A Comparison of Software Refinery and ASF+SDF

Arie van Deursen
Centrum voor Wiskunde en Informatica

Disclaimer This document reflects the situation at the time of writing, which is November 1996. Both ASF+SDF and Refine may have undergone significant changes since then. The document is provided "as is", and no guarantee about the correctness of the contents can be given. The information about ASF+SDF is taken from several years of experience with ASF+SDF by the author; the information about Refine is merely based on the Refine manuals together with two weeks of using Refine.

Executive Summary Software Refinery is a tool for building tools for analyzing and modifying software automatically. The tools can either be built directly using Refinery, or as an extension of one of the Refine Language Tools such as Refine/COBOL or Refine/2000.

Typical application areas are those where existing tools are not available or not satisfactory. Examples include systems built using proprietary languages or site-specific dialects of COBOL, systems requiring special year 2000 modifications or special forms of global analysis, etc. To build such re-engineering tools, Software Refinery uses the Refine Language for programming such tools. The language features set-theoretic operations, logic, lisp-like symbols, objects and inheritance, grammar definitions, and the use of concrete syntax for doing pattern matching.

Users of Software Refinery can benefit from a large number of well-documented API's (Application Programming Interface). Refine/COBOL gives access to (various dialects of) COBOL; Refine/Workbench provides data structures and functions for building control flow graphs, structure charts, and coding standards reports; Refine/2000 provides functions for tracing date-related variables and modifying incorrect date operations.

This document has been prepared for the Resolver project. It aims at providing an introduction to the features and capabilities of Software Refinery. It contrasts Software Refinery with ASF+SDF, the meta-environment for building tools developed at CWI/UvA. A full suite of tools for a toy-language called CALC is given in the document in both Refine and ASF+SDF.

The document can be used (1) for assessing the suitability of Refine for use in a particular renovation project; (2) to position Software Refinery and ASF+SDF in the landscape of re-engineering tools; and (3) for sharing the available knowledge on Refinery with the other Resolver researchers.

1 Introduction

The Software Refinery environment, provided by Reasoning Systems, Palo Alto, is a framework for building re-engineering tools. In the last decade, it has been used in numerous successful projects involving the migration or renovation of legacy software systems. As an example, Markosian et. al describe the use of Refinery for the automatic modularization of a 40,000 LOC COBOL application from Boeing [9]. Other applications are given in [4, 17]. The roots of Refinery go back to the research on knowledge based systems performed at Kestrel Institute [21].

In this document, we summarize the Refine language and the Software Refinery system, and contrast it briefly with the ASF+SDF formalism and the ASF+SDF Meta-Environment [1, 8, 5]. In aims, Software Refinery and ASF+SDF are rather similar: both provide support for the development of tools for processing software (program sources). In this document, we show how to use both for building tools for an example language called CALC. We give the complete CALC definitions in both Refine and ASF+SDF, and use them to highlight the differences and similarities between ASF+SDF. Then we discuss three extensions of Refine (you may also think of them as libraries) in more detail: The *workbench*, Refine/COBOL, and Refine/2000.

The paper was written after two weeks of intensive use of Refinery (including a one week course), and after several years of experience with ASF+SDF. It is intended for those who are (1) looking for a renovation tool and are considering using Refine; (2) familiar with only one of ASF+SDF and Refine, and would like to know more about the other. *The aim of this paper is to give an overview of the Refine language and its underlying technology, and to show how it can be used for tool building.* Readers need not be familiar with ASF+SDF; those who are not can skip the paragraphs comparing ASF+SDF and Software Refinery.

Acknowledgments I would like to thank Jan Heering, Jasper Kamperman, Paul Klint, Lawrence Markosian, Theo Wiggerts, and David Zimmerman for answering many questions and for being fruitful discussion partners. The CALC specification of Appendix A comes from Reasoning Systems as part of their training material.

2 Overview of Software Refinery

Software Refinery is a framework supporting the construction of re-engineering tools [12]. Typical tools are converters between COBOL dialects, translators from COBOL to, e.g., PL/I, maintenance tools for proprietary 4GLs dialects, tools for migrating proprietary DBMS to, e.g., IMS; an overview is provided in the bibliography given in [17].

Software Refinery is based on the observation that all such tools are programs that manipulate software, i.e., programs operating on pieces of source code. Typical functionality of such tools includes parsing, program transformation, program

translation, report presentation, source code browsing, pretty printing, etc. Software Refinery provides a language, Refine, which makes it easy to write such tools. Moreover, it consists of a well-considered library of language-independent data structures and functions that are necessary when writing re-engineering tools (Refine/Dialect for parser generation, pretty printing, and for the extension to include "surface syntax" (concrete syntax) patterns in the Refine language [10], Refine/Intervista for GUI functionality [11], Refine/Workbench for presenting control flow graphs, structure charts, and coding standards reports [16]).

To date, Software Refinery has been used to construct tools for a wide variety of languages. Commercially available are four of such "instantiations": Refine/COBOL, -/Fortran, -/Ada, and -/C. The Refine/COBOL, for example, includes a parser for various COBOL dialects, generation of control flow graphs for paragraphs, inter-paragraph control flow, copy book expansion, COBOL coding standards checking, syntactic browsing, and pretty printing. Refine/COBOL comes with the full grammar of COBOL, which makes it easy to customize it to include additional functionality. For example, Reasoning is about to release a Year-2000 extension for their COBOL environment.

The distinguishing property of Software Refinery is its high level of customizability. The five instantiations, Refine/C, -/Ada, -/Fortran, and -/Year 2000, are all well-documented, and can be easily adapted by the tool users, if necessary. Moreover, Software Refinery is ideally suited for building new re-engineering tools. As they state themselves [9, p.60],

> there are two rules of thumb to decide whether a re-engineering task is amenable to the Software Refinery approach. The first is that the task should not be "too easy" — for example, readily solved by regular expression-based tools such as AWK or PERL (...).
>
> Another rule of thumb is this: can you describe a procedure that an analyst can use to perform the analyses and modifications by hand? If the answer is "yes" then it may be feasible to automate the procedure. The more difficult the re-engineering task, the greater the leverage that the Software Refinery technology provides relative to standard techniques.

The sections below should give an impression how difficult it is to write such tools.

3 The Refine Language by Example

We introduce the Refine language by discussing the definition of a simple toy-language "CALC". This is the language used during the Refine course[1] offered by Reasoning

[1] The course takes one week and covers the use of Refine. The practical material includes building tools for CALC; Moreover, groups following the course can bring their own project. More information is available in [18].

Systems. Appendix A gives the "official" Reasoning solution to the exercises during that course. For those familiar with ASF+SDF, the definition of CALC in ASF+SDF is given in Appendix B.

3.1 Defining Syntactic Domains

When writing a tool for a particular language L, the first thing to do is to give a *domain model* for L. Since Refine is an *object-based* language[2], the main ingredient of this model is the set of object classes that constitute the nodes in the abstract syntax trees. As can be seen in Section A.1, example classes are for `CALC-EXPRESSION`, `CALC-STATEMENT`, etc.

The attributes (instance variables) of a class C are defined as maps from C to some target domain. When defining the language domain model, the attributes given are the links to the children in the abstract syntax tree. For example, the class `ADD-EXPRESSION` has two attributes, `Add-Arg1` and `Add-Arg2`, containing the left and right-hand side of the add expression. The `subtype-of` construct is used to define inheritance of classes; lower classes inherit all attributes of the higher ones.

Attributes can be used to represent such branches in abstract syntax trees; they can also be used to store computed attributes in the abstract syntax trees (in the sense of attribute grammars). The attributes used for branches in the trees are special in refine, and therefore are explicitly declared as such, using, e.g.,

```
define-tree-attributes('add-expression, {'add-arg1, 'add-arg2})
```
[3]

The use of quotes in this declaration reveals the influence of Lisp on the Refine language. Refine includes a Lisp-like type called "symbol", it is modularized in the way Common Lisp is modularized (mainly using packages giving all identifiers some dedicated prefix), and it allows the use of global variables. It is compiled into Common Lisp, and there is a mechanism to call any Common Lisp function from a Refine program.

The definition of the domain model allows one to build abstract syntax trees, using, e.g., calls to `make-object`. Another way is to attach *surface syntax* (concrete syntax) to classes. In Section A.2, the Refine grammar for the surface syntax of the CALC language is shown. For example:

```
add-expression ::= [add-arg1 "+" add-arg2] builds add-expression
```

The grammar is used to generate an LALR parser[4] (in fact, Bison code is generated, the GNU Yacc implementation). The `builds` clause indicates that when during parsing the grammar rule is applicable, the effect is that a node of class `add-expression` is to be built. Alternatively, a `semantics` f clause can be given, which indicates that Refine function f is to be called when reducing this grammar

[2] It is not fully *object-oriented* because of its limited support for *encapsulation* and *data hiding*.

[3] Observe that Refine is case-insensitive, i.e., `add-arg1` is the same as `Add-Arg1`.

[4] Refine is currently working on an extension to *generalized LR parsing* — the technique that is used for parsing in the ASF+SDF Meta-Environment [6, 20].

rule. From these rules a pretty printer (unparser) is generated as well, and, if necessary, rules to be used for pretty printing only can be given.

The CALC grammar in ASF+SDF is given in Module Calc (B.1). The domain model of Refine corresponds to a signature in ASF+SDF terminology. In ASF+SDF, the signature definition is derived from the SDF specification, which defines lexical, concrete and abstract syntax at once. For that reason, the ASF+SDF specification is much shorter.

In Refine, it is possible to associate different surface syntaxes with one language model. In ASF+SDF every signature has exactly one concrete syntax (which usually is sufficient); extra concrete syntactic constructs can be added by defining them seperately, together with rewrite rules for mapping them to the signature corresponding to the domain model to be used.

3.2 Collecting Context-Sensitive Information

The scope analyzer of Section A.3 adds attributes to the abstract syntax tree for representing declaration and use of identifiers. Each declaration of an identifier is given the attribute `REFERENCES`, which is to be filled with all occurrences of uses of that identifier. If the variable `x` is used four times in the statements of a CALC program, then the value of the `REFERENCES` attribute associated with the declaration of `x` will be a set containing pointers to these four occurrences of `x`.

The converse of the `REFERENCES` attribute is the `REF-TO` attribute, which for each use of an identifier in some CALC statement gives a pointer back to the occurrence of that same identifier in the declarations part.

The function computing the values for these attributes, is called `SCOPE-CALC-PROGRAM`. It takes a program, and decorates it with the appropriate values for the `REF-TO` and `REFERENCES` attributes. To do so, it first uses the Refine built-in function `descendants-of-class`, which given an abstract syntax tree node, follows all syntactic attributes and returns the set of all subtrees of the given class name. Next, it uses a so-called *transform* statement (here of the form P_1 & P_2&$P_3 \Rightarrow Q$) to express which logical formula must be fulfilled when all the `REF-TO` and `REFERENCES` are given their value in the given tree: For every identifier occurrence `ref` and identifier definition `id` dealing with the same identifier value, the `REF-TO` attribute of `ref` should be the `id`.

This scope analyzer illustrates some further distinctions between ASF+SDF and Refine:

- In ASF+SDF, trees are always compared using structural equivalence. Refine, by contrast, manipulates *occurrences* (pointers) in trees. For example, in ASF+SDF the `REFERENCES` attribute example discussed above would be written as a set {`x, x, x, x`}, which would be the same as just {`x`}. In Refine, each `x` has an identity and knows where it comes from; hence the four `x`'s in the references set are different.

The ASF+SDF approach here is much easier to understand (and to debug), although the Refine approach is more powerful.

- ASF+SDF has no equivalent of the `descendants-of-class` function. To achieve this, the specifier must write out an explicit function (or collection of hidden functions) to perform the program traversal.

- The transform statement changes the (attributes of) an abstract syntax tree. By contrast, ASF+SDF is purely functional: it can compute values for certain (sub)trees, but does not store these values into the tree.

- Refine has a large set of built-in operations for manipulating sets, lists, integers, reals, tuples, strings, etc.

  ASF+SDF has *no* built-in data types. Instead, such data types come from a library of standard modules.

- As a last remark, the Refine compiler infers types for all variables, functions, and built-in operations. Unfortunately, it gives all user-defined *objects* (as opposed to built-in data types such as sets, lists, etc.) the same type (`object-type`), i.e., although each class corresponds to a type, the typechecker does not use this.

  In ASF+SDF, every term and variable is of a certain sort, which can be determined at compile time (in that sense it is strongly typed).

3.3 Type Checking

The type checker reports about typing errors in CALC expressions and statements. The rules in Section A.4 cover all cases to be checked, such as integer and real constants, identifier references (whose types are found using the scope rules of the previous section), various forms of arithmetic expressions, and assignment statements.

The Refine *rule* is a special form of a *transform* statement. Every rule is to be applied to an abstract syntax tree node. The Refine function `postorder-transform` called in function `TYPE-CHECK-CALC-PROGRAM` takes care of trying all rules listed in its second argument on each node of the given program. There also exists a similar function called `preorder-transform`.

Observe the use of concrete syntax in the rules for typechecking the arithmetic operators and the assignment statement, such as for example in the condition `a = '@lhs := @rhs'` of rule `TYPE-CHECK-ASSIGNMENT`. The *pattern* `'@lhs := @rhs'` uses the surface syntax for objects of type `ASSIGNMENT-STATEMENT`. Object `a` is matched against this pattern, and if it succeeds, the variables `lhs` and `rhs` are filled with the left-hand side identifier and the right-hand side expression of the assignment, respectively.

Comparing this with ASF+SDF, we see that:

- Refine allows surface syntax to be used within a Refine program. The pattern is written between `‘...’` pairs, and should not contain itself the closing "’" back quote character.

  In ASF+SDF, the integration of concrete and abstract syntax is much smoother: every term used is written in concrete syntax, and there are no special delimiters needed for recognizing pieces of program.

- ASF+SDF does not have the operators for performing a pre- or post order traversal of a tree. Instead, again explicit traversal functions need to be written.

  On the other hand Refine programs do have to make sure that the rules are not applied over and over again; To achieve this, all rules in the type checker have a condition checking whether the `data-type` attribute of this node is still undefined.

3.4 Remaining Calc specification

The specification of the Calc coding standards checker given in Section A.5 is a simple extension of the type checker to give extra warnings for identifiers that are not used, not declared, or not assigned. The specification of the Calc expressions simplifier in Section A.6 tries to reduce expressions by applying laws such as $X + 0 = X$. In ASF+SDF these algebraic equalities can be directly specified as equations. In Refine, they are expressed as *transforms*, modifying the abstract syntax tree.

The last part of the Refine CALC definition shows how to build a browser for CALC programs. It provides for syntactic browsing, showing the structure of syntactic components inspected. Moreover, it supports *hyper links*, and the function `SCOPE-MOUSE-HANDLER` makes sure that a click on an identifier reference highlights the corresponding identifier definition and vice versa using the `ref-to` and `references` attributes.

In ASF+SDF, syntactic browsing comes for free using the GSE Generic Syntax-directed Editor. Highlighting context-sensitive information is at this moment impossible; A combination of *origin tracking* [5, Chapter 7] and pretty printing might be able to fill this gap in the near future.

4 Other Refinery Features of Interest

4.1 The Workbench

A programming language is as useful as the libraries that come with it. The Refine language comes with the *workbench* [16], a library for manipulating structure charts (a graphical representation of the overall, inter-procedural structure), control-flow graphs (for recording the flow of control between individual statements in a proce-

dure), and coding standards reports (for informing the programmer about def/use relations of variables, coding standard violations, etc.).

The Workbench manual describes the complete Workbench API — the Application Programming Interface consisting of all functions, classes, variables etc. that a programmer can use when building tools with the workbench. The functionality includes data structures for building graphs, functions for connecting (hyper-linking) graphs and source code fragments, procedures for displaying graphs in windows, for customizing the interactive display of structure charts, exporting data structures to external CASE tools, etc.

The Workbench's graphical component is based on Refine/Intervista [11], the graphical component of Refine. INTERVISTA is an API for constructing diagrams, pop-up menus, and mouse-sensitive text-windows. Currently, it is based on Common Windows, the X-windows interface of Common Lisp, an implementation Reasoning is not particularly happy with. In the future, Reasoning plans to re-implement it on basis of a Java library.

4.2 Debugging and Efficiency

When writing large Refine applications, debugging becomes a relevant issue. The typical Refine edit-compile-run cycle consists of using Emacs for editing the Refine program, compiling it, loading it into Refine, testing a particular function, and inspecting the results. For the latter, Software Refinery provides an "inspector" for viewing object "frames", i.e., an object and all its attribute values. Each object has a "magic number" associated with it, and this number is given as a sort of pointer to the inspector. A debugging facility allows the user to set breaks and inspect intermediate values. The debugger, however, often operates at the level of the *generated* lisp code, rather than the Refine source code, which makes it difficult to interpret its messages. Likewise, compilation error messages are often hard to understand.

Refine objects are built once, and during a Refine session they cannot be destroyed (free-ing the memory taken by them). There is no automatic garbage collection of objects either. There is a function, erase-object, that undefines the attributes of an object (which also should break inverse attributes to it) and turns it into an "erased object" but it does not allow recovery of storage space. Typical Software Refinery applications that are applied to very large systems extract and save the required information from each module to file (via the *persistent object base*, see below), then the application is exited and started again, accessing whatever saved information is required.

Another concern could be the efficiency when dealing with very large Refine applications, or when Refine has to process huge amounts of data (e.g., a large collection of legacy programs). Concerning the first issue, Software Refinery provides *profiling tools* which help to record and display performance data on time and swap space consumed [16, Chapter 5]. Moreover, Reasoning systems has compiled a list containing advice on how to optimize Software Refinery applications [15].

As for the second issue, the size of the data analyzed, As briefly mentioned above, Software Refinery supports a *persistent object base*. In other words, there is no need to keep all data in core, and the Refine programmer can indicate which (intermediate) results should be dumped on file, rather than be kept in main memory [16, Chapter 4].

5 COBOL Specific Extensions

5.1 Refine/COBOL

An overview of the functionality of the Refine/COBOL language tool is given at Reasoning System's WWW site [19, 14]. We have attached a copy of this page to the current document. Refine/COBOL is an extensible tool. The COBOL specific programmer's manual consists of the full COBOL language model (grammar) used [13].

5.2 The Refine/Year 2000 Solution

One possible application of Refine/COBOL is to build an impact analysis or automatic correction tool for the year 2000 problem as occurring in COBOL software. Anyone with a good idea on how to solve the year 2000 problem for COBOL in a way that can be automated, should consider using Refine/COBOL to implement their tool.

Reasoning Systems themselves have done exactly this: extend Refine/COBOL with their own year 2000 solution, which is marketed as a separate product. At the time of writing (September 1996) only a demo version was available. The discussion below is based on a 20 minute demo and the Software Refinery "Year 2000 white paper" [2]. The Refine/2000 solution consists of the following steps:

Building the system model: The system model describes all modules, datafiles, data bases, and jobs that comprise the organization's software portfolio, as well as the interfaces and data flow between these. The system model is used to identify subsystems that can be processed in isolation.[5]

Most of the system model is automatically generated by Refine/Cobol after it has parsed and analyzed the JCL and Cobol programs. The system model is presented as an interactive graph where different icons represent the elements of the model, and links represent the relationship between the elements.

Analysis: The analysis phase determines *type information* about the data elements, and identifies those that actually might need year 2000 correction. Using a "data-flow-directed inference process" this information is propagated through the program logic.

[5] The full data model used in the system model is considered proprietary information.

The type information of a data type is a collection of properties that characterizes the values that can be stored in the data element. The sort of information inferred includes:

- The units of the data element (years, quarters, months, days, etc.);
- Whether the data element is an absolute date or a time interval;
- The range of values that the data element will store;
- Related data elements that serve as higher-order or lower digits.

The initial type clues are referred to as *seeds*, and are given a *confidence level*. The most reliable seeds come from known date manipulating or calendar routines. A less reliable source is pattern matching the names of data elements against a library of names such as `YYMMDD`, `YEAR`, `ANNIVERSARY`, etc.

Analysis is performed at both intra- and inter-module level. When an intra-module analysis is complete, type information about externalized time-related date elements is communicated to the inter-module analysis. These are then propagated, using the system model, to other modules that read or write the same data elements. Intra- and inter-module analyses are alternated until all type information has been propagated.

Program correction: Infected programs can be corrected using the *widen-the-data* approach, which changes centuries to four digits. First the data division is corrected, carefully taking care of aliases (RENAMES, REDEFINES). The procedure division must be corrected to cater for number constants such as "95". Moreover, the user is warned about screen maps (which may not have sufficient space for four digits), and JCL scripts which might have to be adapted (e.g., if they contain constants).

As an example of how to specify such a program correction in the Refine language, consider the following [3]:

```
rule CORRECT-DATE-DIFFERENCES
  a = 'COMPUTE @RESULT = @D1 - @D2'
  & date(D1)
  & date(D2)
-->
  a = 'CALL COMPUTE-INTERVAL USING @D1 @D2'
```

This is a Refine *transform* rule, similar to the ones given for the Calc simplifier in Section A.6. The rule recognizes that the node a matches the surface syntax for a computation involving subtraction. Moreover, the two conditions inspect the results of the analysis stored in the `date` attribute, indicating that both arguments are indeed dates. It then changes the node a to call a library routine `COMPUTE-INTERVAL` for computing date differences correctly.

Data correction: In the widen-the-data approach, the years stored in two digits in databases need to be expanded. Refine/2000 offers the options of a *batch conversion* or an *on-the-fly conversion.*

In the future, Refine/2000 will also support other correction schemes, such as (sliding) windows. In general, the widen-the-data approach is considered the least attractive solution, precisely because of the need for data correction or screen adaptation.

On the Use of Slicing Refine/2000 uses *program slicing* [23, 7, 22] during the analysis phase, as discussed in Reasoning's presentation [3]. Slicing is a technique for determining those parts of a program that are responsible for giving a certain variable a particular value. Refine/2000 is built on top of Refine/COBOL; their slicer, however, has been designed to be *language-independent.* Future work of Reasoning includes connecting the generic slicer to other language models (e.g., PL/I).

During the analysis phase, Refine/2000 first detects which variables are potential dates. Then it uses *forward* slicing,[6] which finds all the lines of code impacted by the value of such a variable. This information is used to find other, related date fields. In [3], the following example is given:

```
01 ...
  05 YEAR            PIC 99.
  05 ANV             PIC 99.
01 ...
  05 PARTS-PER-ORDER PIC 99.
  05 PARTS-COUNT     PIC 99.
01 ...
01 TEMP              PIC 99.
P1.
  MOVE YEAR TO TEMP.
  ...
  MOVE TEMP TO ANV.
P2.
  MOVE PARTS-PER-ORDER TO TEMP.
  ...
  MOVE TEMP TO PART-COUNT.
```

Assuming that the slicer knows that `YEAR` is a date field (derived, e.g., from the name), it finds out that within its forward slice (paragraph `P1`), the variables `ANV`

[6] Slicing is known to be an expensive technique, and for slicing in the presence of procedure calls (PERFORM), unstructured control flow (GOTO), and aliasing (REDEFINE) there are many unsolved (efficiency) problems. Reasoning's claimed solution to these problems has not been publicized. Reasoning attempts to minimize these costs by making the slicer *lazy*: only the slice information that is actually needed because of some explicitly request is computed. The details of the slicing method used in Refine/2000 are proprietary.

and `TEMP` are used as dates. Outside this slice (in paragraph `P2`), it does *not* assume that `TEMP` or `ANV` are dates. In particular, Refine/2000 does not conclude from the last statement `MOVE TEMP TO PART-COUNT` that `PART-COUNT` must be a date; This statement is out of the scope of the forward slice of `YEAR`, and hence in that statement it is not assumed that `TEMP` is a date field, and therefore it is not concluded either that `PART-COUNT` is a date field.

6 Concluding Remarks

6.1 The Research Perspective

There are a number of areas in the field of language technology in which both Reasoning Systems and the ASF+SDF group are interested: Reasoning because of the commercial value and potential applications, and ASF+SDF for creating new knowledge in the field of re-engineering. Some example areas of mutual interest include:

Generalized LR parsing: extensively investigated for ASF+SDF since 1989, and to be incorporated in future versions of Refine/Dialect.

Modular COBOL grammar: A COBOL syntax definition which can be easily split in relevant subsets: Currently Refine provides one monolithic grammar for various dialects; for ASF+SDF a modular COBOL grammar exists.

Origin tracking: Refine uses explicit pointer manipulations to link analysis results (graphs, reports) to the sources. ASF+SDF tries to simplify this by building such links *automatically*.

Syntactic querying: Refine has a number of high-level set-theoretic, logical, and pattern matching operators that can be used to find source fragments meeting certain criteria. Currently, the ASF+SDF group is investigating a *query algebra*, which has explicit support for, e.g., "regions" and easy composition of queries.

Data flow analysis: Refine/2000 provides COBOL-specific data flow analysis (DFA) and program slicing for tracing dates. For ASF+SDF, a generic DFA architecture has been proposed recently, which can handle control flow normalization and alias analysis.

Most likely, there are many more areas of mutual interest. A closer cooperation between Reasoning Systems, and the ASF+SDF group, for sharing ideas, experiences, and problems may well be beneficial for both parties.

6.2 The Application Perspective

Software Refinery is a tool for building tools for analyzing and modifying software automatically. The tools can either be built directly using Refinery, or as an extension of one of the Refine Language Tools such as Refine/COBOL or Refine/2000.

Typical application areas are those where existing tools are not available or not satisfactory. Examples include systems built using proprietary languages or site-specific dialects of, e.g., COBOL, systems requiring special year 2000 modifications or special forms of global analysis, etc.

Comparing the ASF+SDF and Refine formalisms, the ASF+SDF language is significantly easier to use than Refine. Refine is a wide-spectrum language and includes many features. ASF+SDF, by contrast, is much simpler being based exclusively on the side-effect free notion of term rewriting. Moreover, the syntactic component (parser generation) is much better integrated. In terms of usage, it may be the case that ASF+SDF is better suited for writing language transformations. Refine, on the other hand, has better ways of attributing trees, which is useful when doing complicated language analysis.

Refine tool builders will have to be familiar with Lisp, object orientation, logic, set theory, and compiler construction technology. Not everyone employed in the software industry is equiped with this knowledge, but those who are can take advantage of all capabilities of Software Refinery, enabling them to construct re-engineering tools with moderate effort.

References

[1] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

[2] W. Brew. Reengineering your software for the millennium. Reasoning Systems Inc, 16 pages, June 1996.

[3] W. A. Brew, K. Schimpf, and L. Z. Markosian. *Application of Program Slicing and Program Transformation to Solving the Year 2000 Problem.* Reasoning Systems, Palo Alto, California, 1996. Presentation at the 5th Reengineering Forum. 17 Slides.

[4] S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating software re-engineering. In *Proceedings 14th International Computer Software and Applications Conference (COMPSAC)*, pages 314–322. IEEE, 1990. Also in R. S. Arnold (Ed.), *Software Reengineering*, pages 275–283, IEEE, 1993.

[5] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific Publishing Co., 1996.

[6] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, SE-16:1344–1351, 1990.

[7] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering ICSE-14*. IEEE, 1992.

[8] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[9] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, 1994. Special issue on reverse engineering.

[10] Reasoning Systems, Palo Alto, California. *DIALECT user's guide*, 1992.

[11] Reasoning Systems, Palo Alto, California. *INTERVISTA user's guide*, 1992.

[12] Reasoning Systems, Palo Alto, California. *REFINE User's Guide*, 1992.

[13] Reasoning Systems, Palo Alto, California. *Refine/COBOL Programmer's Guide*, 1992.

[14] Reasoning Systems, Palo Alto, California. *Refine/COBOL User's Guide*, 1992.

[15] Reasoning Systems, Palo Alto, California. *Optimizing Software Refinery Applications: Guidelines and Techniques*, 1993.

[16] Reasoning Systems, Palo Alto, California. *Software Refinery 4.0 Beta New Features*, 1993.

[17] Reasoning Systems, Palo Alto, California. *Bibliography: Applications of Software Refinery and Refine Language Tools*, 1996. URL: `http://mosaic.reasoning.com/rsi-bibliography.html`.

[18] Reasoning Systems, Palo Alto, California. *REFINE Training Manual*, 1996. URL: `http://mosaic.reasoning.com/REFINE-Training-Course.html`.

[19] Reasoning Systems, Palo Alto, California. *Refine/Cobol, Graphical workbench for maintaining, improving, and reengineering existing COBOL systems*, 1996. 4 pages. Available at URL http://mosaic.reasoning.com/ REFINE-COBOL.html.

[20] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.

[21] D. Smith, G. Kotik, and S. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Transactions on Software Engineering*, SE-11(11), November 1985.

[22] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[23] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

A The CALC Language in Refine

The definition of the CALC project using Refine. These are the solutions offered by Reasoning Systems; observe that, e.g., the scope analyzer contains several alternative ways of defining the scope analysing function.

A.1 Domain Model

```
!! in-package("RU")
!! in-grammar('user)

%%% ------------------------------------------------------------------------
%%%                   DOMAIN MODEL FOR CALC LANGUAGE MODEL
%%% ------------------------------------------------------------------------

%%% ------------------------------------------------------------------
%%% Object Classes and Attributes

var CALC-OBJECT:              object-class subtype-of user-object

var   CALC-EXPRESSION:        object-class subtype-of calc-object

var     IDENTIFIER-REF:       object-class subtype-of calc-expression
var     Identifier-Ref-Name:  map(identifier-ref, symbol)              = {||}

var     INTEGER-CONSTANT:     object-class subtype-of calc-expression
var     Integer-Value:        map(integer-constant, integer)          = {||}

var     REAL-CONSTANT:        object-class subtype-of calc-expression
var     Real-Value:           map(real-constant, real)                = {||}

var     ADD-EXPRESSION:       object-class subtype-of calc-expression
var     Add-Arg1:             map(add-expression, calc-expression)    = {||}
var     Add-Arg2:             map(add-expression, calc-expression)    = {||}

var     SUB-EXPRESSION:       object-class subtype-of calc-expression
var     Sub-Arg1:             map(sub-expression, calc-expression)    = {||}
var     Sub-Arg2:             map(sub-expression, calc-expression)    = {||}

var     MULT-EXPRESSION:      object-class subtype-of calc-expression
var     Mult-Arg1:            map(mult-expression, calc-expression)   = {||}
var     Mult-Arg2:            map(mult-expression, calc-expression)   = {||}

var     DIV-EXPRESSION:       object-class subtype-of calc-expression
var     Div-Arg1:             map(div-expression, calc-expression)    = {||}
```

```
var     Div-Arg2:               map(div-expression, calc-expression)      = {||}

var   CALC-STATEMENT:           object-class subtype-of calc-object

var     ASSIGNMENT-STATEMENT:   object-class subtype-of calc-statement
var     Assignment-Lhs:         map(assignment-statement, identifier-ref) = {||}
var     Assignment-Rhs:         map(assignment-statement,
                                    calc-expression)                      = {||}

var     PRINT-STATEMENT:        object-class subtype-of calc-statement
var     Print-Args:             map(print-statement, seq(calc-expression))
                                  computed-using print-args(@@)           = []

var   IDENTIFIER:               object-class subtype-of calc-object
var   Identifier-Name:          map(identifier, symbol)                   = {||}

var   VARIABLE-DECLARATION:     object-class subtype-of calc-object
var   Variable-Identifier:      map(variable-declaration, identifier)     = {||}
var   Variable-Type:            map(variable-declaration, calc-type)      = {||}

var   CALC-TYPE:                object-class subtype-of calc-object

var     REAL-TYPE:              object-class subtype-of calc-type

var     INTEGER-TYPE:           object-class subtype-of calc-type

var   CALC-PROGRAM:             object-class subtype-of calc-object
var   Program-Name:             map(calc-program, symbol)                 = {||}
var   Program-Declarations:     map(calc-program, seq(variable-declaration))
                                  computed-using program-declarations(@@) = []

var   Program-Statements:       Map(Calc-Program, seQ(Calc-Statement))
                                  Computed-Using program-statements(@@)   = []


%%%  ------------------------------------------------------------------
%%%  Tree Attribute Definitions

form DECLARE-CALC-TREE-ATTRIBUTES
    define-tree-attributes('identifier-ref,       {'identifier-ref-name});
    define-tree-attributes('integer-constant,     {'integer-value});
    define-tree-attributes('real-constant,        {'real-value});
    define-tree-attributes('add-expression,       {'add-arg1, 'add-arg2});
    define-tree-attributes('sub-expression,       {'sub-arg1, 'sub-arg2});
    define-tree-attributes('mult-expression,      {'mult-arg1, 'mult-arg2});
```

```
    define-tree-attributes('div-expression,       {'div-arg1, 'div-arg2});
    define-tree-attributes('assignment-statement, {'assignment-lhs,
                                                   'assignment-rhs});
    define-tree-attributes('print-statement,      {'print-args});
    define-tree-attributes('variable-declaration, {'variable-identifier,
                                                   'variable-type});
    define-tree-attributes('identifier,           {'identifier-name});
    define-tree-attributes('calc-program,         {'program-name,
                                                   'program-declarations,
                                                   'program-statements})
%%%  ------------------------------------------------------------------
%%%  Function to build a calc-program programatically

function BUILD-SMALL-CALC-PROGRAM () : calc-program =
  let (the-program = make-object('calc-program),
       var-x-decl  = make-object('variable-declaration),
       var-y-decl  = make-object('variable-declaration),
       var-x       = make-object('identifier),
       var-y       = make-object('identifier),
       assign-1    = make-object('assignment-statement),
       assign-2    = make-object('assignment-statement),
       print-stat  = make-object('print-statement),
       x-ref-1     = make-object('identifier-ref),
       x-ref-2     = make-object('identifier-ref),
       y-ref-1     = make-object('identifier-ref),
       y-ref-2     = make-object('identifier-ref),
       lit-int-zero = make-object('integer-constant),
       lit-real-one = make-object('real-constant),
       x-times-y   = make-object('mult-expression),
       int-type    = make-object('integer-type),
       r-type      = make-object('real-type))

   program-name(the-program)         <- 'EXAMPLE;
   program-declarations(the-program)  <- [var-x-decl, var-y-decl];
   program-statements(the-program)   <- [assign-1, assign-2, print-stat];

   variable-identifier(var-x-decl)   <- var-x;
   variable-identifier(var-y-decl)   <- var-y;

   variable-type(var-x-decl)              <- int-type;
   variable-type(var-y-decl)              <- r-type;

   identifier-name(var-x)            <- 'X;
   identifier-name(var-y)            <- 'Y;
```

```
   identifier-ref-name(x-ref-1)                <- 'X;
   identifier-ref-name(x-ref-2)                <- 'X;

   identifier-ref-name(y-ref-1)                <- 'Y;
   identifier-ref-name(y-ref-2)                <- 'Y;

   integer-value(lit-int-zero)       <- 0;
   real-value(lit-real-one)          <- 1.0;

   assignment-lhs(assign-1)          <- x-ref-1;
   assignment-rhs(assign-1)          <- lit-int-zero;

   assignment-lhs(assign-2)          <- y-ref-1;
   assignment-rhs(assign-2)          <- lit-real-one;

   print-args(print-stat)            <- [x-times-y];

   mult-arg1(x-times-y)              <- x-ref-2;
   mult-arg2(x-times-y)              <- y-ref-2;

   the-program
```

A.2 CALC Grammar

```
!! in-package("RU")
!! in-grammar('syntax)

% ------------------------------------------------------------------
%                 GRAMMAR FOR CALC LANGUAGE MODEL
% ------------------------------------------------------------------

grammar CALC
   file-classes calc-program
   productions
      identifier-ref        ::= [identifier-ref-name]      builds identifier-ref,
      integer-constant      ::= [integer-value]            builds integer-constant,
      real-constant         ::= [real-value]               builds real-constant,
      add-expression        ::= [add-arg1 "+" add-arg2]    builds add-expression,
      sub-expression        ::= [sub-arg1 "-" sub-arg2]    builds sub-expression,
      mult-expression       ::= [mult-arg1 "*" mult-arg2]  builds mult-expression,
      div-expression        ::= [div-arg1 "/" div-arg2]    builds div-expression,
      assignment-statement  ::= [assignment-lhs ":="
                                 assignment-rhs]           builds assignment-statement,
      print-statement       ::= ["print" "("
```

```
                                    print-args + "," ")"]   builds print-statement,
      identifier            ::= [identifier-name]          builds identifier,
      variable-declaration  ::= ["var" variable-identifier
                                    ":" variable-type]     builds variable-declaration,
      calc-program          ::= ["program" program-name
                                 program-declarations * ";"
                                 "begin"
                                    program-statements * ";"
                                 "end"]                    builds calc-program,
      real-type             ::= ["real"]                   builds real-type,
      integer-type          ::= ["integer"]                builds integer-type

   precedence
      for calc-expression brackets "(" matching ")"
        (same-level "+", "-" associativity left),
        (same-level "*", "/" associativity left)

   special-syntax
      ["#id" identifier] in-all-ancestors
end

#||

By default, this grammar allows everything to be parsed at top level.
As a result, the parser might have to determine whether a SYMBOL
type start symbol, is an IDENTIFIER or an IDENTIFIER-REF.
Introduction of the "special-syntax" clause resolves this ambiguity and prevents the
reduce/reduce conflict that would occur without it.

Type the following sequence of commands to the REFINE Command Interface,
to observe how the calc parser uses this special syntax:
   (in-grammar 'calc)
   :parse xxx
   (pup)
   :parse #id xxx
   (pup)
   :parse var xxx: integer
   (pup)

||#

!! in-grammar('user, 'user)

function PARSE-CALC-PROGRAM (file: string): calc-program =
  let (*package* = find-package("RU"))
```

```
    parse-from-file(file, 'calc, true)


function TEST-CALC( calc-file-name: string) : calc-program =
  let (prgrm:calc-program = parse-calc-program(calc-file-name))
    prgrm



#||

To parse a CALC file and return a CALC-PROGRAM, type

  .> :eval test-calc("~/calc-project/example.calc")


To make the returned calc-program the current node so you can examine it with
the INSPECTOR, immediately type

  .> (mcn *)

followed by

  .> :inspect-object

||#
```

A.3 Scope Analyzer

```
!! in-package("RU")
!! in-grammar('user)


% ------------------------------------------------------------------
%                 SCOPE ANALYZER FOR CALC LANGUAGE MODEL
% ------------------------------------------------------------------

var REFERENCES: map(identifier, set(identifier-ref))
  computed-using references(@@) = {}

var REF-TO: map(identifier-ref, identifier) = {||}

form DEFINE-REFERENCES-CONVERSE
  define-fun-converses('ref-to, 'references, true)
```

```
%%%  We present three possible solutions for SCOPE-CALC-PROGRAM.  All do
%%%  exactly the same thing so which you use is a matter of programming
%%%  style.  The second and third solutions are enclosed in comments.

function SCOPE-CALC-PROGRAM (program: calc-program) =
  let (idents = descendants-of-class(program, 'identifier),
       ident-refs = descendants-of-class(program, 'identifier-ref))
     ref in ident-refs &
     id in idents &
     identifier-name(id) = identifier-ref-name(ref)
     -->
     ref-to(ref) = id


#||  The most straightforward approach using nested loops

function SCOPE-CALC-PROGRAM (program: calc-program) =
  let (idents = descendants-of-class(program, 'identifier),
       ident-refs = descendants-of-class(program, 'identifier-ref))
    enumerate ref over ident-refs do
      enumerate id over idents do
        if identifier-name(id) = identifier-ref-name(ref) then
          ref-to(ref) <- id
||#

#||  Finally, a more elaborate solution that serves to illustrate
     a few of the more advanced logic and set operators in Refine.

function SCOPE-CALC-PROGRAM (program: calc-program) =
  let (idents = image(variable-identifier, program-declarations(program)),
       ident-refs = descendants-of-class(program, 'identifier-ref))
     ref in ident-refs &
     the-ident = some(id)
                   (id in idents &
                    identifier-name(id) = identifier-ref-name(ref))
     -->
     ref-to(ref) = the-ident
||#

function TEST-CALC( calc-file-name: string) : calc-program =
  let (prgrm:calc-program = parse-calc-program(calc-file-name))
    scope-calc-program(prgrm);
    prgrm
```

A.4 Type Checker

```
!! in-package("RU")
!! in-grammar('user, 'calc)

% ------------------------------------------------------------------
%                     TYPE CHECKER FOR CALC PROGRAMS
% ------------------------------------------------------------------

var DATA-TYPE: map(calc-expression, calc-type) = {||}

var TYPE-ERROR-STRING: map(calc-object, string) = {||}

var *TYPE-RULES* : seq(symbol) =
  [
   'type-check-plus,
   'type-check-minus,
   'type-check-times,
   'type-check-divides,
   'type-check-identifier-ref,
   'type-check-integer-constant,
   'type-check-real-constant,
   'type-check-assignment
  ]

function TYPE-CHECK-CALC-PROGRAM (program: calc-program) =
  postorder-transform(program, *type-rules*, false)

rule TYPE-CHECK-IDENTIFIER-REF (a)
  identifier-ref(a)
  &  undefined?(data-type(a))
  &  id = ref-to(a)  &  defined?(id)
  &  id-type = variable-type(parent-expr(id))
  &  defined?(id-type)
 -->
  data-type(a) = id-type

rule TYPE-CHECK-INTEGER-CONSTANT (a)
  integer-constant(a)
  &  undefined?(data-type(a))
 -->
  data-type(a) = `integer'

rule TYPE-CHECK-REAL-CONSTANT (a)
  real-constant(a)
  &  undefined?(data-type(a))
```

```
 -->
  data-type(a) = 'real'

rule TYPE-CHECK-PLUS (a)
    a = '@e1 + @e2'
    &  undefined?(data-type(a))   % Prevents rule from firing over and over
    &  e1-type = data-type(e1)  & defined?(e1-type)
    &  e2-type = data-type(e2)  & defined?(e2-type)
    &  type-is-real? = (e1-type = 'real' or e2-type='real')
  -->
    data-type(a) = (if type-is-real?
                      then 'real'
                      else 'integer')
rule TYPE-CHECK-MINUS (a)
    a = '@e1 - @e2'
    &  undefined?(data-type(a))   % Prevents rule from firing over and over
    &  e1-type = data-type(e1)  & defined?(e1-type)
    &  e2-type = data-type(e2)  & defined?(e2-type)
    &  type-is-real? = (e1-type = 'real' or e2-type='real')
  -->
    data-type(a) = (if type-is-real?
                      then 'real'
                      else 'integer')

rule TYPE-CHECK-TIMES (a)
    a = '@e1 * @e2'
    &  undefined?(data-type(a))   % Prevents rule from firing over and over
    &  e1-type = data-type(e1)  & defined?(e1-type)
    &  e2-type = data-type(e2)  & defined?(e2-type)
    &  type-is-real? = (e1-type = 'real' or e2-type='real')
  -->
    data-type(a) = (if type-is-real?
                      then 'real'
                      else 'integer')

rule TYPE-CHECK-DIVIDES (a)
    a = '@e1 / @e2'
    &  undefined?(data-type(a))   % Prevents rule from firing over and over
    &  e1-type = data-type(e1)  & defined?(e1-type)
    &  e2-type = data-type(e2)  & defined?(e2-type)
    &  type-is-real? = (e1-type = 'real' or e2-type='real')
  -->
    data-type(a) = (if type-is-real?
                      then 'real'
                      else 'integer')
```

```
rule TYPE-CHECK-ASSIGNMENT (a)
  a = `@lhs := @rhs'
  &  undefined?(type-error-string(a))
  &  lhs-type = data-type(lhs)
  &  rhs-type = data-type(rhs)
  &  ~term-equal?(lhs-type, rhs-type)
 -->
  type-error-string(a) =
   let (*print-grammar-name* = 'calc)
    format(false,
          "Attempt to assign a ~/pp/ into a variable of type ~/pp/ in statement ~/pp/",
          data-type(rhs), data-type(lhs), a)


function TEST-CALC( calc-file-name: string) : calc-program =
  let (prgrm:calc-program = parse-calc-program(calc-file-name))
    scope-calc-program(prgrm);
    type-check-calc-program(prgrm);
    prgrm
```

A.5 Coding Standards Checker

```
!! in-package("RU")
!! in-grammar('user, 'calc)

% ------------------------------------------------------------------
%          CODING STANDARDS CHECKER FOR CALC PROGRAMS
% ------------------------------------------------------------------


var *CODING-STANDARD-RULES* : seq(symbol) =
  [
   'warn-about-undeclared-variable,
   'warn-about-unused-variable,
   'warn-about-type-error-assignment,
   'warn-about-unset-identifier-ref
  ]

var *ALREADY-WARNED-OBJECTS* : set(object) = {}

function CHECK-CALC-PROGRAM (program: calc-program) =
  let (*already-warned-objects* = {})
    preorder-transform(program, *coding-standard-rules*)
```

```
rule WARN-ABOUT-UNDECLARED-VARIABLE (a)
  identifier-ref(a)
  & undefined?(ref-to(a))
  & a ~in *already-warned-objects*
 -->
  a in *already-warned-objects*
  & format(true, "~2&Variable ~s is used but not declared.", identifier-ref-name(a))

rule WARN-ABOUT-UNUSED-VARIABLE (a)
  identifier(a)
  & empty(references(a))
  & a ~in *already-warned-objects*
 -->
  a in *already-warned-objects*
  & format(true, "~2&Variable ~s is declared but never used.", identifier-name(a))

rule WARN-ABOUT-TYPE-ERROR-ASSIGNMENT (a)
  assignment-statement(a)
  & error-message = type-error-string(a)
  & defined?(error-message)
  & a ~in *already-warned-objects*
 -->
  a in *already-warned-objects*
  & format(true, error-message)

rule WARN-ABOUT-UNSET-IDENTIFIER-REF (a)
  identifier-ref(a)
  & a ~in *already-warned-objects*
  & y = least-ancestor-of-class(a, 'calc-statement)
  & ((assignment-statement(y) & a neq assignment-lhs(y))
      or
     ~assignment-statement(y))
  & pgm = least-ancestor-of-class(y, 'calc-program)
  & pgm = `program @@ .. begin $preceding-stmts; @y; .. end'
  & ~ ex(stmt)(stmt in preceding-stmts &
               assignment-statement(stmt) &
               assignment-lhs(stmt) in references(ref-to(a)))
 -->
  a in *already-warned-objects*
  & format(true,
           "~2&identifier-ref  ~/pp/ is used in statement
             ~/pp/ ~%without a preceding assignment statement.",
           a,
           y)
```

```
function TEST-CALC( calc-file-name: string) : calc-program =
  let (prgrm:calc-program = parse-calc-program(calc-file-name))
    scope-calc-program(prgrm);
    type-check-calc-program(prgrm);
    check-calc-program(prgrm);
    prgrm
```

A.6 Expression Simplification

```
!! in-package("RU")
!! in-grammar('user, 'calc)

% ------------------------------------------------------------------------
%                 EXPRESSION SIMPLIFIER FOR CALC PROGRAMS
% ------------------------------------------------------------------------

var *SIMPLIFICATION-RULES*: seq(symbol) =
  [
   'simplify-addition-of-constants,
   'simplify-subtraction-of-constants,
   'simplify-multiplication-of-constants,
   'simplify-division-of-constants,
   'simplify-mult-by-one,
   'simplify-add-by-zero,
   'simplify-x-divided-by-x
  ]

function SIMPLIFY-CALC-PROGRAM (program: calc-program) =
  postorder-transform(program, *simplification-rules*, true)

rule SIMPLIFY-ADDITION-OF-CONSTANTS(a)
   a = `@expr1 + @expr2' &
   integer-constant(expr1) &
   integer-constant(expr2) &
   value-1 = integer-value(expr1) &
   value-2 = integer-value(expr2)
  -->
   integer-constant(a) &
   integer-value(a) = value-1 + value-2

rule SIMPLIFY-SUBTRACTION-OF-CONSTANTS(a)
   a = `@expr1 - @expr2' &
```

```
   integer-constant(expr1) &
   integer-constant(expr2) &
   value-1 = integer-value(expr1) &
   value-2 = integer-value(expr2)
  -->
   integer-constant(a) &
   integer-value(a) = value-1 - value-2

rule SIMPLIFY-MULTIPLICATION-OF-CONSTANTS(a)
   a = `@expr1 * @expr2' &
   integer-constant(expr1) &
   integer-constant(expr2) &
   value-1 = integer-value(expr1) &
   value-2 = integer-value(expr2)
  -->
   integer-constant(a) &
   integer-value(a) = value-1 * value-2

rule SIMPLIFY-DIVISION-OF-CONSTANTS(a)
   a = `@expr1 / @expr2' &
   integer-constant(expr1) &
   integer-constant(expr2) &
   value-1 = integer-value(expr1) &
   value-2 = integer-value(expr2) &
   (value-2 ~= 0 and-then value-2 mod value-1 ~= 0)
  -->
   integer-constant(a) &
   integer-value(a) = value-1 div value-2

rule SIMPLIFY-MULT-BY-ONE(a)
   a = `@expr1 * @expr2' &
   {expr1, expr2} = {`1', x}
  -->
   (replace a by x)

rule SIMPLIFY-ADD-BY-ZERO(a)
   a = `@expr1 + @expr2' &
   {expr1, expr2} = {`0', x}
  -->
   (replace a by x)

rule SIMPLIFY-X-DIVIDED-BY-X (a)
  a = `@expr1 / @expr2' & term-equal?(expr1, expr2)
 -->
  a = `1'
```

```
function TEST-CALC( calc-file-name: string) : calc-program =
  let (prgrm:calc-program = parse-calc-program(calc-file-name))
    scope-calc-program(prgrm);
    type-check-calc-program(prgrm);
    check-calc-program(prgrm);
    simplify-calc-program(prgrm);
    prgrm
```

A.7 CALC Browser

```
!! in-package("RU")
!! in-grammar('user)

% ------------------------------------------------------------------
%                     BROWSER FOR CALC PROGRAMS
% ------------------------------------------------------------------

var *CALC-BROWSER-WINDOW-REGION* : ri::region = ri::make-region(100, 100, 400, 600)

function BROWSE-CALC-PROGRAM (program: calc-program) =
  let (win = make-object('ri::msp-window))
    ri::window-region(win) <- *calc-browser-window-region*;
    ri::window-title(win)  <- "Calc Source Browser";
    ri::window-mouse-button-functions(win) <- [ 'scope-mouse-handler ];
    scope-calc-program(program);
    (let (ri::*mouse-sensitive-printing?*  = true)
        ri::msp-format(win, "~/pp/", program));
    ri::Expose-window(win);
    win

function SCOPE-MOUSE-HANDLER
   (mouse-click: symbol, obj: any-type, pos: ri::point, win: ri::msp-window) =
  if ri::mouse-button-match-general?(mouse-click, 'ri::mouse-left) then
    if identifier-ref(obj)
      then (let (id = ref-to(obj))
              if defined?(id) then highlight-objects-briefly(win, {id}))
    elseif identifier(obj)
      then (let (id-refs = references(obj))
              if defined?(id-refs) then highlight-objects-briefly(win, id-refs))

function HIGHLIGHT-OBJECTS-BRIEFLY (win: ri::msp-window, objs: set(object)) =
```

```
   (let (hls = {hl | (o, hl) o in objs
                              & hl = first(ri::msp-window-object-hulls(win, o))
                              & defined?(hl)})
      (hl in hls --> ri::complement-hull(win, hl));
      lisp::sleep(1);
      (hl in hls --> ri::complement-hull(win, hl)))



function TEST-CALC( calc-file-name: string) : calc-program =
  let (prgrm:calc-program = parse-calc-program(calc-file-name))
    scope-calc-program(prgrm);
    type-check-calc-program(prgrm);
    check-calc-program(prgrm);
    simplify-calc-program(prgrm);
    browse-calc-program(prgrm);
    prgrm
```

B The CALC Language in ASF+SDF

Here we redo the definition of the Calc language ASF+SDF. There are a number of differences:

- As SDF defines concrete and abstract syntax at the same time, there is no need to give an explicit domain model.
- To find the sources of messages reported by the type checker or the coding standards checker, origin tracking is required.
- There is no definition for a browser using SDF. For just syntactic structure browsing, a GSE could be used. To see the identifier def/use information derived by the scope Refine program, some clever use of origin information combined with Module Scope (B.2) would be required (or perhaps the BOX pretty printer generator could be used for this). As yet, this is not implemented in ASF+SDF.
- The Expression simplifier is not written as a function, but directly using rewrite rules. This means that the simplification rules are applied always when a reduction including this module is being done.

B.1 Calc

The Calc module provides abstract and concrete syntax for the Calc language. The distinction between ID and ID-REF is omitted (as it is not needed in the ASF+SDF setting).

The variables declared at the bottom are used in the subsequent checking modules. (But clearly they are not "global" variables as Refine (or Lisp) has – here merely the variable declarations are given).

imports Identifiers$^{B.6.3}$ Reals$^{B.6.5}$ Integers$^{B.6.4}$
exports
sorts EXP EXPS STAT STATS TYPE DECL DECLS
PROG
context-free syntax

REAL-CON	→ EXP	
INT-CON	→ EXP	
ID-CON	→ EXP	
EXP "+" EXP	→ EXP	{**left**}
EXP "−" EXP	→ EXP	{**left**}
EXP "∗" EXP	→ EXP	{**left**}
EXP "/" EXP	→ EXP	{**left**}
"(" EXP ")"	→ EXP	{**bracket**}
{EXP ","}+	→ EXPS	

ID-CON ":=" EXP → STAT
print "(" EXPS ")" → STAT

var ID-CON ":" TYPE → DECL
real → TYPE
integer → TYPE

{DECL ";"}* → DECLS
{STAT ";"}* → STATS
program ID-CON DECLS begin STATS end → PROG

priorities

{**left**: EXP "+"EXP → EXP, EXP "−"EXP → EXP} < {**left**: EXP "*"EXP → EXP, EXP "/"EXP → EXP}

variables

E [*0-9*]* → EXP
E [*0-9*]*"+" → {EXP ","}+
Es [*0-9*]* → EXPS
S [*0-9*]* → STAT
S [*0-9*]*"+" → {STAT ";"}+
S [*0-9*]*"*" → {STAT ";"}*
Ss [*0-9*]* → STATS
D [*0-9*]* → DECL
Decls [*0-9*]* → DECLS
D [*0-9*]*"*" → {DECL ";"}*
D [*0-9*]*"+" → {DECL ";"}+
T [*0-9*]* → TYPE

B.2 Scope

This module finds used, assigned, and declared identifiers in parts of CALC programs. In Refine, shorter expressions can be written to achieve the same-effect, using the `descendants-of-class(Tree, Class)` function, which finds all subtrees of the given sort in a given tree.

Observe that this module provides functionality rather different from the Refine `scope` program.

imports Calc[B.1] Booleans[B.6.2]

exports

sorts IDS

context-free syntax

"{" {ID-CON ","}* "}" → IDS
ID-CON ∈ IDS → BOOL
IDS ∪ IDS → IDS

context-free syntax

used-ids(STATS) → IDS
assigned-ids(STATS) → IDS
declared-ids(DECLS) → IDS

hiddens

context-free syntax

used-exp-ids(EXPS) → IDS

variables

Id [*0-9*]*"*" → {ID-CON ","}*

equations

Extract all used identifiers

[i0] used-ids(S_1^+; S_2^+) = used-ids(S_1^+) ∪ used-ids(S_2^+)
[i1] used-ids(*Id-Con* := *E*) = used-exp-ids(*E*)
[i2] used-ids(print(*Es*)) = used-exp-ids(*Es*)

[i3] used-exp-ids(*Id-Con*) = {*Id-Con*}
[i4] used-exp-ids(E_1 + E_2) = used-exp-ids(E_1) ∪ used-exp-ids(E_2)
[i5] used-exp-ids(E_1 * E_2) = used-exp-ids(E_1) ∪ used-exp-ids(E_2)
[i6] used-exp-ids(E_1 − E_2) = used-exp-ids(E_1) ∪ used-exp-ids(E_2)
[i7] used-exp-ids(E_1 / E_2) = used-exp-ids(E_1) ∪ used-exp-ids(E_2)
[i8] used-exp-ids(E_1^+, E_2^+) = used-exp-ids(E_1^+) ∪ used-exp-ids(E_2^+)
[i9] used-exp-ids(*E*) = {} **otherwise**

Extract all assigned identifiers

[a1] assigned-ids(S_1^+; S_2^+) = assigned-ids(S_1^+) ∪ assigned-ids(S_2^+)
[a2] assigned-ids(*Id-Con* := *E*) = {*Id-Con*}
[a3] assigned-ids(*Ss*) = {} **otherwise**

Extract all declared identifiers

[d1] declared-ids() = {}
[d2] declared-ids(D_1^+; D_2^+) = declared-ids(D_1^+) ∪ declared-ids(D_2^+)
[d3] declared-ids(var *Id-Con* : *T*) = {*Id-Con*}

Set Operations

[s0] {Id_1^*, *Id-Con*, Id_2^*, *Id-Con*, Id_3^*} = {Id_1^*, *Id-Con*, Id_2^*, Id_3^*}
[s1] {Id_1^*} ∪ {Id_2^*} = {Id_1^*, Id_2^*}
[s2] *Id-Con* ∈ {Id_1^*, *Id-Con*, Id_2^*} = true
[s3] *Id-Con* ∈ {Id^*} = false **otherwise**

B.3 Type-Checker

The type checker defines the types of expressions, and detects assignments in which the lhs and rhs are of of different types. Observe that we have to use an explicit program traversal using the functions tc-prog and tc-stats; In Refine the traversal is done using the expression `postorder-transform(Tree, Set-of-Rule-Ids)` which applies all rules in the second argument to the first tree argument.

The auxiliary function 'max' returns the highest of two types, usually real, unless both argument types are integer. The Refine definition does not use this function, but it would have been a little shorter if it had.

imports Calc$^{B.1}$ Booleans$^{B.6.2}$
exports
sorts ERROR ERRORS
context-free syntax
"Attempt" to assign a TYPE
into a variable of type TYPE in statement STAT "." → ERROR
"[" ERROR∗ "]" → ERRORS
ERRORS "++" ERRORS → ERRORS
exports
context-free syntax
data-type(EXP, DECLS) → TYPE
tc-prog(PROG) → ERRORS
hiddens
context-free syntax
tc-stats(STATS, DECLS) → ERRORS
max(TYPE, TYPE) → TYPE
compatible(TYPE, TYPE) → BOOL
variables
E [*0-9*]∗"∗" → ERROR∗
Ds [*0-9*]∗ → DECLS
equations

Extract the type of an expression

[t1] data-type(*Real-Con*, *Ds*) = real
[t2] data-type(*Int-Con*, *Ds*) = integer
[t3] data-type(*Id-Con*, D_1^*; var *Id-Con* : *T*; D_2^*) = *T*

[t4] data-type(E_1 + E_2, *Ds*) = max(data-type(E_1, *Ds*), data-type(E_2, *Ds*))
[t5] data-type(E_1 − E_2, *Ds*) = max(data-type(E_1, *Ds*), data-type(E_2, *Ds*))
[t6] data-type(E_1 ∗ E_2, *Ds*) = max(data-type(E_1, *Ds*), data-type(E_2, *Ds*))
[t7] data-type(E_1 / E_2, *Ds*) = max(data-type(E_1, *Ds*), data-type(E_2, *Ds*))

Subtype ordering for reals and integers

[m1] max(real, T) = real
[m2] max(T, real) = real
[m3] max(integer, integer) = integer

[m4] compatible(real, real) = true
[m5] compatible(integer, integer) = true
[m6] compatible(T_1, T_2) = false **otherwise**

Typecheck an entire program

[p1] tc-prog(program *Id-Con Ds* begin *Ss* end) = tc-stats(*Ss*, *Ds*)
[p2] tc-stats(, *Ds*) = []
[p3] tc-stats(S; S^+, *Ds*) = tc-stats(S, *Ds*) ++ tc-stats(S^+, *Ds*)

[p4] $$\frac{\text{compatible(data-type(}\textit{Id-Con}\text{, }\textit{Ds}\text{), data-type(}E\text{, }\textit{Ds}\text{))} = \text{false}}{\begin{array}{l}\text{tc-stats(}\textit{Id-Con} := E\text{, }\textit{Ds}\text{)} = \\ \text{[Attempt to assign a data-type(}\textit{Id-Con}\text{, }\textit{Ds}\text{)} \\ \text{ into a variable of type data-type(}E\text{, }\textit{Ds}\text{) in statement } \textit{Id-Con} := E\text{ .]}\end{array}}$$

[p5] tc-stats(S, *Ds*) = []
otherwise

Error Messages

[e1] $[E_1^*] +\!\!+ [E_2^*] = [E_1^*\ E_2^*]$

B.4 Coding-Standards

Again, for the coding standards ASF+SDF requires explicit program traversal; Refine uses a `preorder-transform(Tree, Set-of-Rules)` to do the program traversal.

On the other hand; the Refine solution also requires a global variable, `*already-warned-objects*` to avoid generating messages more than once.

imports Calc$^{B.1}$ Type-Checker$^{B.3}$ Scope$^{B.2}$
exports
context-free syntax
coding-standards(PROG) → ERRORS

"Variable" ID-CON is used but not "declared." → ERROR
"Variable" ID-CON is declared but never "used." → ERROR
"Variable" ID-CON is used
without a preceding assignment "statement." → ERROR

hiddens

context-free syntax

code-stats(STATS, IDS) → ERRORS
code-decls(DECLS, IDS) → ERRORS
code-exps(EXPS, IDS, IDS) → ERRORS

variables

Declared-Ids → IDS
Assigned-Ids → IDS
Ds → IDS
As → IDS
Used-Ids → IDS

equations

Entire Program

[c1] coding-standards(program *Id-Con Decls* begin *Ss* end)
= code-decls(*Decls*, used-ids(*Ss*) ∪ assigned-ids(*Ss*))
++ code-stats(*Ss*, declared-ids(*Decls*))

Statements

[s1] code-stats(, *Ds*) = []

[s2] code-stats(S^*; print(*Es*), *Declared-Ids*)
= code-stats(S^*, *Declared-Ids*) ++ code-exps(*Es*, *Declared-Ids*, assigned-ids(S^*))

[s2] code-stats(S^*; *Id-Con* := *E*, *Declared-Ids*)
= code-stats(S^*, *Declared-Ids*) ++ code-exps(*E*, *Declared-Ids*, assigned-ids(S^*))

Expressions

[e1] code-exps(E_1^+, E_2^+, *Ds*, *As*)
= code-exps(E_1^+, *Ds*, *As*) ++ code-exps(E_2^+, *Ds*, *As*)

[e2] $$\frac{\textit{Id-Con} \in \textit{Declared-Ids} \;=\; \text{false}}{\text{code-exps}(\textit{Id-Con}, \textit{Declared-Ids}, \textit{As}) \;=\; [\text{Variable } \textit{Id-Con} \text{ is used but not declared.}]}$$

[e3] $$\frac{\textit{Id-Con} \in \textit{Assigned-Ids} \;=\; \text{false}}{\text{code-exps}(\textit{Id-Con}, \textit{Ds}, \textit{Assigned-Ids}) \;=\; [\text{Variable } \textit{Id-Con} \text{ is used without a preceding assignment statement.}]}$$

[e4] $\text{code-exps}(E_1 + E_2, \textit{Ds}, \textit{As}) = \text{code-exps}(E_1, \textit{Ds}, \textit{As}) +\!\!+ \text{code-exps}(E_2, \textit{Ds}, \textit{As})$
[e5] $\text{code-exps}(E_1 * E_2, \textit{Ds}, \textit{As}) = \text{code-exps}(E_1, \textit{Ds}, \textit{As}) +\!\!+ \text{code-exps}(E_2, \textit{Ds}, \textit{As})$
[e6] $\text{code-exps}(E_1 - E_2, \textit{Ds}, \textit{As}) = \text{code-exps}(E_1, \textit{Ds}, \textit{As}) +\!\!+ \text{code-exps}(E_2, \textit{Ds}, \textit{As})$
[e7] $\text{code-exps}(E_1 \,/\, E_2, \textit{Ds}, \textit{As}) = \text{code-exps}(E_1, \textit{Ds}, \textit{As}) +\!\!+ \text{code-exps}(E_2, \textit{Ds}, \textit{As})$

[e8] $\text{code-exps}(E, \textit{Ds}, \textit{As}) = [\,]$ **otherwise**

Declarations

[d1] $\text{code-decls}(D_1^+; D_2^+, \textit{Used-Ids}) = \text{code-decls}(D_1^+, \textit{Used-Ids}) +\!\!+ \text{code-decls}(D_2^+, \textit{Used-Ids})$

[d2] $\text{code-decls}(, \textit{Used-Ids}) = [\,]$

[d3] $$\frac{\textit{Id-Con} \in \textit{Used-Ids} \;=\; \text{false}}{\text{code-decls}(\text{var } \textit{Id-Con} : T, \textit{Used-Ids}) \;=\; [\text{Variable } \textit{Id-Con} \text{ is declared but never used.}]}$$

[d4] $\text{code-decls}(D, \textit{Used-Ids}) = [\,]$ **otherwise**

B.5 Simplify

imports Calc$^{B.1}$ Numbers$^{B.6.6}$
hiddens
variables
X [*0-9*]$* \to$ NUM
equations

[s1] $E * 1 = E$
[s2] $1 * E = E$
[s3] $E + 0 = E$
[s4] $0 + E = E$
[s5] $E \,/\, E = 1$
[s6] $E \,/\, 1 = E$

The equations below map the "+" over expressions to the "+" over naturals. The when takes care of "re-tracting" a natural X injected into an expression E into just the natural X.

[b1] $E_1 + E_2 = X_1 + X_2$ **when** $E_1 = X_1,\ E_2 = X_2$
[b2] $E_1 * E_2 = X_1 * X_2$ **when** $E_1 = X_1,\ E_2 = X_2$
[b3] $E_1 - E_2 = X_1 - X_2$ **when** $E_1 = X_1,\ E_2 = X_2$
[b4] $E_1 \ / \ E_2 = X_1 \ / \ X_2$ **when** $E_1 = X_1,\ E_2 = X_2$

B.6 Standard Modules

B.6.1 Layout

This module defines "white space" used in the Calc language.
exports
lexical syntax
[␣\t\n] → LAYOUT
"%%"~[\n]* → LAYOUT

B.6.2 Booleans

imports Layout$^{B.6.1}$
exports
sorts BOOL
context-free syntax
true → BOOL
false → BOOL
BOOL or BOOL → BOOL {**left**}
BOOL and BOOL → BOOL {**left**}
not BOOL → BOOL
"(" BOOL ")" → BOOL {**bracket**}
priorities
not > and > or

hiddens
variables
p['] * → BOOL
equations

[1] p or true = true
[2] p or false = p

[3] p and false = false
[4] p and true = p
[5] not true = false
[6] not false = true

B.6.3 Identifiers

imports Layout$^{B.6.1}$
exports
sorts ID-CON
lexical syntax
[a-z][a-z0-9\−]* → ID-CON
variables
Id-Con [*0-9*]* → ID-CON

B.6.4 Integers

imports Numbers$^{B.6.6}$
exports
sorts INT-CON
context-free syntax
"−" NUM → INT-CON
NUM → INT-CON
NAT "*" NAT → NAT
NAT "/" NAT → NAT
variables
Int-Con [*0-9*]* → INT-CON

B.6.5 Reals

imports Numbers$^{B.6.6}$
exports
sorts REAL-CON
lexical syntax
NUM "."NUM → REAL-CON
exports
variables
Real-Con [*0-9*]* → REAL-CON

B.6.6 Numbers

The standard module for numberic operations is not shown.

Contents