. However, the CLP implementation that we use, SICStus Prolog [27], requires several seconds to answer this query, since its depth-first search strategy explicitly iterates through the loop in `main` 10,000 times.

To avoid this inefficiency, we are currently developing a CLP implementation optimized towards software model checking. This implementation uses lazy predicate abstraction and counter-example driven abstraction refinement. Our

```
class Rational {

    int num, den;

    Rational(int n, int d) {
        num = n;
        den = d;
    }

    int trunc() {
        assert den != 0;
        return num/den;
    }
}
```

```
public static void main(String[] a) {
    int n = readInt(), d = readInt();
    if( d == 0 ) {
        return;
    }
    Rational r = new Rational(d,n);
    for(int i=0; i<10000; i++) {
        print( r.trunc() );
    }
}
```

Fig. 4. The example program `Rational`.

prototype implementation determines the unsatisfiability of the `Rational` example in just two iterations. We are currently extending this implementation to handle more realistic benchmarks.

7 Related Work

This paper can be viewed as a synthesis of ideas from extended static checking [8, 14] and model checking [5, 24, 3, 23]. An extended static checker translates the given program into a combination of constraints over program variables, and uses sophisticated decision procedures to reason about the validity of these constraints, thus performing a precise, goal-directed analysis. However, the translation of (recursive) procedure calls requires programmer-supplied specifications. We build on top of the ESC approach, but avoid the need for procedure specifications by targeting the extended logic of CLP, in which we can express recursion directly.

The depth-first search of standard CLP implementations [27] corresponds to explicit path exploration, much like that performed by software model checkers, such as Bandera [11]. However, whereas Bandera relies on programmer-supplied abstractions to abstract (infinite-state) data variables, the CLP implementation reasons about data values using collections of constraints, thus providing a form of automatic data abstraction. The programmer-supplied abstractions of Bandera do provide stronger termination guarantees, but may yield false alarms.

The software checkers SLAM [1] and BLAST [18] use a combination of predicate abstraction [17] and automatic predicate inference to avoid false alarms and the need for programmer-supplied abstractions, though they may not terminate. These tools have been successfully

applied to a number of device drivers. Both tools abstract the given imperative program to a finite-state boolean program, which is then model checked.

This paper suggests that the well-studied logic of CLP may also provide a suitable foundation for the development of such tools.

Delzanno and Podelski [7] also explore the use of CLP for model checking. They focus on concurrent systems expressed in the guarded-command specification language proposed by Shankar [26], which does not provide explicit support for dynamic allocation or recursion. The performance of their CLP-based model checking approach is promising.

Bruening [2] has built a dynamic assertion checker based on state-space exploration for multithreaded Java programs. Stoller [28] provides a generalization of Bruening's method to allow model checking of programs with either message-passing or shared-memory communication. Both of these approaches operate on the concrete program without any abstraction. Yahav [30] describes a method to model check multithreaded Java programs using a 3-valued logic [25] to abstract the store.

Abstract interpretation [6] is a standard framework for developing and describing program analyses. It provides the semantics basis for the abstractions in the above model checking tools and it has been applied successfully in many applications, including rocket controllers [29].

Instead of avoiding the need for loop invariants and specifications, another approach is to infer such annotations automatically. The Houdini annotation inference system [15, 13] re-uses ESC/Java as a subroutine in a generate-and-test approach to annotation inference. Daikon uses an empirical approach to find probable invariants [12].

Symbolic execution is the underlying technique of the successful bug-finding tool PREfix for C and C++ programs [4]. For each procedure in the given program, PREfix synthesizes a set of execution paths, called a *model*. Models are used to reason about calls, which makes the process somewhat modular, except that fixpoints of models are approximated iteratively for recursive and mutually recursive calls.

8 Conclusion

This paper explores the connection between two programming paradigms: the traditional imperative paradigm and the constraint logic programming paradigm. We have expressed the correctness of imperative programs in terms of CLP satisfiability, based on a novel, semantics-preserving translation from imperative programs to CLP programs. The resulting CLP programs provide a clean way to reason about the behavior and correctness of the original imperative program.

This connection has immediate practical applications: it enables us to use existing CLP implementations to check correctness properties of imperative programs. For depth-first CLP implementations, this approach yields an efficient method for bounded model checking of software, using a combination of symbolic reasoning for data values and explicit path exploration.

Finally, the logic of CLP is well-studied [19, 21, 20, 22], and may provide optimizations and implementation techniques such as tableaux methods and sub-

sumption [22], which offer the promise of complete model checking on certain classes of infinite-state programs. More experience on practical examples is certainly necessary, and may provide intuition and motivation to develop specialized CLP implementations optimized for software model checking.

References

1. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M. B. Dwyer, editor, *Model Checking Software, 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, May 2001.
2. D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, Massachusetts Institute of Technology, 1999.
3. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
4. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30(7):775–802, June 2000.
5. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*, Lecture Notes in Computer Science, pages 52–71. Springer-Verlag, 1981.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analyses of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on the Principles of Programming Languages*, pages 238–252, 1977.
7. G. Delzanno and A. Podelski. Model checking in CLP. *Lecture Notes in Computer Science*, 1579:223–239, 1999.
8. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Dec. 1998.
9. D. L. Detlefs, G. Nelson, and J. B. Saxe. A theorem prover for program analysis. Manuscript in preparation, 2002.
10. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
11. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, 2001.
12. M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland*, June 2000.
13. C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Inf. Process. Lett.*, 77(2–4):97–108, Feb. 2001.
14. C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

15. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, Mar. 2001.
16. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, Jan. 2001.
17. S. Graf and H. Saïdi. Construction of abstract state graphs via PVS. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.
18. T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, January 2001.
19. J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 111–119, Jan. 1987.
20. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
21. J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
22. M. J. Maher. A logic programming view of CLP. In *International Conference on Logic Programming*, pages 737–753, 1993.
23. K. L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. Kluwer Academic Publishers, 1993.
24. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Fifth International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.
25. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th Symposium on Principles of Programming Languages*, pages 105–118, 1999.
26. A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *Computing Surveys*, 25(3):225–302, 1993.
27. SICStus Prolog. On the web at `http://www.sics.se/sicstus/`.
28. S. Stoller. Model-checking multi-threaded distributed Java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking and Software Verification*, Lecture Notes in Computer Science 1885, pages 224–244. Springer-Verlag, 2000.
29. M. Turin, A. Deutsch, and G. Gonthier. La vérification des programmes d'ariane. *Pour la Science*, 243:21–22, Jan. 1998. (In French).
30. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the 28th Symposium on Principles of Programming Languages*, pages 27–40, January 2001.