Implementing Baker’s SUBTYPEP decision procedure

Léo Valais
Jim E. Newton
Didier Verna
lvalais@lrde.epita.fr
jnewton@lrde.epita.fr
didier@lrde.epita.fr
EPITA/LRDE
Le Kremlin-Bicêtre, France

ABSTRACT

We present here our partial implementation of Baker’s decision procedure for Common Lisp’s subtypep. In his article “A Decision Procedure for Common Lisp’s SUBTYPEP Predicate”, he claims to provide implementation guidelines to obtain a subtypep more accurate and as efficient as the average implementation. However, he did not provide any serious implementation and his description is sometimes obscure. In this paper we present our implementation of part of his procedure, only supporting primitive types, Clos classes, member, range and logical type specifiers. We explain in our words our understanding of his procedure, with much more detail and examples than in Baker’s article. We therefore clarify many parts of his description and fill in some of its gaps or omissions. We also argue in favor and against some of his choices and present our alternative solutions. We further provide some proofs that might be missing in his article and some early efficiency results. We have not released any code yet but we plan to open source it as soon as it is presentable.

CCS CONCEPTS

• Theory of computation → Type theory; Divide and conquer; Pattern matching.

ACM Reference Format:

1 INTRODUCTION

The Common Lisp standard [1] provides the predicate function subtypep for introspecting the sub-typing relationship. Every invocation (subtypep A B) either returns the values (t t) when A is a subtype of B, (nil t) when not, or (nil nil) meaning the predicate could not (or failed to) answer the question. The latter can happen when the type specifier (satisfies P) (representing the type (x | P(x)) for some predicate and total function P) is involved. For example, given two arbitrary predicates F and G, there is no way subtypep can answer the question (subtypep ‘(satisfies F) ’(satisfies G)).

However, some implementations abuse the permission to return (nil nil). For example, in Scl 1.4.10 (the implementation we are currently focusing our efforts on), (subtypep ‘boolean ‘keyword) returns (nil nil), thus violating the standard. The definition of the keyword type is responsible for this failure: as shown in Listing 1, it involves a satisfies type specifier.

Another kind of problem for which subtypep’s accuracy matters is the optimization of the typeof cast construct as shown in [7] and [8]. The aim is to remove redundant checks in the construct and the approach is to use binary decision diagrams. However, to build such a structure, subtypep is repeatedly used. The unreliability of the predicate leads here to many lost BDD reductions and therefore to the generation of sub-optimal code.

Our implementation is still in active development, currently targets Scl and focuses almost entirely on result accuracy. It supports primitive types, user-defined types (deftype, classes and structures), member (and eql) type specifiers and ranges (e.g., (integer * 12)). We present our strategy for implementing each one of these while discussing how and why we decided or not to diverge from Baker’s approach—or potentially filling some gaps or unclear bits. No optimization work has been done yet and the implementation still has bugs and diverge issues, but we have found some encouraging results about accuracy and even about efficiency.

2 THE COMMON LISP TYPE SYSTEM

2.1 Type specifiers

Common Lisp types are not manipulated directly. Instead, the type to be manipulated is described using a type specifier. The type specifier Domain-Specific Language (DSL) allows programmers to describe types by writing S-expressions which obey some rules described in the Common Lisp standard [1].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ELS’19, April 01–02 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.2646982

\begin{small}
\begin{lstlisting}
(defun deftype keyword ()
  `(and symbol (satisfies keyword)))
\end{lstlisting}
\end{small}

Listing 1: The keyword type definition in Scl

1 A function defined over its entire definition domain.
2 The Common Lisp standard requires that no invocation of subtypep involving only primitive types return (nil nil).
3 C.F. bug #1533685 in Scl bug tracker.
while (or symbol (integer * *))

kingdom

In Baker’s terminology, a “type kingdom” designates the types that can be described using only one kind of type specifier. nil (the empty type) belongs to every type kingdom.

It is possible to define parametric aliases using the deftype construct. It is then possible to refer to a whole type specifier using its alias. Listing 2 shows an example of parametric deftype.

2.2 Vocabulary

type canonical t.s. A set of elements. For any type $u: u \equiv \{x \mid x : u\}$

primitive type A type specifier without aliases.

symbolic form A type specifier whose type is symbol.

compound form A compound form whose car is or, and or not.

logical form In Baker’s terminology, a “type kingdom” designates the types that can be described using only one kind of type specifier. nil (the empty type) belongs to every type kingdom.

In this article we focus on two particular type kingdoms:

- the literal type kingdom, represented using only symbolic, member and logical type specifiers, and,
- the range type kingdom, represented only using range and logical type specifiers

For example, (or string symbol) belongs to the literal type kingdom. (and number (not real)) belongs to the range type kingdom. However, (or symbol integer) belongs to the literal type kingdom while (or symbol (integer * *)) belongs to both. This situation is handled in section 4.

There are other type kingdoms that Baker mentions in his article, such as the array type kingdom, represented using only array and logical type specifiers. Note that a type can belong to several kingdoms, as multiple type specifiers can describe it. For example, integer belongs to literal and range kingdoms as the type specifiers integer (symbolic) and (integer * *) (range) both describe it. In Section 4, we describe how to guarantee that a given type is only described by one kind of type specifier, hence restricting it to one kingdom.

3 PROCEDURE’S MECHANISMS OVERVIEW

Figure 1 shows the internals of our implementation. Every step will be detailed in the following sections. There are three major stages:

1. The pre-processing — Both type specifiers are processed in order to simplify further calculations: the aliases are expanded, and each occurrence of numeric types are converted to their equivalent range type specifier. Finally, as explained thereafter, the procedure splits into several sub-procedures, one for each type kingdom, because their internal type representation differ. In order to achieve that, the type specifiers must also be split into equivalent subtype specifiers restricted to each concerned kingdom. This stage is detailed in Section 4.

2. Expert sub-procedures — Once split, each subtype specifier is redirected to the appropriate expert sub-procedure. The job of such a procedure is to prove, in its own kingdom, the assertion “A is a subtype of B” to be wrong. Our procedures currently only support literal and range type specifiers—an expert sub-procedure has been implemented only for these two kingdoms. This stage is detailed in Section 5.

3. Result conjunction — Eventually, all expert sub-procedures return (a Boolean) and the results are accumulated using conjunction. (In practice, as soon as one expert procedure returns false, subtypep returns.)

4 PRE-PROCESSING

4.1 Alias expansion

The very first step is to ensure that the type specifier is in its canonical form, that is, having all its aliases expanded. This is done by the expand function. For example, considering the type created in Listing 2, (expand '(except 12)) should return (not (eql 12)).

Unlike macro expansion, deftype expansion is not standardized in Common Lisp. Thus a solution must be found for each Common Lisp implementation independently. As our efforts are currently focused on SACL, we discuss how we implement the expand function for that compiler.

SACL’s subtypep heavily relies on the function sb-kernel:specifier-type, which does type expansion. It also does type simplification—turning (and integer string) into nil—which could have saved us some work. We hoped we could simplify that function to make it compatible with Baker’s algorithm while keeping the deftype expansion and the range canonicalization work. However we found, thanks to [7] tools, that the function is responsible for most of the work of subtypep, as shown in Figure 2 Considering the lack of efficiency of that function and the fact that it would not be trivial to simplify it to only keep the interesting bits, we decided on another, more cost-effective solution.

The function sb-ext:typexpand takes a type specifier and tries to expand it (not recursively). It either returns the expansion result, or the input type specifier if it is not expandable. (sb-ext:typexpand 'integer) returns integer since it is not a deftype alias whereas (sb-ext:typexpand 'except) returns (not (eql 12)). To expand a whole type specifier, it just needs to walk through it, applying sb-ext:typexpand on each list or atom manually. One subtlety though is that the result of an expansion may itself be an alias to expand\(^4\). For example, let’s say that we have (deftype my-type () '(except 0.0)), then the result of (sb-ext:typexpand 'my-type) is (except 0 0), which is of course an alias to expand again.

\(^4\)Fortunately, sb-ext:typexpand also returns a Boolean indicating whether or not an expansion happened.
4.2 Numeric type specifiers conversion

As explained in Section 3, after pre-processing both type specifiers, the procedure splits in two expert sub-procedures: one for literal type specifiers and one for range type specifiers. Numeric types—types containing numbers (mathematically speaking)—can have different representations: a symbol (e.g., fixnum), a member expression (e.g., (member 1 2 3)) or a range (e.g., (integer 1 6)). However, the first two belong to the literal type kingdom whereas the latter belongs to the range kingdom. Thus, the numerical type information would be distributed over the different expert sub-procedures. For consistency and accuracy, a single internal representation has to be chosen. The symbolic and member numeric types must each be converted into an equivalent type specifier, in which numerical data are only represented using ranges.

- **Symbolic** numeric type specifier — say U, replace it by (U * *). Note the new “type specifier” is likely not to be valid (e.g., (fixnum * *)) is invalid). Because it is never exposed to the user—as it is an intermediate, internal representation—nothing bad can happen. However, it cannot be used with other functions requiring a type specifier, such as typep.
- **member** type specifiers — e.g., (member a 1 2 :b) is converted to (or (member a :b) (bit 1 1) (integer 2 2)). To do that,
  (1) extract the numbers out of the expression,
  (2) map each number, say n, to construct the type specifier ((type-of n) n n)
  (3) and combine the remaining member expression and the ranges with the or logical type specifier.

A subtlety to consider is that super-types of number also contain numerical data that must be extracted. Indeed, the type atom contains both numerical data—(number * *)—and non-numerical data—(and atom (not (number * *))). Thus, its replacement in the numeric type kingdom is straightforward: (number * *). In the literal type kingdom however, its replacement is (or stream array character function standard-object symbol structure-object structure-class). The type t—which is (or stream sequence)—must be converted similarly.

Yet another subtlety is that the type specifiers (and) and (or) respectively describe the types t and nil. Hence every occurrence of (and) must be replaced by the replacement of t described in the previous paragraph. In order to remove that annoying corner case completely, (or) is also replaced, by nil.

4.3 Splitting

Having reached this step, the input now only contains canonical literal and range type specifiers, numeric types being only expressed as ranges. The next stage—expert sub-procedures—requires literal and numeric types to be separated.

Thus the top type t is divided into two disjoint subtypes—“kingdoms” as Baker says. The previous step, described in Section 4.2, ensures that the representation (in terms of type specifiers) of the types in each kingdom is different. All numeric types are

---

**Figure 1: Internal flowchart of (subtypep A B)**

**Figure 2: specifier-type weight in cl:subtypep execution**

^a cached-subtypep-caching-call is just a memoizing wrapper around SACL’s subtypep which is a bit more efficient than the raw implementation.
represented as ranges, and literal types as symbolic and member
(without numbers) type specifiers.
This step roughly consists of an in-depth traversal of the type
specifier, using pattern-matching to recognize which type specifier
represents which type. We use the implementation of [9] because
of its simplicity and versatility.

Our implementation uses a function type-keep-if which takes a
predicate \(P\) and a type specifier \(U\) and returns:

- \(U\) as it is when \(P(U) = \top\),
- \(\text{nil}\) when \(P(U) = \bot\),
- \((\text{op } U_1 \cdots U_n)\) where \(U_i = (\text{type-keep-if } P U_i)\) when
  \(U = (\text{op } U_1 \cdots U_n)\) and \text{op} \in \{\text{and, or, not}\}.

Given the predicate literal-type-p and a type \(U\), type-keep-if
returns \(U\) with every inner type specifier that describes a non-
literal type replaced by \(\text{nil}\) (interpreted as the empty type). The
result is then a subtype of \((\text{and } (\text{not } \text{number}) (\text{not } (\text{array } \ast *))\))
Likewise, given the predicate range-type-p, this function returns
\(U\) with every non-range inner type specifier replaced by \(\text{nil}\)
(interpreted this time as the empty range). Thus, the result is a subtype
of number. Therefore, split can easily be implemented in terms of
type-keep-if.

4.4 Type reformulation
For any types \(U\) and \(V\), \(U \subseteq V \iff U \cap \lnot V = \emptyset\). Therefore, for any
type specifiers \(U\) and \(V\), when \(\text{subtypep } U \lor V\) returns \(\top\), then
(\text{subtypep } `(\text{and } U (\text{not } V)) \text{nil}) also returns \(\top\).

The results of the split function are zipped together using
\((\text{lambda } (x y) `(\text{and }, x \text{ (not } y)))\) before being passed to the
expert sub-procedures. This way, they will not have to prove that
an arbitrary type is a subtype of another arbitrary subtype, but
rather whether one arbitrary type specifier describes the empty
type (which is substantially easier to reason about, and implement).

5 EXPERT SUB-PROCEDURES
Listing 3 shows how subtypep could be defined from a top-
down point of view. It shows that, according to Figure 1, both
type specifiers are processed independently, split into two king-
doms (literal and numeric types) and unified in an \((\text{and } \text{or } \text{not})\)
fashion. The expert sub-procedures, null-literal-type-p and
null-numeric-type-p, each accept one argument—a type specifier,
say \(U\)—and returns a Boolean indicating whether \(U\) describes
the empty type (\(\text{nil}\)).

Each sub-procedure answers restricted to its kingdom—as no
type can (at this point of the procedure) belong to two different
kingdoms, as shown in section 4. With that piece of information,
we can (now) safely assert that:

- the literal type kingdom is the type described by \((\text{and } \text{not number}) \text{ (not (array } \ast *))\)\(^5\), and,
- the numeric type kingdom is the type described by number\(^6\).

5.1 Procedure for literal types
5.1.1 Theory. To represent types in the literal types kingdom, we
suppose at first that there is a way to enumerate every element in
t, say \(e_1, e_2, \ldots, e_{\omega}\). Then, let \(u_1, u_2, \ldots, u_{\omega}\) be all the (non-strict)
subtypes of the top-level type \(t\). We associate to each pair \((u_i, e_j)\)
the bit \(b_{ij}\) with the value \(1\) when \(e_j \in u_i\) and \(0\) when \(e_j \notin u_i\). Let
\(b_{\omega\omega}\) be the representative bit-vector associated to the type \(u_i\), defined
by \([b_{1\omega}, b_{2\omega}, \ldots, b_{i\omega}]\). These bit-vectors are the rows of the
infinite matrix on Eq. \(B_{\omega\omega}\) which illustrates the system.

\[
\begin{pmatrix}
e_1 & e_2 & e_3 & \cdots & e_{\omega} \\
u_1 & 1 & 0 & 0 & 0 & \cdots & 1 \\
u_2 & 0 & 1 & 1 & 0 & \cdots & 0 \\
u_3 & 0 & 0 & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
u_{\omega} & 1 & 0 & 1 & 0 & \cdots & 0
\end{pmatrix}
\]

\((B_{\omega\omega})\)

\[\text{Proof. } \text{Each type has a unique bit-vector representation.}\]

Let \(u_i\) and \(u_j\) be two distinct types. Thus, \((u_i \cup u_j)(u_i \cap u_j) \neq \emptyset\).
Let \(e_k \in (u_i \cup u_j)(u_i \cap u_j)\). By definition, we have \(b_{ik} \neq b_{jk}\). Hence
\(b_{ij} \neq b_{\omega\omega}\). Two distinct types are represented by two different bit-vectors.

Similarly, let \(b_{ij}\) and \(b_{ij}\) be two different bit-vectors. Then
it necessarily exists a \(k\) such as \(b_{ik} \neq b_{jk}\). Therefore
\(\exists e_k, (e_k \neq u_i \vee e_k \notin u_j) \wedge e_k \notin u_i \cap u_j\). Hence \(u_i \neq u_j\). □

\(^5\)Actually this is not completely accurate since the type string can be described using
array type specifiers. However, since the latter are not supported by our implementation
yet, we consider the types string and bit-vector as being literal types since their
symbolic representation is kept through the entire process. This is very likely to
change in the future.

\(^6\)Our implementation does not support complex numbers yet, and considers the complex
type as being empty. Some wrong results arise from that supposition—such as (subtypep
null-numeric-type-p (number-type-p)) returning true. This will change as soon as complex numbers are supported.
To register a new type bit-vector:

1. Let \( u_k = u_1 \cup u_j \). By definition, \( \forall i \in \mathbb{N} \cup \{0\}, b_{ki} = 1 \) if \( b_{ki} = 1 \) or \( b_{ji} = 1 \), that is \( b_{ki} = b_{ki} \lor b_{ji} \). Thus, also by definition:

\[
    b_{vk} = [b_{k0}, b_{k1}, \ldots, b_{kn}]
\]

\[
    = [b_{10} \lor b_{j0}, b_{11} \lor b_{j1}, \ldots, b_{i0} \lor b_{j0}]
\]

\[
    = b_{v0} \lor b_{vj}
\]

2. We proceed similarly for the intersection and the Boolean logical operator “and” (\( \land \)).

3. Let \( u_k = \Pi_i \). We have by definition \( \forall i \in \mathbb{N} \cup \{0\}, b_{ki} = \neg b_{ji} \).

Then:

\[
    b_{vk} = [\neg b_{i0}, \neg b_{i1}, \ldots, \neg b_{in}]
\]

\[
    = \neg b_{vj}
\]

5.1.2 Implementation. Common Lisp cannot enumerate all the possible subtypes of \( t \) nor all of its elements. Fortunately, we do not need them all. We only need to consider the types mentioned in the input type specifier to determine its emptiness.

We also do not need to enumerate all the elements of these types. It is that aspect of the procedure of Baker that makes it both powerful and difficult to understand at first. We only need sufficiently many elements from a type to distinguish it from the other types. Because we are now considering only a finite number of types, say \( u_1, \ldots, u_n \), to register a new type \( u_{n+1} \) to our (now finite) matrix, we only need to find an element \( e \in u_{n+1} \) such as \( e \notin u_1 \cup \cdots \cup u_n \).

Now let’s suppose that the type specifier of \( u_{n+1} \) is in fact (member \( e \)), that \( e \) is itself chosen as a representative element for another type, say \( u_k \), and that \( u_k \) is only distinguished from the other registered types by that element \( e \). To register \( u_{n+1} \) and \( u_k \), we would then have the same bit-vector representation when these types are likely to be distinct. The general solution for that kind of problem is to register all the elements found inside the member type specifier. When there is a conflicting element \( e \) already registered as a representative for another types, we generate additional representatives for these types. That precaution ensures that this kind of conflict never happens and greatly simplifies the implementation of member type specifiers.

To implement that implementation matrix system, we use two functions: \( B : \text{type name} \mapsto \text{bit-vector}, \) with \( B(u_i) = b_{vi} \), and \( I : \text{representative} \mapsto \text{bit index}, \) with \( I(e_i) = i - 1 \). Baker suggests in his small example [3] using the operator set which is deprecated in modern Common Lisp programming. Instead, we use hash tables to represent these functions. Type names are symbols, bit-vectors are bit-vectors and element indexes are positive integers.

To register a new type \( u_{n+1} \), it is added to the \( B \) hash table and its bit-vector content \( b_{(n+1)i} \) is evaluated for all the existing representatives \( (i \in [1, m]) \). To register a new representative \( e_{n+1} \), it is added to the \( I \) hash table with the index \( m \). Then we add one bit (the \( m \)-th bit) to each bit-vector \( b_{vj} \) and evaluate it in respect to the type \( u_j \). Thus, to retrieve the bit-vector of a registered primitive or user-defined type \( t \), we just lookup its value \( B(t) \). To compute the bit-vector of a member expression \( \text{member} \ e_1 \cdots e_n \), we use the value \( B(\text{member} \ e_1 \cdots e_n) = \bigwedge_{i=1}^n \beta(I(e_i)) \), where \( \beta(x) \) returns the null bit-vector with the \( x \)-th bit activated.

The bit-vector of logical type specifiers are given in Eq. 1, Eq. 2 and Eq. 3.

\[
    B((\text{and} \ U_1 \cdots U_n)) = \bigwedge_{i=1}^n B(U_i)
\]

\[
    B((\text{or} \ U_1 \cdots U_n)) = \bigvee_{i=1}^n B(U_i)
\]

\[
    B((\text{not} \ U)) = \neg B(U)
\]

5.1.3 Issues. The method for choosing the representative elements for a type depends of its nature: it can be a primitive type, a user-defined type (class, structure or condition) or a member expression.

Since primitive types are known (c.f. table 4.2 of [2]), their representative elements are chosen at compile-time. The \( u_{n+1} \) sublety above should still be kept in mind. For instance, the type null is a subtype of both symbol and list; so three representative elements are needed: null, a non-empty list and a symbol other than null. Note that some primitive types are an exhaustive partition of other types (e.g., character \( \equiv \) (or base-char extended-char)). Obviously, in that case, such a precaution does not apply.

For user-defined types, Baker suggests to extend the type creation mechanism—thus modifying the implementation’s internal functions—to register a dummy element as a representative. We decided not to follow his approach because of the poor portability of his solution. Indeed, this work, often non-trivial, would have to be repeated for each targeted Common Lisp implementation. (We would like to avoid modifying the Sclt internal mechanisms.) Moreover, it would register a representative for every class created, thus increasing bit-vectors’ size uselessly since only a few of these classes are likely to appear in a subtypep type specifier. But more importantly, the main drawback of his solution is that creating that dummy element might have unexpected side-effects, as it may need to use slot’s default values and/or initialize-instance. We decided instead to use the Meta Object Protocol (Mop) [6], more specifically class prototypes. Class prototypes are pseudo-instances of a class, created without executing initialize-instance and which typep and eq view as traditional instances. However, to create a class prototype, the class needs to be finalized and it cannot be guaranteed until it is instantiated. Since that class may be involved in a subtypep call before that happens, when a new class is encountered, we force its finalization using the function ensure-finalized from the (portable) closer-mop package10. Then, we create the prototype of the class using sb-mop:class-prototype and register it. This method is much more portable than Baker’s and does not require to hook inside the implementation.

Since (in Sclt11) conditions are classes, they are supported automatically. The Common Lisp standard [1] states that “defstruct without a :type option defines a class with the structure name as its name”, hence in that case no additional work is required. The standard also states that “Specifying this option [...] prevents the

10http://common-lisp.net/project/closer/
11Every major lisp implementations implement conditions as Clos classes—the most obvious way to do it. We ignore exotic condition implementations.
structure name from becoming a valid type specifier recognizable by typep. Thus, subtypep is not concerned by these types of structures.

To address the misrepresentation problem when member type specifiers are involved, as discussed in Section 5.1.3, we must ensure that a new representative element is generated and registered. The Common Lisp standard ([1]) states that the member type specifier is defined in terms of eql. That is, \((\text{typep} \ e \ '(\text{member} \ e_1 \ \ldots \ e_n))\) uses eql to compare \(e\) to the successive \(e_k\) to check the membership. That precise property reduces the misrepresentation problem to only two types: symbol and character (and their subtypes).

To better understand why it is the case, first consider a reduced version of the top-level type \(t = (\text{or string list symbol})\). Then, let \(R = (\text{"hello" (1 2 3) foo})\) be our list of representatives.

1. Let’s ask the question \((\text{subtypep} \ ('\text{symbol} \ (\text{member} \ \text{foo})))\).
2. As discussed in Section 5.1.3, we add the elements of the member expression to \(R\). To conform with the specification, we first check whether or not \(\text{foo}\) is already in \(R\) eql-wise: \(\text{foo} \equiv\text{eql} \ R\), so \(R\) does not change.
3. As shown in Eq. 4, the emptiness check passes, meaning that \(\text{symbol}\) is indeed a subtype of \((\text{member} \ \text{foo})\), which is obviously wrong.

\[
B(\text{symbol}) \land \neg B((\text{member} \ \text{foo})) = 001 \land \neg001 \tag{4}
\]

However, for lists, that problem does not appear, thanks to the eql-wise comparison.

1. \((\text{subtypep} \ ('\text{list} \ (\text{member} \ (1 \ 2 \ 3))))\)
2. \((1 \ 2 \ 3) \neq\text{eql} \ R \Rightarrow R = (\text{"hello" (1 2 3) foo (1 2 3)})\)
3. As shown in Eq. 5, the emptiness check fails and the answer is correct.

\[
B(\text{list}) \land \neg B((\text{member} \ (1 \ 2 \ 3))) = 0101 \land \neg0001 \tag{5}
\]

Within the literal types kingdom, the only types for which this problem occurs—since the representatives are not supposed to be accessible to the user of subtypep—are then symbol and character. Therefore, only the representatives of these types need to be actually checked when registering member’s elements.

To generate a new symbol, we use alexandria:symbolicate\(^{12}\). The keyword subtype of symbol is also subject to the problem. (Actually, solving the problem for keywords also solves the problem for symbols.) To generate a new character, we first need to know whether it is a base-char or an extended-char. Then we pick a character of that type not registered yet. When all the characters of that type are registered there is nothing to do (since the type is fully represented in the matrix, no misinterpretation can occur).

We have not addressed the problem of a type specifier involving a user class \(C\) and a member expression containing the class prototype of \(C\) yet.

\(^{12}\)https://common-lisp.net/project/alexandria/

### 5.2 Procedure for numeric types

Unlike the literal type kingdom, the range type kingdom does not need an internal state to represent numeric types. Indeed, the expert sub-procedure takes as input an already precise enough representation of the type described. Range type specifiers allow to describe which kind of number is specified (its type, e.g., integer, ratio, etc.), its bounds (inclusive and exclusive, e.g., ((integer (0) 6)) and is able to represent non-bounded intervals through the symbol * meaning infinity (e.g., (float * 0.0) \(\equiv\) \([-\infty;0.0]\)). The range type specifier is as precise as the mathematical range notation. Additionally, the mathematical union, the intersection and complement of these ranges can be expressed equally using the corresponding logical type specifier.

Therefore, to assert about the emptiness of the input type specifier, checking whether the canonicalized version of this interval expression describes the empty range (i.e., nil) is sufficient. The calculation is performed by three successive steps, which we describe in the following sections.

This algorithm suffers from an exponential time and space complexity. However, Baker claims that in practice, that theoretical complexity is not an issue (it only appears for “highly artificial cases”). We have not tried to prove (or invalidate) his statement but Section 6 shows some early results that tend to support his claim.

We use a custom abstraction, the interval class, closer to the mathematical object (with type, bounds and limits slots). Thus we avoid the annoying manipulation of lists (with the many standardized ranges syntaxes). The first step is to write a function range->interval that converts (using pattern matching) a range type specifier to its corresponding interval instance. This function also takes care of the exotic compound forms—such as (unsigned-byte \(s\)) which describes the integer range \([0;2^s-1]\). We also use a similar structure for interval operations to fully discard the list representation.

We also need the following interval functions:

- \((\text{interval-and} \ I_1 \ I_2)\) — returns \(I_1 \land I_2\) if their types are eql, or \(\emptyset\) otherwise.
- \((\text{interval-or} \ I_1 \ I_2)\) — returns \(I_1 \cup I_2\) if their types are eql and \(I_1 \land I_2 \neq \emptyset\), or \(\emptyset\) otherwise.
- \((\text{interval-minus} \ I_1 \ I_2)\) — returns \(I_1 \setminus I_2\) (may return two values when \(I_2 \subset I_1\)) if their types are eql, or \(I_1\) otherwise.
- \((\text{interval-empty-p} \ I)\) — returns whether \(I = \emptyset\).

#### 5.2.1 Type diversity reduction

Functions working with intervals must be aware of the relationship of the types of these intervals. For example, the intersection of two integer intervals might be non-empty whereas the intersection of one integer and one single-float intervals is always null as these two types are disjoint. However, integer and fixnum are different types but the intersection of intervals of such types might be non-empty. The subtype relationship of the types of intervals needs to be introspected to accurately apply some operations (such as intersection or union).

The type number (complex numbers being ignored) is an exhaustive partition of six mutually disjoint types: integer, ratio, single-float, short-float, double-float, and long-float. Baker advises to define what he calls “simple intervals”, that is intervals guaranteed to have their type equal to one of these six types. This
way, as these types are mutually disjoint, operations on intervals of such types have their implementation greatly simplified.

To convert each numeric type into its equivalent using only the six types above, a two-step conversion is required.

(1) For intervals whose type is a supertype of one of these types, the conversion table 1 is used.

(2) For intervals whose type is a bounded subtype (i.e., having defined bounds, not infinity) of these six types, their actual bounds have to be constrained to fit within the bounds of their type, before being converted to their corresponding supertype. For example, (fixnum 12 2^100), has to be converted to (integer most-negative-fixnum most-positive-fixnum), where most-positive-fixnum < 2^100 as a bignum, thus discarding the numbers in between. A similar procedure is applied to the types bit, short-float, single-float, double-float and long-float.

Eventually, the type of every interval is constrained to one of the six types above, with the bounds (if some) of their original type preserved.

**5.2.2 Canonicalization.** To check the emptiness of the interval expression, it is canonicalized. Let $\Gamma$ be the canonicalization function. Its parameter is either an interval or an operation on intervals $\chi$ (intersection, union or complement). $\Gamma$ either returns $\emptyset$, an interval or a union of disjoint intervals—the three possible outcomes of a mathematical interval canonicalization.

First and foremost, anytime $\Gamma$ encounters or returns a union, it must ensure that it is flattened (no nested unions). It must also ensure that the intervals inside the union are disjoint. As shown in Section 5.2.1, intervals with different types are necessarily disjoint. Touching intervals [3] are merged using interval-or.

$\Gamma(\emptyset)$ and $\Gamma(I)$ are straightforward, as shown in Eq. end-$\emptyset$ and Eq. end-$I$. These are the terminal cases of the recursion of $\Gamma$.

$$\Gamma(\emptyset) = \emptyset$$  \hspace{1cm} (end-$\emptyset$)

$$\Gamma(I) = I$$  \hspace{1cm} (end-$I$)

Intersections (and logical type specifiers) are reduced as soon as they are encountered. Their operands need to be processed by $\Gamma$ first (hence the implicit mapping "$k \rightarrow n$"). Eq. and-apply shows how to reduce intersections. The $\Phi$ operator denotes a fold [5] operation using the function $f$. $\Gamma \circ \cap$ denotes the composition of the $\Gamma$ function and the intersection operator. To break it down in a bottom-up fashion:

(1) Eq. and-final — the application of the intersection function.

(2) Eq. and-distribution’ — the distribution of the intersection over the union. Next step is Eq. and-final.

(3) Eq. and-distribution — also the distribution of the intersection over the union. However, $\Gamma(\chi)$ may return an union, leading the execution either to Eq. and-distribution’ or directly to Eq. and-final.

(4) Eq. and-apply — the canonicalization of the $\chi_n$ forms using mapping. The results are then folded using $\Gamma \circ \cap$, thus initiating the recursive intersection distribution.

$$\Gamma \left( \bigcap_{n} \chi_n \right) = \Phi_{\Gamma \circ \cap} \left( \Gamma(\chi)_{k \rightarrow n} \right)$$  \hspace{1cm} (and-apply)

$$\Gamma \left( \chi \cap \bigcup_{n} I_n \right) = \bigcup_{n} \left( \Gamma(\chi) \cap I_n \right)$$  \hspace{1cm} (and-distribution)

$$\Gamma \left( \bigcup_{n} I_n \cap I \right) = \bigcup_{n} \left( \Gamma(I_n \cap I) \right)$$  \hspace{1cm} (and-distribution’)

$$\Gamma(I_1 \cap I_2) = \langle \text{interval-and} \rangle I_1 I_2$$  \hspace{1cm} (and-final)

Complements (not logical type specifiers) are also reduced as soon as they are encountered. Their only operand is first canonicalized. Complementing $U$ in number (the top-level type of the range type kingdom) is equivalent to the difference number $-$ $U$, as shown in Eq. not-apply. The difference canonicalization goes through a similar recursive distribution path than the intersection, that is Eq. minus-distribution and then Eq. minus-apply. Note that this path is taken every time since the interval difference is an internal operation and that its left-hand operand is always $U$.

$$\Gamma(\bar{\chi}) = \Gamma(U - \Gamma(\chi))$$  \hspace{1cm} (not-apply)

$U = \langle \text{type diversity reduction of} \rangle \text{number} *$)

$$\Gamma \left( \chi - \bigcup_{n} I_n \right) = \bigcup_{n} \Gamma(\chi - I_n)$$  \hspace{1cm} (minus-distribution)

$$\Gamma \left( \bigcup_{n} I_n - I \right) = \bigcup_{n} \langle \text{interval-minus} \rangle I_n I$$  \hspace{1cm} (minus-apply)

**5.2.3 Range emptiness check.** Once an interval expression canonicalized, checking its emptiness is trivial. The predicate interval-empty-p, given the result of the first $\Gamma$ call, just returns the Boolean that null-numeric-type-p has to return.

**5.3 Array types and cons type specifiers**

This section presents some preliminary work and research results found on array and cons type specifiers. Obviously, since the implementation of the expert sub-procedures for these kingdoms is still a work in progress, no result nor implementation guidelines are provided here. It does, however, give some insights about how Baker procedure applies to modern Common Lisp implementations such as Sbcl.

Array type specifiers are complex to handle because they are bi-dimensional: it has an element type and bounds (e.g., (array integer (* 2 *)). Internally, Common Lisp implementations do not store which exact type specifier is specified but rather only store...
Implementing Baker’s SUBTYPEP decision procedure

Our implementation is overall slower in the range type kingdom. Heavy users of `member` seems to experience a slower execution. Perhaps, as predicted by Baker, the reason is that the systematic registration of the elements makes the size of the bit-vectors grow quickly, thus making every subsequent operation slower.

For the symbolic type specifiers—primitive types, Clos classes and conditions—our implementation already outperforms SACL’s.

7 CONCLUSION AND FUTURE WORK

Throughout this article we presented our implementation of Baker’s decision procedure. In Section 2 we introduced the Common Lisp type system, the notion of type specifier and some vocabulary. In Section 4 we explained how to pre-process the caller’s type specifiers to make the work of the expert sub-procedures presented in Section 5 easier. We described our implementation for the symbolic, `member`, range and logical type specifiers. We also gave some insights about the implementation for the `array` and `cons` type specifiers. We finally presented some early efficiency measures, which are globally encouraging.

Our implementation is still a work in progress and highly experimental. But with some cleaning and the implementation of both `array` and `cons` expert sub-procedures, it could be a viable alternative to existing subtypep implementations. We will have open sourced its code by then. We still have to find a solution for the `satisfies-type` specifier and the related uncertainty. Indeed, in some situations, `subtypep` still can answer even though the type specifier is involved. For example, in `{subtypep 'string `(and number `(satisfies evenp))}`, as the second operand is guaranteed to be a subtype of `number`, the predicate can safely return false. Finally, a lot of measures on accuracy and efficiency are needed to assert whether Baker’s intuition about his procedure was correct or not.

Even if, in the future, we are to conclude that our implementation is less efficient than those which already exists, Baker’s algorithm would still likely to improve the predicate’s accuracy. Lispers would then have the ability to choose whichever `subtypep` implementation fits their needs the best.

REFERENCES