Efficient Well-Definedness Checking

Ádám Darvas, Farhad Mehta, and Arsenii Rudich

ETH Zurich, Switzerland,
{adam.darvas,farhad.mehta,arsenii.rudich}@inf.ethz.ch

Abstract. Formal specifications often contain partial functions that may lead to ill-defined terms. A common technique to eliminate ill-defined terms is to require well-definedness conditions to be proven. The main advantage of this technique is that it allows us to reason in a two-valued logic even if the underlying specification language has a three-valued semantics. Current approaches generate well-definedness conditions that grow exponentially with respect to the input formula. As a result, many tools prove shorter, but stronger approximations of these well-definedness conditions instead.
We present a procedure which generates well-definedness conditions that grow linearly with respect to the input formula. The procedure has been implemented in the Spec# verification tool. We also present empirical results that demonstrate the improvements made.

1 Introduction

Formal specifications often allow terms to contain applications of partial functions, such as division $x\,/\,y$ or factorial $\mathit{fact}(z)$. However, it is not clear what value $x\,/\,y$ yields if y is 0, or what value $\mathit{fact}(z)$ yields if z is negative. Specification languages need to handle *ill-defined terms*, that is, either have to define the semantics of partial-function applications whose arguments fall outside their domains or have to eliminate such applications.

One of the standard approaches to handle ill-defined terms is to define a three-valued semantics [22] by considering ill-defined terms to have a special value, *undefined*, denoted by $\bot$. That is, both $x\,/\,0$ and $\mathit{fact}(-5)$ are considered to evaluate to $\bot$. In order to reason about specifications with a three-valued semantics, undefinedness is lifted to formulas by extending their denoted truth values to $\{\mathbf{true}, \mathbf{false}, \bot\}$.

A common technique to reason about specifications with a three-valued semantics is to *eliminate* ill-defined terms before starting the actual proof. *Well-definedness conditions* are generated, whose validity ensures that all formulas at hand can be evaluated to either **true** or **false**. That is, once the well-definedness conditions have been discharged, $\bot$ is guaranteed to never be encountered.

The advantage of the technique is that both the well-definedness conditions and the actual proof obligations are to be proven in classical two-valued logic, which is simpler, better understood, more widely used, and has better automated tool support [30] than three-valued logics.

The technique of eliminating ill-defined terms in specifications by generating well-definedness conditions is used in several approaches, for instance, B [2], PVS [13], CVC Lite [6], and ESC/Java2 [21].

Motivation. A drawback of this approach is that well-definedness conditions can be very large, causing significant time overhead in the proof process. As an example, consider the following formula:

$$x\,/\,y = c_1 \;\wedge\; \mathit{fact}(y) = c_2 \;\wedge\; y > 5 \tag{1}$$

where x and y are variables, and c_1 and c_2 are constants. The formula is well-defined, that is, it always evaluates to either **true** or **false**. This can be justified by a case split on the third conjunct, which is always well-defined:

1. if the third conjunct evaluates to **true**, then the division and factorial functions are known to be applied within their domains, and thus, the first and second conjuncts can be evaluated to **true** or **false**. This means that the whole formula can be evaluated to either **true** or **false**.
2. if the third conjunct evaluates to **false**, then the whole formula evaluates to **false** (according to the semantics we use in the paper).

The literature [8, 27, 3, 9] proposes the procedure $\mathcal{D}$ to generate well-definedness conditions. The procedure is *complete* [8, 9], that is, the well-definedness condition generated from a formula is provable if and only if the formula is well-defined. Procedure $\mathcal{D}$ would generate the following condition for (1):

$$\begin{array}{l}(y \neq 0 \wedge (y \geq 0 \vee (y \geq 0 \wedge \mathit{fact}(y) \neq c_2) \vee y \leq 5)) \;\vee \\ (y \neq 0 \wedge x/y \neq c_1) \;\vee \\ ((y \geq 0 \vee (y \geq 0 \wedge \mathit{fact}(y) \neq c_2) \vee y \leq 5) \wedge \neg(\mathit{fact}(y) \neq c_2 \wedge y > 5))\end{array}$$

As expected, the condition is provable. However, the size of the condition is striking, given that the original formula contained only three sub-formulas and two partial-function applications. In fact, procedure $\mathcal{D}$ yields well-formedness conditions that *grow exponentially* with respect to the input formula. This is a major problem for tools that have to prove well-definedness of considerably larger formulas than (1), for instance, the well-definedness conditions for B models, as presented in [8].

Due to the exponential blow-up of well-definedness conditions, the $\mathcal{D}$ procedure is not used in practice [8, 27, 3]. Instead, another procedure $\mathcal{L}$ is used, which generates much smaller conditions with linear growth, but which is *incomplete*. That is, the procedure may generate unprovable well-definedness conditions for well-defined formulas. This is the case with formula (1), for which the procedure would yield the following unprovable condition:

$$y \neq 0 \wedge (x/y = c_1 \Rightarrow y \geq 0)$$

Incompleteness of the procedure originates from its "sensitivity" to the order of sub-formulas. For instance, after proper re-ordering of the sub-formulas of (1), the procedure would yield a provable condition. This may be tedious for large formulas and may appear unnatural to users who are familiar with logics in which the order of sub-formulas is irrelevant for proof. Furthermore, there are situations (for instance, our example in Section 4) where such a manual re-ordering cannot be done.

Contributions. Our main contribution is a new procedure $\mathcal{Y}$, which unifies the advantages of $\mathcal{D}$ and $\mathcal{L}$, while eliminating their weaknesses. That is, (1) $\mathcal{Y}$ yields well-definedness conditions that *grow linearly* with respect to the size of the input formula, and (2) $\mathcal{Y}$ is equivalent to $\mathcal{D}$, and therefore complete and insensitive to the order of sub-formulas. To our knowledge, this is the first procedure that has *both* of these two properties.

The definition of the new procedure is very intuitive and straightforward. We prove that it is equivalent with $\mathcal{D}$ in two ways: (1) in a syntactical manner, as $\mathcal{D}$ was derived in [3], and (2) in a semantical way, as $\mathcal{D}$ was introduced in [8].

We have implemented the new procedure in the Spec# verification tool [5] in the context of the well-formedness checking of method specifications [26]. We have compared our procedure with $\mathcal{D}$ and $\mathcal{L}$ using two automated theorem provers. The empirical results clearly show that not only the size of generated well-definedness conditions are significantly smaller than what $\mathcal{D}$ produces, but the time to prove validity of the conditions is also decreased by the use of $\mathcal{Y}$. Furthermore, our results show that the performance of $\mathcal{Y}$ is also better than that of $\mathcal{L}$ in terms of the size of generated conditions.

Outline. The rest of the paper is structured as follows. Section 2 formally defines procedures $\mathcal{D}$ and $\mathcal{L}$, and highlights their main differences. Section 3 presents our main contribution: the $\mathcal{Y}$ procedure and the proof of its equivalence with $\mathcal{D}$. Section 4 demonstrates the improvements of our approach through empirical results. We discuss related work in Section 5 and conclude in Section 6.

2 Eliminating Ill-definedness

The main idea behind the technique of eliminating ill-definedness in specifications is to reduce the three-valued domain to a two-valued domain by ensuring that $\perp$ is never encountered. $\mathcal{D}$ is used for this purpose. Hoogewijs introduced $\mathcal{D}$ in the form of the logical connective Δ in [19], and proposed a first-order calculus, which includes this connective. Later, $\mathcal{D}$ was reformulated as a formula transformer, for instance, in [8, 3, 9] for the above syntax. $\mathcal{D}$ takes a formula ϕ and produces another formula $\mathcal{D}(\phi)$. The interpretation of the formula $\mathcal{D}(\phi)$ in two-valued logic is **true** if and only if the interpretation of ϕ in three-valued logic is different from $\perp$.

$$
\begin{array}{llll}
\mathit{Term} ::= \mathit{Var} & & \mathit{Formula} ::= P(t_1, \ldots, t_n) & \\
\quad\mid\ f(t_1, \ldots, f_n) & & \quad\mid\ \mathit{true} & \mid\ \mathit{false} \\
 & & \quad\mid\ \neg\phi & \\
 & & \quad\mid\ \phi_1 \wedge \phi_2 & \mid\ \phi_1 \vee \phi_2 \\
 & & \quad\mid\ \forall\, x.\ \phi & \mid\ \exists\, x.\ \phi
\end{array}
$$

Fig. 1. Syntax of terms and formulas we consider in this paper.

$$
\begin{array}{lll}
\delta(\mathit{Var}) & \triangleq & \mathit{true} \\
\delta(f(e_1, \ldots, e_n)) & \triangleq & d_f(e_1, \ldots, e_n)\ \wedge\ \bigwedge_{i=1}^{n} \delta(e_i) \\
\mathcal{D}(P(e_1, \ldots, e_n)) & \triangleq & \bigwedge_{i=1}^{n} \delta(e_i) \\
\mathcal{D}(\mathit{true}) & \triangleq & \mathit{true} \\
\mathcal{D}(\mathit{false}) & \triangleq & \mathit{true} \\
\mathcal{D}(\neg\phi) & \triangleq & \mathcal{D}(\phi) \\
\mathcal{D}(\phi_1 \wedge \phi_2) & \triangleq & (\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2))\ \vee\ (\mathcal{D}(\phi_1) \wedge \neg\phi_1)\ \vee\ (\mathcal{D}(\phi_2) \wedge \neg\phi_2) \\
\mathcal{D}(\phi_1 \vee \phi_2) & \triangleq & (\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2))\ \vee\ (\mathcal{D}(\phi_1) \wedge \phi_1)\ \vee\ (\mathcal{D}(\phi_2) \wedge \phi_2) \\
\mathcal{D}(\forall\, x.\ \phi) & \triangleq & \forall\, x.\, \mathcal{D}(\phi)\ \vee\ \exists\, x.\, (\mathcal{D}(\phi) \wedge \neg\phi) \\
\mathcal{D}(\exists\, x.\ \phi) & \triangleq & \forall\, x.\, \mathcal{D}(\phi)\ \vee\ \exists\, x.\, (\mathcal{D}(\phi) \wedge \phi)
\end{array}
$$

Fig. 2. Definition of the δ and $\mathcal{D}$ operators as given by Behm et al. [8].

In order to have a basis for formal definitions, we define the syntax of terms and formulas that we consider in this paper. We follow the standard syntax-definition given in Figure 1. Throughout the paper we use **true**, **false**, and $\perp$ to denote the semantic truth values, and *true* and *false* to refer to the syntactic entities.

2.1 Defining the $\mathcal{D}$ Operator

The definition of $\mathcal{D}$ is given in Figure 2. Operator δ handles terms and $\mathcal{D}$ handles formulas. A variable is always well-defined. Application of function f is well-defined if and only if f's *domain restriction* d_f holds and all parameters e_i are well-defined. Each function is associated with a domain restriction, which is a predicate that represents the domain of the function. Such predicates should only contain total-function applications. For instance, the domain restriction of the factorial function is that the parameter is non-negative.

A predicate is well-defined if and only if all parameters are well-defined. Note that this definition assumes predicates to be total. Although an extension to partial predicates is straightforward, we use this definition for simplicity and to have a direct comparison of our approach to [8]. Constants *true* and *false* are always well-defined. Well-definedness of logical connectives is defined according to Strong Kleene connectives [22]. For instance, as the truth table in Figure 3(a)

$\wedge$	**true**	**false**	$\perp$
true	**true**	**false**	$\perp$
false	**false**	**false**	**false**
$\perp$	$\perp$	**false**	$\perp$

(a) Strong Kleene

$\wedge$	**true**	**false**	$\perp$
true	**true**	**false**	$\perp$
false	**false**	**false**	**false**
$\perp$	$\perp$	$\perp$	$\perp$

(b) McCarthy

Fig. 3. Kleene's and McCarthy's interpretation of conjunction.

shows, a conjunction is well-defined if and only if either (1) both conjuncts are well-defined, or (2) if one of the conjuncts is well-defined and evaluates to **false**. Intuitively, in case (1) the classical two-valued evaluation can be applied, while in case (2) the truth value of the conjunction is **false** independently of the well-definedness and value of the other conjunct.

Well-definedness of universal quantification can be thought of as the generalization of the well-definedness of conjunction. Disjunction and existential quantification are the duals of conjunction and universal quantification, respectively. Soundness and completeness of $\mathcal{D}$ was proven in [19, 8, 9].

2.2 An Approximation of the $\mathcal{D}$ Operator

As mentioned before in Section 1, the problem with the $\mathcal{D}$ operator is that it yields well-definedness conditions that grow exponentially with respect to the size of the input formula. This problem has been recognized in several approaches, for instance, in B [3] and PVS [27]. As a consequence, these approaches use a simpler, but stricter operator $\mathcal{L}$ [8, 3] for computing well-definedness conditions. The definition of $\mathcal{L}$ differs from that of $\mathcal{D}$ only for the following connectives:[1]

$$\begin{aligned} \mathcal{L}(\phi_1 \wedge \phi_2) &\triangleq \mathcal{L}(\phi_1) \wedge (\phi_1 \Rightarrow \mathcal{L}(\phi_2)) & \qquad \mathcal{L}(\forall\, x.\ \phi) &\triangleq \forall\, x.\, \mathcal{L}(\phi) \\ \mathcal{L}(\phi_1 \vee \phi_2) &\triangleq \mathcal{L}(\phi_1) \wedge (\neg\phi_1 \Rightarrow \mathcal{L}(\phi_2)) & \qquad \mathcal{L}(\exists\, x.\ \phi) &\triangleq \forall\, x.\, \mathcal{L}(\phi) \end{aligned}$$

Looking at the definition, we can see that $\mathcal{L}$ yields well-definedness conditions that grow linearly with respect to the input formula. This is a great advantage over $\mathcal{D}$. However, the $\mathcal{L}$ operator is stronger than $\mathcal{D}$, that is, $\mathcal{L}(\phi) \Rightarrow \mathcal{D}(\phi)$ holds, but $\mathcal{D}(\phi) \Rightarrow \mathcal{L}(\phi)$ does not necessarily hold, as shown for formula (1) in Section 1. This means that we lose completeness with the use of $\mathcal{L}$.

For quantifiers, $\mathcal{L}$ requires that the quantified formula is well-defined for all instantiations of the quantified variable. As a result, a universal quantification may be considered ill-defined although an instance is known to evaluate to **false**. Similarly, an existential quantification may also be considered ill-defined although an instance is known to evaluate to **true**. The $\mathcal{D}$ operator takes these "short-circuits" into account.

[1] Although our formula-syntax does not contain implication, we use it below to keep the intuition behind the definition.

The other source of incompleteness originates from defining conjunction and disjunction according to McCarthy's interpretation [24], which evaluates formulas *sequentially.* That is, if the first operand of a connective is $\bot$, then the result is defined to be $\bot$, independently of the second operand. The truth table in Figure 3(b) presents McCarthy's interpretation of conjunction. The only difference from Kleene's interpretation is in the interpretation of $\bot \wedge$ **false**, which yields $\bot$. This reveals the most important difference between the two interpretations: in McCarthy's interpretation conjunction and disjunction are not commutative.

As a consequence, for instance, $\phi_1 \;\wedge\; \phi_2$ may be considered ill-defined, although $\phi_2 \;\wedge\; \phi_1$ is considered well-defined. Such cases might come unexpected to users who are used to classical logic where conjunction and disjunction are commutative.

In most cases this incompleteness issue can be resolved by manually re-ordering sub-formulas. However, as pointed out by Rushby et al. [27], the manual re-ordering of sub-formulas is not an option when specifications are automatically generated from some other representation. Furthermore, Cheng and Jones [12], and Rushby et al. [27] give examples for which even manual re-ordering does not help, and well-defined formulas are inevitably rejected by $\mathcal{L}$.

3 An Efficient Equivalent of the $\mathcal{D}$ Operator

In this section we present our main contribution: a new procedure $\mathcal{Y}$ that yields considerably smaller well-definedness conditions than $\mathcal{D}$, and that retains completeness. We prove equivalence of $\mathcal{Y}$ and $\mathcal{D}$ in two ways: (1) we syntactically derive the definition of $\mathcal{Y}$, (2) using three-valued interpretation we prove by induction that the definition of $\mathcal{Y}$ is equivalent to that of $\mathcal{D}$. Both proofs demonstrate the intuitive and simple nature of $\mathcal{Y}$'s definition.

3.1 Syntactical Derivation of $\mathcal{Y}$

We introduce two new formula transformers $\mathcal{T}$ and $\mathcal{F}$, and define them as follows:

$$\mathcal{T}(\phi) \;\triangleq\; \mathcal{D}(\phi) \;\wedge\; \phi \qquad \text{and} \qquad \mathcal{F}(\phi) \;\triangleq\; \mathcal{D}(\phi) \;\wedge\; \neg\,\phi$$

That is, $\mathcal{T}(\phi)$ yields **true** if and only if ϕ is well-defined and evaluates to **true**. Analogously, $\mathcal{F}(\phi)$ yields **true** if and only if ϕ is well-defined and evaluates to **false**. From the definitions the following theorem follows.

Theorem 1. $\mathcal{D}(\phi) \;\Leftrightarrow\; \mathcal{T}(\phi) \;\vee\; \mathcal{F}(\phi)$
Proof. $\mathcal{D}(\phi) \;\Leftrightarrow\; \mathcal{D}(\phi) \;\wedge\; (\phi \;\vee\; \neg\,\phi) \;\Leftrightarrow\; (\mathcal{D}(\phi) \;\wedge\; \phi) \;\vee\; (\mathcal{D}(\phi) \;\wedge\; \neg\,\phi) \;\Leftrightarrow\; \mathcal{T}(\phi) \;\vee\; \mathcal{F}(\phi)$ $\square$

Intuitively, the theorem expresses that formula ϕ is well-defined if and only if ϕ evaluates either to **true** or to **false**. This directly corresponds to the interpretation of $\mathcal{D}$ given by Hoogewijs [19].

From the definitions of $\mathcal{T}$ and $\mathcal{F}$, the equivalences presented in Figure 4 can be derived. To demonstrate the simplicity of these derivations, we give the derivation of $\mathcal{T}(\phi_1 \wedge \phi_2)$ and $\mathcal{F}(\forall x.\ \phi)$:

$$\begin{aligned}
&\mathcal{T}(\phi_1 \wedge \phi_2) \;\Leftrightarrow\; \mathcal{D}(\phi_1 \wedge \phi_2) \wedge \phi_1 \wedge \phi_2 \;\Leftrightarrow\\
&((\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2)) \vee (\mathcal{D}(\phi_1) \wedge \neg\phi_1) \vee (\mathcal{D}(\phi_2) \wedge \neg\phi_2)) \wedge \phi_1 \wedge \phi_2 \;\Leftrightarrow\\
&\mathcal{D}(\phi_1) \wedge \mathcal{D}(\phi_2) \wedge \phi_1 \wedge \phi_2 \;\Leftrightarrow\; \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{F}(\forall x.\ \phi) \;\Leftrightarrow\; \mathcal{D}(\forall x.\ \phi) \wedge \neg\forall x.\ \phi \;\Leftrightarrow\\
&(\forall x.\ \mathcal{D}(\phi) \vee (\exists x.\ (\mathcal{D}(\phi) \wedge \neg\phi))) \wedge \exists x.\ \neg\phi \;\Leftrightarrow\\
&\exists x.\ (\mathcal{D}(\phi) \wedge \neg\phi) \;\Leftrightarrow\; \exists x.\ \mathcal{F}(\phi)
\end{aligned}$$

The derived equivalences are very intuitive. Both $\mathcal{T}$ and $\mathcal{F}$ reflect the standard two-valued interpretation of formulas. For instance, $\mathcal{F}$ essentially realizes de Morgan's laws. The handling of terms is the same as before using the δ operator. Note that $\mathcal{T}$ and $\mathcal{F}$ are mutually recursive in the equivalences. Termination of the mutual application of the operators is trivially guaranteed: the size of formulas yields the measure for termination.

The more involved semantic proof of equivalence is presented in the appendix. The proof, in particular **Lemma 4**, highlights the intuition behind $\mathcal{Y}$'s definition.

The definition of our new procedure $\mathcal{Y}$, based on **Theorem 1**, is the following:

$$\mathcal{Y}(\phi) \;\triangleq\; \mathcal{T}(\phi) \;\vee\; \mathcal{F}(\phi)$$

It is easy to see that (1) our procedure begins by duplicating the size of the input formula ϕ, and (2) afterwards applies operators $\mathcal{T}$ and $\mathcal{F}$ that yield formulas that are linear in size with respect to their input formulas.

That is, overall our procedure yields well-definedness conditions that grow linearly with respect to the size of the input formula. This is a significant improvement over $\mathcal{D}$ which yields formulas that are exponential in size with respect to the input formula. Intuitively, this improvement can be explained as follows: $\mathcal{D}$ makes case distinctions on the well-definedness of sub-formulas at each step of its application, whereas $\mathcal{Y}$ only performs a single initial case distinction on the validity of the entire formula. In spite of this difference, our procedure is equivalent to $\mathcal{D}$, thus it is symmetric, as opposed to $\mathcal{L}$.

4 Implementation and Empirical Results

We have implemented a well-formedness checker in the context of the verification of object-oriented programs. Our implementation extends the Spec# verification tool [5] by a new module that performs well-definedness and well-foundedness checks on specifications using the $\mathcal{Y}$ procedure. Details of the technique applied in the well-formedness checker are described in [26].

$$
\begin{array}{lll|lll}
\mathcal{T}(P(e_1,..,e_n)) & \Leftrightarrow & P(e_1,..,e_n) \wedge \bigwedge_{i=1}^{n} \delta(e_i) & \mathcal{F}(P(e_1,..,e_n)) & \Leftrightarrow & \neg P(e_1,..,e_n) \wedge \bigwedge_{i=1}^{n} \delta(e_i) \\
\mathcal{T}(\mathit{true}) & \Leftrightarrow & \mathit{true} & \mathcal{F}(\mathit{true}) & \Leftrightarrow & \mathit{false} \\
\mathcal{T}(\mathit{false}) & \Leftrightarrow & \mathit{false} & \mathcal{F}(\mathit{false}) & \Leftrightarrow & \mathit{true} \\
\mathcal{T}(\neg\phi) & \Leftrightarrow & \mathcal{F}(\phi) & \mathcal{F}(\neg\phi) & \Leftrightarrow & \mathcal{T}(\phi) \\
\mathcal{T}(\phi_1 \wedge \phi_2) & \Leftrightarrow & \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2) & \mathcal{F}(\phi_1 \wedge \phi_2) & \Leftrightarrow & \mathcal{F}(\phi_1) \vee \mathcal{F}(\phi_2) \\
\mathcal{T}(\phi_1 \vee \phi_2) & \Leftrightarrow & \mathcal{T}(\phi_1) \vee \mathcal{T}(\phi_2) & \mathcal{F}(\phi_1 \vee \phi_2) & \Leftrightarrow & \mathcal{F}(\phi_1) \wedge \mathcal{F}(\phi_2) \\
\mathcal{T}(\forall x.\ \phi) & \Leftrightarrow & \forall x.\ \mathcal{T}(\phi) & \mathcal{F}(\forall x.\ \phi) & \Leftrightarrow & \exists x.\ \mathcal{F}(\phi) \\
\mathcal{T}(\exists x.\ \phi) & \Leftrightarrow & \exists x.\ \mathcal{T}(\phi) & \mathcal{F}(\exists x.\ \phi) & \Leftrightarrow & \forall x.\ \mathcal{F}(\phi)
\end{array}
$$

Fig. 4. Derived equivalences for $\mathcal{T}$ and $\mathcal{F}$.

Additionally, in order to be able to compare the different procedures, we have built a prototype that implements the $\mathcal{D}$, $\mathcal{L}$, and $\mathcal{Y}$ procedures for the syntax given in Figure 1. We used the two automated theorem provers that are integrated with Spec#: Simplify [15] and Z3 [14], both of which are used by several other tools as prover back-ends too. The experiment was performed on a machine with Intel Pentium M (1.86 GHz) and 2 GB RAM.

The benchmark. We have used the following inductively defined formula, which allowed us to experiment with formula sizes that grow linearly with respect to n, and which is well-defined for every natural number n:

$$
\begin{array}{lcl}
\phi_0 & \triangleq & f(x) = x \ \vee \ f(-x) = -x \\
\phi_n & \triangleq & \phi_{n-1} \ \wedge \ (f(x+n) = x+n \ \vee \ f(-x-n) = -x-n)
\end{array}
$$

where the definition and domain restriction of f is as follows:

$$\forall x.\ x \geq 0 \ \Rightarrow \ f(x) = x \qquad \text{and} \qquad d_f\colon \ x \geq 0$$

Note that formula ϕ_n is well-defined for any n. However, its well-definedness cannot be proven using $\mathcal{L}$ for any n, and no re-ordering would help this situation.

Empirical results. Figure 5(a) shows that well-definedness conditions generated by $\mathcal{D}$ grow exponentially, whereas conditions generated by $\mathcal{L}$ and $\mathcal{Y}$ grow linearly. This was expected from their definitions. Note that the y axis uses a logarithmic scale. The figure also shows, that the sizes of conditions generated using $\mathcal{Y}$ are smaller than those generated by $\mathcal{L}$ for $n > 4$.

Figure 5(b) compares the time that Simplify (version 1.5.4) required to prove the well-definedness conditions generated from our input formula. As required by its interface, these conditions were given to Simplify as plain text. We see that the time required to prove formulas generated by $\mathcal{D}$ grows exponentially, whereas with $\mathcal{Y}$ the required time grows linearly. Note that the y axis uses a logarithmic scale. Additionally, for $\mathcal{D}$ our prototype was not able to generate the well-definedness condition for input formulas with $n > 16$ because it ran out of memory.

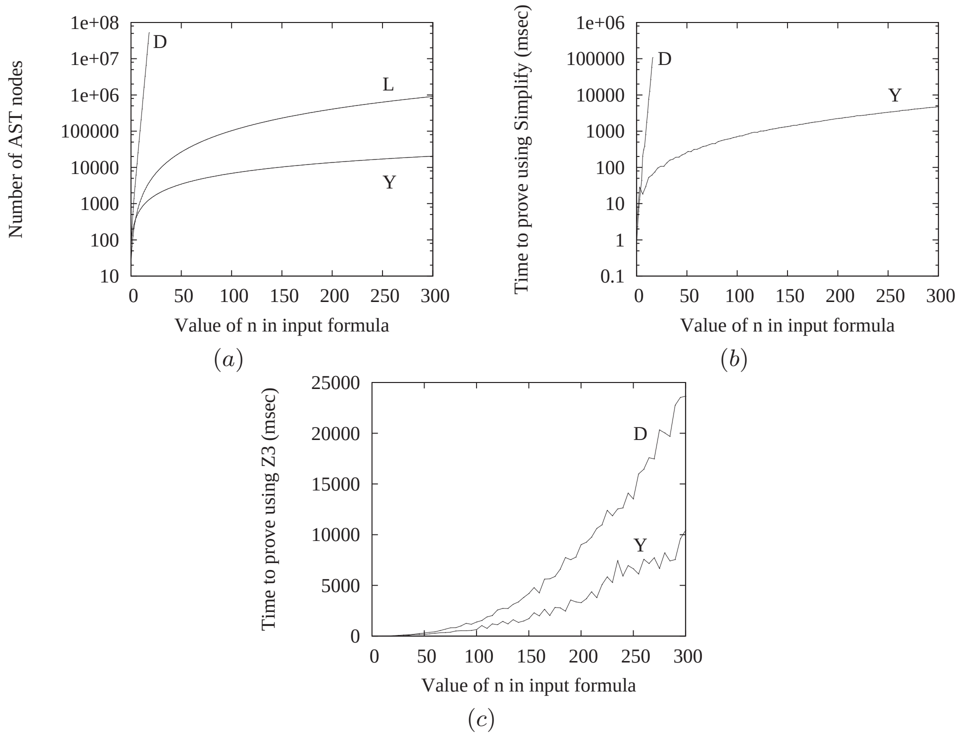

Fig. 5. (a) Size of well-definedness conditions generated by procedures $\mathcal{D}$, $\mathcal{L}$, and $\mathcal{Y}$; (b) Time to prove well-definedness conditions using Simplify; (c) Time to prove well-definedness conditions using Z3.

Figure 5(c) shows the results of the same experiment using Z3 (version 1.2). Note that the y axis is linear. From this graph we see that although the times required to prove well-definedness conditions show the same growth pattern for both procedure $\mathcal{D}$ and $\mathcal{Y}$, the times recorded for $\mathcal{Y}$ are approximately 1/3 to 1/2 below that of for $\mathcal{D}$. For instance, with $n = 200$, Z3 proves the condition generated by $\mathcal{D}$ in 9 seconds, while it takes 3.5 seconds for the condition generated by $\mathcal{Y}$. For $n = 300$, these figures are 23.5 and 10.5 seconds, respectively. Note that we could successfully prove much larger well-definedness conditions generated by $\mathcal{D}$ in Z3 as compared to Simplify. This is because (1) we used the native API of Z3 in order to construct formulas with maximal sharing, and (2) due to the use of its API, Z3 may have benefited from sub-formula sharing, which could have made the size of the resulting formula representation linear. In spite of this, procedure $\mathcal{Y}$ performs better than $\mathcal{D}$.

Note that Figure 5(b) and 5(c) do not plot the results of $\mathcal{L}$. This is because the $\mathcal{L}$ procedure cannot prove well-definedness of the input formulas.

Finally, we note that the whole sequence of formulas were passed to a single session of Simplify or Z3, respectively.

5 Related Work

The handling of partial functions in formal specifications has been studied extensively and several different approaches have been proposed. Here we only mention three mainstream approaches and refer the reader for detailed accounts to Arthan's paper [4], which classifies different approaches to undefinedness, to Abrial and Mussat's paper [3, Section 1.7], and to Hähnle's survey [18].

Eliminating undefinedness. As mentioned already in the paper, eliminating undefinedness by the generation of well-definedness conditions is a common technique to handle partial functions, and is applied in several approaches, such as B [8, 3], PVS [27], CVC Lite [9], and ESC/Java2 [11]. The two procedures proposed in these papers are $\mathcal{D}$ and $\mathcal{L}$.

PVS combines proving well-definedness conditions with type checking. In PVS, partial functions are modeled as total functions whose domain is a predicate subtype [27]. This makes the type system undecidable requiring Type Correctness Conditions to be proven. PVS uses the $\mathcal{L}$ operator because $\mathcal{D}$ was found to be inefficient [27].

CVC Lite uses the $\mathcal{D}$ procedure for the well-definedness checking of formulas. Berezin et al. [9] mention that if formulas are represented as DAGs, then the worst-case size of $\mathcal{D}(\phi)$ is linear with respect to the size of ϕ. However, there are no empirical results presented to confirm any advantages of using the DAG representation in terms of proving times.

Recent work on ESC/Java2 by Chalin [11] requires proving the well-definedness of specifications written in the Java Modeling Language (JML) [23]. Chalin uses the $\mathcal{L}$ procedure, however, as opposed to other approaches, not because of inefficiency issues. The $\mathcal{L}$ procedure directly captures the semantics of conditional boolean operators (e.g. `&&` and `||` in Java) that many programming languages contain, and which are often used, for instance, in JML specifications. Chalin's survey [10] indicates that the use of $\mathcal{L}$ is better suited for program verification than $\mathcal{D}$, since it yields well-definedness conditions that are closer to the expectations of programmers.

Schieder and Broy [28] propose a different approach to the checking of well-definedness of formulas. They define a formula under a three-valued interpretation to be well-defined if and only if its interpretation yields **true** both if $\perp$ is interpreted as **true**, and if $\perp$ is interpreted as **false**. Although checking well-definedness of formulas becomes relatively simple, the interpretation may be unintuitive for users. For example, formula $\perp \vee \neg\perp$ is considered to be well-defined. We prefer to eliminate such formulas by using classical Kleene logic.

Three-valued logics. Another standard way to handle partial functions is to fully integrate ill-defined terms into the formal logic by developing a three-valued logic. This approach is attributed to Kleene [22]. A well-known three-valued logic is LPF [7, 12] developed by C.B. Jones et al. in the context of VDM [20]. Other languages that follow this approach include Z [29] and OCL [1].

A well-known drawback of three-valued logics is that they may seem unnatural to proof engineers. For instance, in LPF, the law of the excluded middle and the deduction rule (a.k.a. ImpI) do not hold. Furthermore, a second notion of equality (called "weak equality") is required to avoid proving, for instance, that $x / 0 = \mathit{fact}(-5)$ holds. Another major drawback is that there is significantly less tool support for three-valued logics than there is for two-valued logics.

Underspecification. The approach of underspecification assigns an ill-defined term a definite, but unknown value from the type of the term [16]. Thus, the resulting interpretation is two-valued, however, in certain cases the truth value of formulas cannot be determined due to the unknown values. For instance, the truth value of $x / 0 = \mathit{fact}(-5)$ is known to be either **true** or **false**, but there is no way to deduce which of the two. However, for instance, $x / 0 = x / 0$ is trivially provable. This might not be a desired behavior. For instance, the survey by Chalin [10] argues that this is against the intuition of programmers, who would rather expect an error to occur in the above case. Underspecification is applied, for instance, in the Isabelle theorem prover [25], the Larch specification language [17], and JML [23].

6 Conclusion

A commonly applied technique to handle partial-function applications in formal specifications is to pose well-definedness conditions, which guarantee that undefined terms and formulas are never encountered. This technique allows one to use two-valued logic to reason about specifications that have a three-valued semantics. Previous work proposed two procedures, each having some drawback. The $\mathcal{D}$ procedure yields formulas that are too large to be used in practice. The $\mathcal{L}$ procedure is incomplete, resulting in the rejection of well-defined formulas.

In this paper we proposed a new procedure $\mathcal{Y}$, which eliminates these drawbacks: $\mathcal{Y}$ is complete and yields formulas that grow linearly with respect to the size of the input formula. Approaches that apply the $\mathcal{D}$ or $\mathcal{L}$ procedures (for instance, B, PVS, and CVC Lite) could benefit from our procedure. The required implementation overhead would be minimal.

Our procedure has been implemented in the Spec# verification tool to enforce well-formedness of invariants and method specifications. Additionally, we implemented a prototype to allow us to compare the new procedure with $\mathcal{D}$ and $\mathcal{L}$. Beyond the expected benefits of shorter well-definedness conditions, our experiments also show that theorem provers need less time to prove the conditions generated using $\mathcal{Y}$.

Acknowledgments. We would like to thank Peter Müller and the anonymous reviewers for their helpful comments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

1. UML 2.0 OCL Specification. Available at `http://www.omg.org/docs/formal/06-05-01.pdf`, May 2006.
2. J.-R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
3. J.-R. Abrial and L. Mussat. On using conditional definitions in formal theories. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002*, volume 2272 of *LNCS*, pages 242–269. Springer-Verlag, 2002.
4. R. Arthan. Undefinedness in Z: Issues for specification and proof. Presented at CADE Workshop on Mechanization of Partial Functions, 1996.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.
6. C. W. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker category B. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *LNCS*, pages 515–518. Springer-Verlag, 2004.
7. H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.
8. P. Behm, L. Burdy, and J.-M. Meynadier. Well Defined B. In D. Bert, editor, *International B Conference*, volume 1393 of *LNCS*, pages 29–45. Springer-Verlag, 1998.
9. S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in CVC Lite. In *Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2004.
10. P. Chalin. Are the logical foundations of verifying compiler prototypes matching user expectations? *Formal Aspects of Computing*, 19(2):139–158, 2007.
11. P. Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *ICSE*, pages 23–33. IEEE Computer Society, 2007.
12. J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In *Refinement Workshop*, pages 51–69, 1991.
13. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, 1995.
14. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.
15. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.
16. D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In *Computer Science Today*, volume 1000 of *LNCS*, pages 366–373. Springer-Verlag, 1995.
17. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.
18. R. Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IGPL*, 13(4):415–433, 2005.
19. A. Hoogewijs. On a formalization of the non-definedness notion. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 25:213–217, 1979.
20. C. B. Jones. *Systematic software development using VDM.* Prentice Hall, 1986.
21. J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005.

22. S. C. Kleene. On a notation for ordinal numbers. *Journal of Symbolic Logic*, 3:150–155, 1938.
23. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
24. J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.
25. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
26. A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In J. Cuellar and T. Maibaum, editors, *Formal Methods (FM)*, volume 5014 of *LNCS*, pages 68–83. Springer-Verlag, 2008.
27. J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.
28. B. Schieder and M. Broy. Adapting calculational logic to the undefined. *The Computer Journal*, 42(2):73–81, 1999.
29. J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.
30. G. Sutcliffe, C. B. Suttner, and T. Yemenis. The TPTP Problem Library. In A. Bundy, editor, *CADE*, volume 814 of *LNCS*, pages 252–266. Springer-Verlag, 1994.

A Semantic Proof of Equivalence

Structures. We define structures and interpretations in a way similar to as Behm et al. [8]. Let $\mathbf{A}$ be a set that does not contain $\perp$. We define $\mathbf{A}_\perp$ as $\mathbf{A} \cup \{\perp\}$. Let $\mathbf{F}$ be a set of function symbols, and $\mathbf{P}$ a set of predicate symbols. Let $\mathbf{I}$ be a mapping from $\mathbf{F}$ to the set of functions from $\mathbf{A}^n$ to $\mathbf{A}_\perp$, and from $\mathbf{P}$ to the set of predicates from $\mathbf{A}^n$ to $\{\mathbf{true}, \mathbf{false}\}$ (for simplicity, we assume that the interpretation of predicates is total), where n is the arity of the corresponding function or predicate symbol. We say that $\mathbf{M} = \langle \mathbf{A}, \mathbf{I} \rangle$ is a structure for our language with carrier set $\mathbf{A}$ and interpretation $\mathbf{I}$. We call a structure total if the interpretation of every function $\mathbf{f} \in \mathbf{F}$ is total, which means $\mathbf{f}(\ldots) \neq \perp$. We call the structure partial otherwise. A partial structure $\mathbf{M}$ can be extended to a total structure $\hat{\mathbf{M}}$ by having functions evaluated outside their domains return arbitrary values.

Interpretation. For a term $\mathbf{t}$, structure $\mathbf{M}$, and variable assignment $\mathbf{e}$, we denote the interpretation of $\mathbf{t}$ as $[\mathbf{t}]^{\mathbf{e}}_{\mathbf{M}}$. *Variable assignment* $\mathbf{e}$ maps the free variables of $\mathbf{t}$ to values. We define the interpretation of terms as given in Figure 6. Interpretation of formula φ denoted as $[\varphi]^{\mathbf{e}}_{\mathbf{M}}$ is given in Figure 7. **Dom** yields the domain of the interpretation of function symbols.
To check whether or not a value $\mathbf{l}$ is defined, we use function $\mathbf{wd}$:

$$\mathbf{wd}(\mathbf{l}) = \begin{cases} \mathbf{true}, & \text{if } \mathbf{l} \in \{\mathbf{true}, \mathbf{false}\} \\ \mathbf{false}, & \text{if } \mathbf{l} = \perp \end{cases}$$

$$
\begin{array}{lcl}
[\,\mathbf{v}\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \mathbf{e}(\mathbf{v}) \text{ where } \mathbf{v} \text{ is a variable} \\
[\,\mathbf{f}(\mathbf{t}_1,\ldots,\mathbf{t}_n)\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{I}(\mathbf{f})([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}), \text{ if } \langle[\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}\rangle \in \mathbf{Dom}(\mathbf{I}(\mathbf{f})) \\ \qquad\qquad\qquad\qquad \text{and } [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot,\ldots,\ [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \\ \bot, \text{otherwise} \end{cases}
\end{array}
$$

Fig. 6. Interpretation of terms.

Lemma 1. *For every total structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*, we have* $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$.
Proof. By induction over the structure of φ. □

Lemma 2. *For every structure* $\mathbf{M}$*, if* $\mathbf{M}$ *is extended to total structure* $\hat{\mathbf{M}}$*, then* $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\hat{\mathbf{M}}}) = \mathbf{true}$.
Proof. Trivial consequence of the way $\mathbf{M}$ is extended and of **Lemma 1**. □

Domain restrictions. Each function $\mathbf{f}$ is associated with a domain restriction d_f, which is a predicate that represents the domain of function $\mathbf{f}$. A structure $\mathbf{M}$ is a model for domain restrictions of functions in $\mathbf{F}$ (denoted by $d_{\mathbf{F}}(\mathbf{M})$) if and only if:

- The domain formulas are defined. That is, for each $\mathbf{f} \in \mathbf{F}$ and for all $\mathbf{e}$:
  $\mathbf{wd}([d_f]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$
- Domain restrictions characterize the domains of function interpretations for $\mathbf{M}$. That is, for each $\mathbf{f} \in \mathbf{F}$ and $\mathbf{l}_1,\ldots,\mathbf{l}_n \in \mathbf{A}$:
  $[d_f]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if $\langle \mathbf{l}_1,\ldots,\mathbf{l}_n\rangle \in \mathbf{Dom}(\mathbf{I}(f))$
  where $e = [v_1 \rightarrow \mathbf{l}_1,\ldots,v_n \rightarrow \mathbf{l}_n]$ and $\{v_1,\ldots,v_k\}$ are $\mathbf{f}$'s parameter names.

In the following we prove two lemmas and finally our two main theorems.

Lemma 3. *For each structure* $\mathbf{M}$*, term* $\mathbf{t}$*, and variable assignment* $\mathbf{e}$*:*
if $d_{\mathbf{F}}(\mathbf{M})$ *then* $[\mathbf{t}]^{\mathbf{e}}_{\mathbf{M}} \neq \bot$ *if and only if* $[\delta(\mathbf{t})]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$.
Proof. By induction on the structure of $\mathbf{t}$ under the assumption that $d_{\mathbf{F}}(\mathbf{M})$.
Induction base: $\mathbf{t}$ is variable $\mathbf{v}$.
Since $[\mathbf{v}]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{e}(\mathbf{v}) \neq \bot$ and $[\delta(\mathbf{v})]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$, we have the desired property.
Induction step: $\mathbf{t}$ is function application $\mathbf{f}(\mathbf{t}_1,\ldots,\mathbf{t}_n)$.
From definition of interpretation we get that $[\mathbf{f}(\mathbf{t}_1,\ldots,\mathbf{t}_n)]^{\mathbf{e}}_{\mathbf{M}} \neq \bot$ if and only if:

$$\langle[\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}\rangle \in \mathbf{Dom}(\mathbf{I}(\mathbf{f})) \;\wedge\; [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \wedge \ldots \wedge [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot$$

By the definition of $d_{\mathbf{F}}(\mathbf{M})$ and the induction hypothesis, it is equivalent to:

$$[d_f(\mathbf{t}_1,\ldots,\mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge [\delta(\mathbf{t}_1)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge \ldots \wedge [\delta(\mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$$

$$
\begin{array}{lll}
[\,\mathit{true}\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \mathbf{true} \\
[\,\mathit{false}\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \mathbf{false} \\
[\,P(\mathbf{t}_1,\ldots,\mathbf{t}_n)\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } \mathbf{I}(P)([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true} \text{ and} \\ & \quad [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot,\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \\ \mathbf{false}, & \text{if } \mathbf{I}(P)([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{false} \text{ and} \\ & \quad [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot,\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \\ \bot, & \text{otherwise} \end{cases} \\
[\,\neg\varphi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \\ \mathbf{false}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\varphi \wedge \phi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \text{ and } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \text{ or } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\varphi \vee \phi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \text{ or } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if } [\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \text{ and } [\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\forall\, x.\ \varphi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if for all } \mathbf{l} \in \mathbf{A},\ [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if there exists } \mathbf{l} \in \mathbf{A} \text{ such that} [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\exists\, x.\ \varphi\,]^{\mathbf{e}}_{\mathbf{M}} & \triangleq & \begin{cases} \mathbf{true}, & \text{if there exists } \mathbf{l} \in \mathbf{A} \text{ such that} [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{true} \\ \mathbf{false}, & \text{if for all } \mathbf{l} \in \mathbf{A},\ [\varphi]^{\mathbf{e}[x\leftarrow\mathbf{l}]}_{\mathbf{M}} = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 7. Interpretation of formulas.

which is, by the definition of δ, equivalent to $[\delta(\mathbf{f}(\mathbf{t}_1,\ldots,\mathbf{t}_n))]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. $\square$

Lemma 4. *For each structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*:*
if $d_{\mathbf{F}}(\mathbf{M})$ *then* $[\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ *if and only if* $[\mathcal{T}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ *and*
$[\varphi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false}$ *if and only if* $[\mathcal{F}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$*.*

Proof. By induction on the structure of φ under the assumption that $d_{\mathbf{F}}(\mathbf{M})$.
Induction base: φ is predicate $P(\mathbf{t}_1,\ldots,\mathbf{t}_n)$.
From definition of interpretation we get $[P(\mathbf{t}_1,\ldots,\mathbf{t}_n)]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if:

$$\mathbf{I}(P)([\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}},\ldots,[\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true} \ \wedge\ [\mathbf{t}_1]^{\mathbf{e}}_{\mathbf{M}} \neq \bot \wedge \ldots \wedge [\mathbf{t}_n]^{\mathbf{e}}_{\mathbf{M}} \neq \bot$$

which is, by the assumption that the interpretation of predicates is total and by **Lemma 3**, equivalent to:

$$[P(\mathbf{t}_1,\ldots,\mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge [\delta(\mathbf{t}_1)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true} \wedge \ldots \wedge [\delta(\mathbf{t}_n)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$$

which is, by the definition of $\mathcal{T}$, equivalent to $[\mathcal{T}(P(\mathbf{t}_1, \ldots, \mathbf{t}_n))]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$.
The proof is analogous for $\mathcal{F}$.
Induction step: For brevity, we only present the proof of those two cases for which the syntactic derivation was shown on page 7. The proofs are analogous for all other cases.

1. We prove that if $d_{\mathbf{F}}(\mathbf{M})$ then $[\gamma \wedge \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if $[\mathcal{T}(\gamma \wedge \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$.
From definition of interpretation we get that $[\gamma \wedge \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ if and only if $[\gamma]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$ and $[\phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$, which is, by the induction hypothesis, equivalent to $[\mathcal{T}(\gamma)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ and $[\mathcal{T}(\phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$, which is, by the definition of $\mathcal{T}$, equivalent to $[\mathcal{T}(\gamma \wedge \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$.

2. We prove that if $d_{\mathbf{F}}(\mathbf{M})$ then $[\forall x.\ \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false}$ iff $[\mathcal{F}(\forall x.\ \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$.
From the definition of the interpretation function we get that $[\forall x.\ \phi]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{false}$ if and only if there exists $\mathbf{l} \in \mathbf{A}$ such that $[\phi]^{\mathbf{e}[x \leftarrow \mathbf{l}]}_{\mathbf{M}} = \mathbf{false}$. By the induction hypothesis, this is equivalent to the existence of $\mathbf{l} \in \mathbf{A}$ such that $[\mathcal{F}(\phi)]^{\mathbf{e}[x \leftarrow \mathbf{l}]}_{\hat{\mathbf{M}}} = \mathbf{true}$, which is, by the definition of $\mathcal{F}$, equivalent to $[\mathcal{F}(\forall x.\ \phi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. □

Theorem 2. *For each structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*:*
if $d_{\mathbf{F}}(\mathbf{M})$ *then* $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = [\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}}$
Proof. From the definition of **wd** we know that $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}})$ is defined. Furthermore, from **Lemma 2** (with φ substituted by $\mathcal{Y}(\varphi)$) we know that $[\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}}$ is defined. Thus, it is enough to prove that $\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$ if and only if $[\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$. Under the assumption that $d_{\mathbf{F}}(\mathbf{M})$, we have:
$\mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = \mathbf{true}$ if and only if [by definition of **wd**]
$[\varphi]^{\mathbf{e}}_{\mathbf{M}} \in \{\mathbf{true}, \mathbf{false}\}$ if and only if [by **Lemma 4**]
$[\mathcal{T}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ or $[\mathcal{F}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ if and only if [by definition of $\mathcal{Y}$]
$[\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} = \mathbf{true}$ □

Berezin et al. [6] proved the following characteristic property of $\mathcal{D}$:

$$\text{if } d_{\mathbf{F}}(\mathbf{M}) \text{ then } \mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}}) = [\mathcal{D}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}} \tag{2}$$

Theorem 3. *For each total structure* $\mathbf{M}$*, formula* φ*, and variable assignment* $\mathbf{e}$*:*
$[\mathcal{D}(\varphi) \Leftrightarrow \mathcal{Y}(\varphi)]^{\mathbf{e}}_{\mathbf{M}} = \mathbf{true}$
Proof. For each total structure $\mathbf{M}$ there exists a partial structure $\mathbf{M}'$ such that $\mathbf{M} = \hat{\mathbf{M}}'$ and $d_{\mathbf{F}}(\mathbf{M}')$. We can build $\mathbf{M}'$ from $\mathbf{M}$ by restricting the domain of partial functions according to the domain restrictions.
By **Theorem 2** and (2) we get $[\mathcal{D}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}'} = \mathbf{wd}([\varphi]^{\mathbf{e}}_{\mathbf{M}'}) = [\mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}'}$. Which is equivalent to $[\mathcal{D}(\varphi) \Leftrightarrow \mathcal{Y}(\varphi)]^{\mathbf{e}}_{\hat{\mathbf{M}}'} = \mathbf{true}$. Since $\mathbf{M} = \hat{\mathbf{M}}'$ we get the desired property. □