*TRAP*ping Modelica with Python

Thilo Ernst

GMD FIRST Research Institute for Computer Architecture
and Software Technology
Rudower Chaussee 5, D-12489 Berlin, Germany
Thilo.Ernst@gmd.de

Abstract. This short paper[1] introduces *TRAP*, a small but powerful compiler development system based on the object-oriented, dynamic language Python. Employing a very high level language as a compiler tool's base language reduces the need for additional tool support and importing library functionality to a minimum. Python in particular has the additional advantage of being a powerful and already quite popular general-purpose component integration framework, which can be utilized both for incorporating subcomponents and for embedding the compiler developed into a larger system. Exploiting these strengths, *TRAP* enables rapid prototyping and development of compilers - in particular, translators for medium-complexity special purpose languages - on a very high level of abstraction.

1 Starting Point: Modelica, SMILE, and Python

Modelica[1] is a unified, object-oriented description language for dynamic models of complex physical systems being developed in an international effort. GMD is developing an experimental Modelica compiler for integration into the SMILE dynamic simulation environment[2] which in its latest revision heavily builds on the object-oriented dynamic language Python[3] as an open component integration platform. For that reason the Modelica compiler, too, was to be integrated using a Python interface. This situation triggered a closer investigation of the idea of implementing the compiler directly in Python. The language turned out to be very suitable for the purpose; in addition basic compiler construction tool components in Python were already available. As the extra effort needed was limited, the approach taken was to integrate these components and fill the remaining gaps to obtain a Python-based compiler construction system called *TRAP*, based on which the Modelica compiler is implemented.

2 Python as a Compiler Implementation Language

Python has many features using which common compiler problem patterns can be approached effectively. These problems would require considerable coding,

[1] A long version is available from the author on request

S. Jähnichen (Ed.): CC'99, LNCS 1575, pp. 288–292, 1999.

debugging and maintenance efforts using standard implementation languages, whereas compiler tool support is often insufficient, overspecialized, or handicapped by unrelated tool limitations. The following Python features are most relevant:

Lists and tuples, i.e. (mutable and immutable) *sequence types* with their built-in operations are useful to express simple, incremental set-manipulation based algorithms which are ubiquitous in compilers.

Dictionaries represent *mappings* between sets of Python objects, and perfectly model compiler concepts like "symbol table" or "name space".

Python's Object model is well-suited for modeling abstract syntax tree- or graph-shaped *internal representations* (IRs) , i.e., *node types* ordered into a hierarchy. The language's dynamic type system can be easily exploited to enforce local IR wellformedness constraints. Generally, the language enables the application of modern, object-oriented compiler implementation techniques.

Functional *and* imperative programming. Python's object-reference passing semantics together with the sequence types already mentioned allows the user to mix functional and imperative programming as appropriate for the task at hand.

Automatic memory management saves the developer much trouble as extensive manipulation of recursive data structures is central in compilers.

Of course there are other very high level languages offering comparably well-suited, or even richer feature sets. However, compilers (in particular those for special-purpose languages, like in the SMILE/Modelica case) often need to be embedded into larger software systems, at which point aspects like ease of integration, portability, ease of reuse, readability, ease of learning, popularity, availability of library modules and development tools, and development productivity become at least equally important as the language's internal qualities. Being admittedly prepossessed, we still find Python to represent a pretty unique balance among this interdependent (and partly conflicting) set of goals.

3 Designing *TRAP*

Compiler construction tools offer a set of more or less compiler-specific programming abstractions in the form of dedicated *compiler description language* constructs. From the description, the tool generates the actual compiler in some *implementation* or *base language*. As each tool's functionality has its limits, escape mechanisms to the base language are usually offered (and tend to be heavily used in nontrivial compilers). Unfortunately, this language often turns out to be inadequate, i.e., too low-level, for the problem that required the escape; C is employed frequently in this role. Each concrete instance of this problem can be "solved" by putting in additional coding effort, integrating suitable libraries, etc. However apart from the immediate cost, this can increase the complexity of

the compiler as a software system to an extent not justified by the purpose and severely damage its readability and maintainability.

To generally avoid this problem while at the same time reduce the functionality to be added by the compiler tool to a necessary minimum, we propose to use a very high level language in the role of the base language. As motivated in the previous section, we found Python to be an excellent candidate.

The main (technical) problem areas in compiler development, which should be well supported by any compiler tool or set thereof, are: construction of the front-end (scanner and parser), definition of the internal IR data structure, and description of the transformation algorithms working on the IR.

With Python as the implementation language, we found that a new generative tool would only be needed to support the definition of IR data structures: Front-end generation tools are available and can be reused easily. W.r.t. transformation algorithms, we found Python's own expressiveness (enhanced by a set of generated IR manipulation methods) sufficient.

Based on these observations, a compiler description language consisting of a grammar description part and an IR description part was designed, and a Python-based compiler tool *TRAP* (Translator RApid Prototyping) was developed, processing it as follows: From the grammar description part, a scanner and parser module is generated using an LR parser generator[2]. From the IR description part, an IR module is generated. The resulting compiler consists of these generated Python modules and an arbitrary number of transformation and auxilliary modules directly written in or interfaced with Python. In more detail, *TRAP* has the following main characteristics:

Integrated syntax/semantics description. An EBNF-like grammar description language is offered which tightly integrates concrete syntax (including lexics), the construction of initial semantics values., and the specification of type constraints to the latter. The computation of a semantics value is expressed as Python code right in the corresponding grammar rule. Where no semantics action is specified, an appropriate default semantics takes effect. EBNF-style repetition constructs are supported, with an automatic semantics constructing Python sequences. Semantically insignificant separators and terminators can be specified here, too. A sample nonterminal definition with one rule (including an explicit semantics action):

```
nterm print_stmt::STMT     # type constraint: node type
     <- "print" ["," expr *]=eList:    # comma-sep. list
         Print_stmt(eList)      # invoke node constructor
```

Hierarchical IR description language. The IR description allows to concisely describe IR data structure patterns (i.e., node sorts with typed fields) ordered into a type hierarchy, very similar to *ast* [7]. From this description, a set of Python class definitions to be used in the compiler being developed is generated. These classes also contain generated methods for constructing,

[2] currently, A.Watters' kwParsing package (`http://www.chordate.com/kwParsing`)

typechecking, printing, traversing, and matching IR subgraphs, to be used in initial semantics actions as well as in transformation modules.

"No" transformation language. Transformations and other IR processing algorithms are expressed directly in Python, relying on generated methods in the IR module as needed. Python is flexible enough to express important compiler-specific concepts without syntactic extensions; for example, Pythons '^' operator was overloaded to represent pattern matching when applied to IR objects, with patterns expressed by nested node constructors, and '_' acting as a wildcard. Thus, for transformations, there is no separate description level, no interfacing problems between both levels, and no generation step in each (transformation) development cycle (only IR definition changes require re-generation).

Rapid prototyping/development (RP/RAD). Python's power as a component integration framework and its efficient development methodology are applicable to the domain of compiler construction. It is easy to replace a compiler component developed in Python by an "extension", i.e., an optimized low-level implementation. For instance, a graph algorithm is prototyped as a Python class and later on substituted by a bitset-based C implementation equipped with the same Python class interface, without a need for modifying its "client code".

The design of *TRAP*'s compiler description language was inspired heavily by concepts from and/or experience with the following tools: Gentle[4], CoSy[5], Smart[6], Cocktail[7], Depot[8], PCCTS[9], and JAMOOS[10]. Its "look and feel" attempts to stay close to the base language.

4 Conclusions

TRAP, a Python-based compiler construction tool was presented. *TRAP* largely integrates successful concepts from other compiler tools. Its main contribution is their combination with a very high level , object-oriented implementation language. *TRAP* was succesfully applied for generating compilers for two nontrivial languages: *TRAP*'s own compiler description language, and Modelica, the latter being already a medium-sized example. The RP/RAD methodology was proven feasible by integrating a fast, handwritten Modelica lexer subcomponent.

Its characteristics should make the *TRAP* system well-suited for a broad range of compiler prototyping and development tasks, in particular for special-purpose languages of small-to-medium complexity. Its high level of abstraction and its platform-independent availability could make it attractive also for educational purposes.

References

1. The Modelica Design Group: *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling*. http://www.dynasim.se/Modelica, 1997 288
2. Ernst, T.; Jähnichen, S; Klose, M.: *The Architecture of the* SMILE*/M Simulation Environment*. Proc. 15^{th} IMACS World Congress on Sci. Comp., Berlin, 1997 288
3. van Rossum, G.: *Python Reference Manuals*. http://www/python.org/doc 288
4. Schröer, F.W.: *The Gentle Compiler Construction System*. GMD Bericht 290, 1997 291
5. Alt, M.; Assmann, U.; van Someren, H.: *CoSy Compiler Phase Embedding with the CoSy Compiler Model*. LNCS 786, Springer 1994 291
6. Schröer, F.W.: *Smart Reference Manual (Draft)*. Internal report, GMD, 1993. 291
7. Grosch, J.: *A Tool Box for Compiler Construction*. LNCS 477, pp.106-116, Springer 1990 290, 291
8. Lampe, J: *An Extensible Translator-Generator for Use in Branch Software Construction*. J. Comp. Inf. 2(1996)1, pp.1057-1067 291
9. Parr, T.J.: *Language Translation Using PCCTS and C++*. Automata, 1997. 291
10. Gil, J.: JAMOOS Page. http://oop.cs.technion.ac.il/236704-5/JAMOOS/, 1997 291