Logic with Partial Functions in CVC Lite*

Sergey Berezin[1], Clark Barrett[2], Igor Shikanian[2], Marsha Chechik[3], Arie Gurfinkel[3], and David L. Dill[1]

[1]Stanford University, {berezin,dill}@stanford.edu
[2]New York University, {barrett,chikania}@cs.nyu.edu
[3]University of Toronto, {chechik,arie}@cs.toronto.edu

Abstract. Partial functions occur commonly in mathematics and formal verification. Surprisingly though, the problem of reasoning about partial functions has been largely ignored in mathematical logic and automated deduction research. A few approaches propose logics of partial functions and their translations to classical logic, but there is still no consensus on which is "the right" logic to use. In this paper, we propose using a three-valued Kleene logic, where partial functions return the "undefined" value when applied outside of their domains. The particular semantics are chosen according to the *principle of least surprise* to the user; if it is unclear what the value of the formula should be (from the classical point of view), its evaluation is undefined. We also enrich the logic with the if-then-else operator, which is necessary to model many large-scale formal verification problems, and devise a practically efficient extension of the existing validity checking algorithm implemented in the CVC Lite tool.

1 Introduction

First-order logic is an invaluable tool for modeling properties and behaviors of systems. Recent progress in automated reasoning and theorem proving has led to a broader and more successful application of logic as a tool for analyzing systems. Most standard approaches to theorem proving and deduction using first-order logic assume that all functions and predicates are total. However, many applications are more naturally modeled using partial functions and predicates.

Although it is generally agreed that a logic which can accommodate partial functions is useful for a wide variety of applications, there is general disagreement on which logic should be used. An overview of the different approaches can be found in [3, 6]. Of the approaches which take partiality seriously as opposed to attempting a work-around, there are two main alternatives. The first allows terms to be undefined, but requires that all formulas be either true or false. The unusual feature of this approach is that a predicate applied to an undefined

* This research was supported by GSRC contract DABT63-96-C-0097-P00005, and by National Science Foundation CCR-0121403. The content of this paper does not necessarily reflect the position or the policy of GSRC, NSF, or the Government, and no official endorsement should be inferred.

term is defined to be false. Although this logic preserves some nice features of classical logic (the deduction theorem, for instance), in a certain sense there is a loss of information because the undefinedness does not propagate to formulas. For example, if we assume the term $1/0$ is undefined, then the formula $\neg P(1/0)$ will be valid.

The second approach is based on Kleene's strong three-valued logic [7], and allows both terms and formulas to be undefined. This approach is more conservative in the sense that any formula which is valid in the second approach will be valid in the first approach, but there are some formulas, such as $\neg P(1/0)$, which may be valid in the first approach but will be undefined in the second.

Although a previous implementation of our theorem prover adopted the first approach [12], we prefer the second approach based on a *principle of least surprise*. That is, a formula should be valid only when there is no disagreement on whether that is a reasonable conclusion. This is particularly important in verification applications, as the integrity of a system may be judged by whether a theorem about the system is valid. Furthermore, it is our experience that any theorem which really should be valid can be formulated in such a way that it is valid according to this second approach.

A more pragmatic issue that must be dealt with is that most theorem-provers are based on classical logic. Various approaches have been advocated for modifying standard theorem-proving to accommodate logics with partial functions [5, 6, 8, 13]. However, we are interested in finding a method for supporting partiality without modifying the theorem prover. One way to do this is by building over- and under-approximations for the formula. This technique has been successfully applied for three-valued model-checking [2, 4].

PVS (Prototype Verification System [10]) uses a completely different approach which involves constructing and proving additional formulas called *type correctness conditions* (TCCs). The validity of TCCs guarantees that all the relevant terms and formulas are always defined. However, TCCs in PVS are computed in a somewhat *ad hoc* manner, which can yield surprising results. For example, it is possible to have a formula of the form $A \Rightarrow B$ with a valid TCC whose contrapositive $\neg B \Rightarrow \neg A$ has an invalid TCC.

In this paper, we propose a technique for checking the validity of a formula in three-valued logic by reducing the problem to checking two formulas in standard two-valued logic. Similarly to PVS, we construct a TCC formula whose validity implies that the original formula is always defined. After checking the TCC, we check the original formula. Both of these checks can be done using standard two-valued logic. Note that, unlike in PVS, our method is *precise* in the sense that if a TCC is invalid, the validity of the original formula is indeed undefined in the three-valued semantics.

The paper is organized as follows. Section 2 gives the syntax and semantics for our three-valued logic. Section 3 gives two fundamental theorems which justify the reduction to two-valued logic. Section 4 describes results obtained by implementing these ideas in the theorem prover CVC Lite, and Section 5 concludes.

2 Three-Valued Logic: Syntax and Semantics

2.1 Syntax

Let $\Sigma = (S, F, P, C, \Delta)$ be a signature, where $S = \{s_1, \ldots\}$ is a set of sorts, $F = \{f_1, \ldots\}$, $P = \{p_1, \ldots\}$ and $C = \{c_1, \ldots\}$ are sets of function predicate, and constant symbols, and $\Delta = \{\delta_q \mid q \in P \cup F\}$ is the set of *domain formulas* for function and predicate symbols (explained below). Each symbol has a type built out of the sorts in Σ. Define a term t as follows:

$$t ::= x \mid c \mid f(t_1, \ldots, t_n) \mid \textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif},$$

where x is a variable, c is a constant in Σ, $f(t_1, \ldots, t_n)$ is an application of a function symbol f from Σ to terms t_1 through t_n, and **if** ϕ **then** t_1 **else** t_2 **endif** is a conditional operator whose condition is a formula ϕ. A *formula* ϕ is defined as follows:

$$\begin{aligned}\phi ::= \textbf{true} \mid \textbf{false} \mid p(t_1, \ldots, t_n) \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi_1 \mid \\ \textbf{if } \phi_0 \textbf{ then } \phi_1 \textbf{ else } \phi_2 \textbf{ endif} \mid \forall x : s_i .\, \phi_1 \mid \exists x : s_i .\, \phi_1,\end{aligned}$$

where p is a predicate from Σ. We also use the following syntactic abbreviations:

$$\begin{aligned}\phi_1 \Rightarrow \phi_2 &= \neg\phi_1 \vee \phi_2 \\ \phi_1 \Leftrightarrow \phi_2 &= (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)\end{aligned}$$

It is important to distinguish the two versions of the if-then-else operator: the one for terms, and the one for formulas. Also note that the if-then-else operators are not expressible in terms of other operators or logical connectives in 3-valued logic.[1]

For our purposes, we will assume that included with every signature Σ is a set Δ of *domain formulas*, one for each function and predicate symbol in Σ. The domain formula for a function symbol f is a Σ-formula with k free variables where k is the arity of f and is denoted $\delta_f[x_1, \ldots, x_k]$. The domain formula for a predicate symbol p of arity k is defined similarly and is denoted $\delta_p[x_1, \ldots, x_k]$. An instantiation of a domain formula δ_f with terms $t_1, \ldots, t_k$ is written $\delta_f[t_1, \ldots, t_k]$ and denotes the result of replacing each x_i with t_i in the domain formula $\delta_f[x_1, \ldots, x_k]$.

Intuitively, the domain formula for f defines the set of points where f is defined. As a result, it is important that the domain formulas themselves always be defined. To ensure this, we require that if s is a function or predicate symbol appearing in a domain formula, then $\delta_s[x_1, \ldots, x_n] = \textbf{true}$.

[1] The obvious 2-valued translations $(\phi_0 \Rightarrow \phi_1) \wedge (\neg\phi_0 \Rightarrow \phi_2)$ and $(\phi_0 \wedge \phi_1) \vee (\neg\phi_0 \wedge \phi_2)$ are actually over- and under-approximations of the 3-valued operator **if** ϕ_0 **then** ϕ_1 **else** ϕ_2 **endif**.

2.2 Three-Valued Semantics with Partial Functions

Given a signature Σ, a model $M = \langle A, I \rangle$ of Σ is a pair of:

1. An S-indexed family of nonempty carrier sets $A = \{A_s \mid s \in S\}$ for each sort s in Σ.
2. A mapping I from each constant symbol $c : s$, function symbol $f : s_1 \times \cdots \times s_n \to s$, and predicate symbol $p : s_1 \times \cdots \times s_n \to \mathsf{bool}$ in Σ to an element $c^M \in A_s$, a partial function $f^M : A_{s_1} \times \cdots \times A_{s_n} \to A_s$ and a relation $p^M \subseteq A_{s_1} \times \cdots \times A_{s_n}$, respectively.

Given a model M and a variable assignment e which maps each variable to an element of some A_s, the value of an expression (a term or a formula) α is denoted $[\![\alpha]\!]_M e$ and is defined in Fig. 1. The value of a term may be an element of some A_s or a distinguished value $\bot_t$ not in any A_s. The value of a formula may be true, false, or $\bot_\phi$. We will use $\bot$ to represent both $\bot_t$ and $\bot_\phi$ since terms and formulas are always syntactically separated from each other, and the particular kind of $\bot$ is always clear from the context.

A model is required to satisfy the following additional condition imposed by the domain formulas Δ:

- $[\![\delta_f[x_1, \ldots, x_k]]\!]_M e = \mathsf{true}$ iff f^M is defined at $([\![x_1]\!]_M e, \ldots, [\![x_k]\!]_M e)$.

We say that two expressions α and β are logically equivalent, and write $\alpha \equiv \beta$ if $[\![\alpha]\!]_M e = [\![\beta]\!]_M e$ for every model M and variable assignment e.

2.3 Semantics of if-then-else

Notice that the interpretation of the if-then-else operator (for terms) is undefined if the condition is undefined, even if the other two children evaluate to the same value. One reason for this choice of the semantics is simply that it turns out to be practical in real applications. In real programs, if a partial function is applied to an argument outside of its domain, the program may crash or raise an exception; in other words, it results in an abnormal behavior. Therefore, detecting a possible $\bot$ value in the condition of an if-then-else provides the user with useful information, namely, that the program may crash during execution under certain conditions. For example, consider the following piece of C code:

```
int *p = malloc(sizeof(int));
int x = (*p > 0)? y : z;
```

In this example, the if-then-else operator (which is `(·)? ·:·` in C) will cause the program to crash if `p` happens to be `NULL`, even if $y = z$ in this particular program state. Here `*p` is a partial function defined over non-null pointers to integers, and returning an integer.

The logical if-then-else is defined similarly to the term if-then-else, so that DeMorgan law for negation and the if-lifting properties for any predicate symbol p in Σ are preserved:

$$\neg(\textbf{if } \phi \textbf{ then } \phi_1 \textbf{ else } \phi_2 \textbf{ endif}) \equiv \textbf{if } \phi \textbf{ then } \neg\phi_1 \textbf{ else } \neg\phi_2 \textbf{ endif}$$

$$p(\textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}) \equiv \textbf{if } \phi \textbf{ then } p(t_1) \textbf{ else } p(t_2) \textbf{ endif}$$

$$
\begin{aligned}
[\![c]\!]_M e &= c^M \\
[\![x]\!]_M e &= e(x) \\
[\![f(t_1,\ldots,t_n)]\!]_M e &= \begin{cases} f^M([\![t_1]\!]_M e,\ \ldots,\ [\![t_n]\!]_M e), & \text{if } [\![t_i]\!]_M \neq \bot \text{ for all } i \in [1..n] \\ & \text{and } [\![\delta_f[t_1,\ldots,t_n]]\!]_M e = \mathsf{true}; \\ \bot & \text{otherwise} \end{cases} \\
[\![\textbf{if } \phi \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}]\!]_M e &= \begin{cases} \bot, & \text{if } [\![\phi]\!]_M e = \bot \\ [\![t_1]\!]_M e, & \text{if } [\![\phi]\!]_M e = \mathsf{true} \\ [\![t_2]\!]_M e, & \text{if } [\![\phi]\!]_M e = \mathsf{false} \end{cases} \\
[\![\textbf{true}]\!]_M e &= \mathsf{true} \\
[\![\textbf{false}]\!]_M e &= \mathsf{false} \\
[\![p(t_1,\ldots,t_n)]\!]_M e &= \begin{cases} p^M([\![t_1]\!]_M e,\ \ldots,\ [\![t_n]\!]_M e), & \text{if } [\![t_i]\!]_M \neq \bot \text{ for all } i \in [1..n] \\ & \text{and } [\![\delta_p[t_1,\ldots,t_n]]\!]_M e = \mathsf{true}; \\ \bot & \text{otherwise} \end{cases} \\
[\![\phi_1 \wedge \phi_2]\!]_M e &= \begin{cases} \mathsf{true}, & \text{if } [\![\phi_1]\!]_M e = \mathsf{true} \text{ and } [\![\phi_2]\!]_M e = \mathsf{true}; \\ \mathsf{false} & \text{if } [\![\phi_1]\!]_M e = \mathsf{false} \text{ or } [\![\phi_2]\!]_M e = \mathsf{false}; \\ \bot & \text{otherwise} \end{cases} \\
[\![\phi_1 \vee \phi_2]\!]_M e &= \begin{cases} \mathsf{true}, & \text{if } [\![\phi_1]\!]_M e = \mathsf{true} \text{ or } [\![\phi_2]\!]_M e = \mathsf{true}; \\ \mathsf{false} & \text{if } [\![\phi_1]\!]_M e = \mathsf{false} \text{ and } [\![\phi_2]\!]_M e = \mathsf{false}; \\ \bot & \text{otherwise} \end{cases} \\
[\![\neg\phi]\!]_M e &= \begin{cases} \mathsf{true}, & \text{if } [\![\phi]\!]_M e = \mathsf{false} \\ \mathsf{false} & \text{if } [\![\phi]\!]_M e = \mathsf{true} \\ \bot & \text{if } [\![\phi]\!]_M e = \bot \end{cases} \\
[\![\textbf{if } \phi \textbf{ then } \phi_1 \textbf{ else } \phi_2 \textbf{ endif}]\!]_M e &= \begin{cases} \bot, & \text{if } [\![\phi]\!]_M e = \bot \\ [\![\phi_1]\!]_M e, & \text{if } [\![\phi]\!]_M e = \mathsf{true} \\ [\![\phi_2]\!]_M e, & \text{if } [\![\phi]\!]_M e = \mathsf{false} \end{cases} \\
[\![\forall x : s.\, \phi]\!]_M e &= \begin{cases} \mathsf{true}, & \text{if for all } a \in A_s\text{: } [\![\phi]\!]_M e[x \leftarrow a] = \mathsf{true}; \\ \mathsf{false}, & \text{if for some } a \in A_s\text{: } [\![\phi]\!]_M e[x \leftarrow a] = \mathsf{false}; \\ \bot & \text{otherwise} \end{cases} \\
[\![\exists x : s.\, \phi]\!]_M e &= \begin{cases} \mathsf{true}, & \text{if for some } a \in A_s\text{: } [\![\phi]\!]_M e[x \leftarrow a] = \mathsf{true}; \\ \mathsf{false}, & \text{if for all } a \in A_s\text{: } [\![\phi]\!]_M e[x \leftarrow a] = \mathsf{false}; \\ \bot & \text{otherwise.} \end{cases}
\end{aligned}
$$

Fig. 1. Three-valued semantics: $[\![\phi]\!]_M e$.

2.4 Three-Valued Validity

The three-valued semantics can be extended to validity of formulas in the following way. A formula is considered *valid*, if in all models M and for all variable assignments e, $[\![\phi]\!]_M e = \mathsf{true}$. A formula is *invalid* if there is at least one such model M and one such assignment e that $[\![\phi]\!]_M e = \mathsf{false}$. Otherwise (if the formula always evaluates to either true or $\perp$) the validity is undefined. We denote the three-valued validity as $\models \phi$, which may hold, not hold, or be undefined.

3 Reduction from Three-Valued Logic to Two-Valued Logic

Suppose we wish to determine the three-valued validity of some Σ-formula ϕ. Our general strategy is first to compute a formula called a Type Correctness Condition (TCC) which can be used to check whether ϕ can ever be undefined. If this check succeeds, that is, ϕ is always defined, we can then check the original formula. Both of these checks can be done using standard two-valued logic. To justify this claim, we first introduce TCCs and then prove that they have the required properties.

3.1 Type Correctness Conditions (TCCs)

A *Type Correctness Condition* for a formula ϕ of our three-valued logic is a formula which does not involve any partial symbols (symbols whose domain formulas are not **true**) and which evaluates to true iff ϕ is not undefined.

First, observe that if we have a term $f(x)$, then by definition its TCC is simply $\delta_f[x]$. We can generalize this to arbitrary terms or formulas quite easily. Figure 2 gives a recursive definition of $\mathcal{D}_\phi$, the TCC for an arbitrary formula ϕ.

The TCC not only identifies whether or not the formula ϕ is defined, but it can also be used to reduce the three-valued evaluation of ϕ to an evaluation in standard two-valued logic with total models.

Suppose M is a partial model of Σ. Let $\hat{\Sigma}$ be equivalent to Σ except that all of its domain formulas are **true**. Let $\hat{M}$ be a model of $\hat{\Sigma}$ which agrees with M wherever M is defined. Finally, let $[\![S]\!]^2_{\hat{M}} e$ denote the evaluation of an expression S in the total model $\hat{M}$ using standard two-valued semantics. The following two theorems justify our use of TCCs and are proved in the appendix.

Theorem 1. *Let S be any Σ-term or formula, and let $\hat{M}$ denote an arbitrary extension of a Σ-model M to a total model over $\hat{\Sigma}$. Then:*

$$[\![\mathcal{D}_S]\!]^2_{\hat{M}} e = \mathsf{true} \quad \Rightarrow \quad [\![S]\!]^2_{\hat{M}} e = [\![S]\!]_M e$$

Theorem 2. *Let S be any Σ-term or formula, and let $\hat{M}$ denote an arbitrary extension of a Σ-model M to a total model over $\hat{\Sigma}$. Then:*

$$
\begin{aligned}
\mathcal{D}_x &\equiv \mathbf{true} \\
\mathcal{D}_c &\equiv \mathbf{true} \\
\mathcal{D}_{f(t_1,\ldots,t_n)} &\equiv \delta_f[t_1,\ldots,t_n] \wedge \bigwedge_{i=1}^{n} \mathcal{D}_{t_i} \\
\mathcal{D}_{\mathbf{if}\ \phi\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{endif}} &\equiv \mathcal{D}_\phi \wedge (\mathbf{if}\ \phi\ \mathbf{then}\ \mathcal{D}_{t_1}\ \mathbf{else}\ \mathcal{D}_{t_2}\ \mathbf{endif}) \\
\mathcal{D}_{\mathbf{if}\ \phi\ \mathbf{then}\ \phi_1\ \mathbf{else}\ \phi_2\ \mathbf{endif}} &\equiv \mathcal{D}_\phi \wedge (\mathbf{if}\ \phi\ \mathbf{then}\ \mathcal{D}_{\phi_1}\ \mathbf{else}\ \mathcal{D}_{\phi_2}\ \mathbf{endif}) \\
\mathcal{D}_{p(t_1,\ldots,t_n)} &\equiv \delta_p[t_1,\ldots,t_n] \wedge \bigwedge_{i=1}^{n} \mathcal{D}_{t_i} \\
\mathcal{D}_{\neg\phi} &\equiv \mathcal{D}_\phi \\
\mathcal{D}_{\phi_1\wedge\phi_2} &\equiv (\mathcal{D}_{\phi_1} \wedge \neg\phi_1) \vee (\mathcal{D}_{\phi_2} \wedge \neg\phi_2) \vee (\mathcal{D}_{\phi_1} \wedge \mathcal{D}_{\phi_2}) \\
\mathcal{D}_{\phi_1\vee\phi_2} &\equiv (\mathcal{D}_{\phi_1} \wedge \phi_1) \vee (\mathcal{D}_{\phi_2} \wedge \phi_2) \vee (\mathcal{D}_{\phi_1} \wedge \mathcal{D}_{\phi_2}) \\
\mathcal{D}_{\forall x.\,\phi} &\equiv (\exists x.\,\mathcal{D}_\phi \wedge \neg\phi) \vee (\forall x.\,\mathcal{D}_\phi) \\
\mathcal{D}_{\exists x.\,\phi} &\equiv (\exists x.\,\mathcal{D}_\phi \wedge \phi) \vee (\forall x.\,\mathcal{D}_\phi)
\end{aligned}
$$

Fig. 2. Definition of TCCs for terms and formulas.

$$[\![\mathcal{D}_S]\!]^2_{\hat{M}} e = \mathsf{false} \quad \Rightarrow \quad [\![S]\!]_M e = \bot$$

Another important property of $\mathcal{D}_\phi$ is that if ϕ is represented as a DAG, then the worst-case size of $\mathcal{D}_\phi$ as a DAG is linear in the size of ϕ. This is because at each step of the computation of $\mathcal{D}_\phi$, only a constant number of additional nodes are introduced in addition to those already in ϕ. This is critical for many applications where the size of ϕ may be very large, and even a quadratic increase over the size of ϕ may be unacceptable.

In practice, things are usually even better. Often, the instances of partial functions are relatively sparse, and $\mathcal{D}_\phi$ is very small relative to ϕ.

3.2 Checking Validity

Theorems 1 and 2 and the procedure for constructing $\mathcal{D}_\phi$ effectively provide an algorithm for checking the three-valued validity of a Σ-formula ϕ.

To verify that such a formula is three-valued valid, it suffices to check that $\mathcal{D}_\phi$ is two-valued valid and that ϕ itself is two-valued valid in a set of total models which includes at least one model $\hat{M}$ for each three-valued Σ-model M.

4 Implementation in CVC Lite

We have implemented the three-valued Kleene semantics described above in our tool, CVC Lite [14]. CVC Lite is an efficient *validity checker* for first-order logic over several interpreted theories. The tool takes a logical formula ϕ as an input and returns *Valid* or *Invalid*.

CVC Lite is based on combination of decision procedures [1, 9, 11], and currently supports several theories, including uninterpreted functions, arrays, and linear real arithmetic. It also has some limited support for quantifiers.

The input language of CVC Lite is typed, with support for *predicate subtyping*, that is, types of the form $\tau' = \{x : \tau \mid \phi(x)\}$, where τ is a type, and $\phi(x)$ is a quantifier-free formula over the variable x. The type τ' is called a subtype of τ with type predicate ϕ. The values of any term t of type τ' are restricted to those of type τ which also satisfy $\phi(t)$. For example, the division operator over reals is a function of type:

$$\mathsf{div} : \mathsf{real} \times \{y : \mathsf{real} \mid y \neq 0\} \rightarrow \mathsf{real}.$$

Note that such a function can also be considered as a partial function from ($\mathsf{real} \times \mathsf{real}$) to real which is undefined when the second argument is 0.

In fact, since precise typechecking in the presence of predicate subtypes involves manipulating arbitrary logical formulas, typechecking proper is restricted to matching only the *base types* of function arguments and terms (that is, the maximal supertypes). In particular, $\mathsf{div}(x, 0)$ will be type-correct, since 0 is of type real, which is the base type of the second argument.

The more precise checking of whether an input formula ϕ is always defined is done separately by computing $\mathcal{D}_\phi$ and checking for its validity. If the validity of $\mathcal{D}_\phi$ cannot be established, CVC Lite returns a type error.

If $\mathcal{D}_\phi$ is valid, then ϕ is checked for validity as if it is a formula in the classical two-valued logic where all functions are total. In particular, the decision procedure for arithmetic can extend div to be a total function without compromising the correctness of the result. By Theorem 1, the result of such a check is consistent with the validity of ϕ in the original three-valued logic.

For example, consider the following formula:

$$\phi_0 \equiv \mathsf{div}(x, y) = \mathsf{div}(x, y),$$

where x and y are uninterpreted constants of type real. This formula is clearly valid in the classical two-valued logic. However, the TCC for this formula, $\mathcal{D}_{\phi_0} \equiv y \neq 0$, is not valid, and therefore, the validity of ϕ_0 in the three-valued semantics in undefined. CVC Lite detects this and returns a type error.

However, adding a condition that $y \neq 0$ makes the formula valid in three-valued semantics:

$$\phi_1 \equiv y \neq 0 \Rightarrow \mathsf{div}(x, y) = \mathsf{div}(x, y),$$

since its TCC:

$$\mathcal{D}_{\phi_1} \equiv (\mathbf{true} \wedge \neg(y \neq 0)) \vee (y \neq 0 \wedge \mathsf{div}(x, y) = \mathsf{div}(x, y) \vee (\mathbf{true} \wedge y \neq 0)$$

is trivially true due to the first and the last disjuncts.[2]

[2] Recall, that $\psi_1 \Rightarrow \psi_2 \equiv (\neg\psi_1 \vee \psi_2)$, and the TCC for the implication is the same as the TCC for the corresponding disjunction.

In fact, the contrapositive of that formula is also valid in the three-valued semantics for exactly the same reason, even though this version of the formula may look somewhat startling to a mathematician:

$$\phi_2 \equiv \mathsf{div}(x, y) \neq \mathsf{div}(x, y) \Rightarrow y = 0.$$

CVC Lite correctly proves that both formulas are indeed *Valid*.

From the implementation point of view, the approach was extremely easy to code: it took only a few hours to implement and debug. Furthermore, checking TCCs does not noticeably affect the performance of the tool on classical examples (without subtypes or partial functions), as the TCCs of such formulas immediately simplify to **true**. How it affects the performance on large examples with partial functions still remains to be seen. However, we expect that the typical overhead will be very low, since TCCs often have a much simpler structure than the actual formulas.

5 Conclusion

We have proposed a three-valued Kleene logic for use in applications which are most naturally modeled using partial functions. We have shown how the question of checking validity of formulas in this logic can be solved by checking the formula and a Type Correctness Condition whose size is linear in the size of the original formula. Both of these checks can be done using standard two-valued semantics.

We have a prototype implementation of these ideas in the theorem-prover CVC Lite. Our implementation was able to determine three-valued validity and invalidity of small examples. We plan to use these ideas to test larger examples in CVC Lite.

Future work includes using these ideas to develop a more general notion of validity in the presence of theories with sorts and sub-sorts and dealing with non-strict functions and predicates (those which, like the Boolean operators $\wedge$ and $\vee$ do not have the property that if one of their children evaluates to $\perp$, then the whole expression evaluates to $\perp$).

References

1. C. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First -Order Theories.* PhD thesis, Stanford University, 2003.
2. G. Bruns and P. Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In *Proceedings of Proceedings of 11th International Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 274–287, Trento, Italy, 1999. Springer.
3. William M. Farmer. A Partial Functions Version of Church's Simple Theory of Types. *The Journal of Symbolic Logic*, 55(3):1269–1291, 1990.
4. A. Gurfinkel and M. Chechik. "Multi-Valued Model-Checking via Classical Model-Checking". In *Proceedings of 14th International Conference on Concurrency Theory (CONCUR'03)*, volume 2761 of *LNCS*, September 2003.

5. M. Kerber and M. Kohlhase. A Mechanization of Strong Kleene Logic for Partial Functions. In A. Bundy, editor, *12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 371–385. Springer Verlag, 1994.
6. M. Kerber and M. Kohlhase. Mechanising Partiality without Re-Implementation. In *21st Annual German Conference on Artificial Intelligence*, volume 1303 of *LNAI*, pages 123–134. Springer Verlag, 1997.
7. S. C. Kleene. *Introduction to Metamathematics*. New York: Van Nostrand, 1952.
8. Francisca Lucio-Carrasco and Antonio Gavilanes-Franco. A First Order Logic for Partial Functions. In *Proceedings STACS'89*, volume 349 of *LNCS*, pages 47–58. Springer, 1989.
9. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.
10. N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, 1993. Also appears in Tutorial Notes, *Formal Methods Europe'93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.
11. R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.
12. Aaron Stump. *Checking Validities and Proofs with CVC and flea*. PhD thesis, Stanford University, 2002.
13. Pawel Tichy. Foundations of partial type theory. *Reports on Mathematical Logic*, 14:59–72, 1982.
14. CVC Lite (tool). URL: `http://verify.stanford.edu/CVCL`. The new version of the Cooperating Validity Checker.